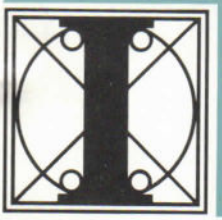
LSX Computer Line (Up to Model 3040)

Operating Systems

X/OS UNIX® System V-based Operating System
System Interfaces and Libraries

Reference Manual

X/OS



olivetti

PUBLICATION ISSUED BY:

Ing. C. Olivetti & C., S.p.A.
Direzione Documentazione
77, Via Jervis - 10015 Ivrea (Italy)

Information from Olivetti Documentation

LSX Computer Line (Up to Model 3040)

Operating Systems

X/OS UNIX® System V-based Operating System
System Interfaces and Libraries

Reference Manual

olivetti

PREFACE

This is a reference manual for system and application programmers, and describes the programming features of the X/OS operating system.

SUMMARY

This manual is divided into three main parts:

- the first part contains the operating system calls - section 2.
- the second part contains the system file formats - section 4.
- the third part contains miscellaneous facilities - section 5.

A list of contents and a permuted index precede these sections.

Subroutines belonging to section 3 are described in the *C Language Programming Manual*, apart from those in subsection 3F, which are described in the *FORTRAN 77 Programming Manual*.

REFERENCES

Read first ...

X/OS User Manual - Code 4043610 C

X/OS Utilities Reference Manual - Code 4041460 V

X/OS Programmer Guide - Code 4051750 T

For further information read ...

C Language Programming Manual - Code 4041520 T

FORTRAN 77 Programming Manual - Code 4043390 E

Common Development Tools User Manual -
Code 4040400 A

DISTRIBUTION: As part of Software Kit (W)

FIRST EDITION: December 1987 - X/OS Release 1.0

This manual describes system calls, file formats, and other programming-related features of the X/OS operating system. Each description takes the form of a formal reference guide to one or more items. The facilities described in this manual are intended for use by programmers.

The section number, which is appended to the title of each item, acts as a guide to the type of facility described. Those in section 2 are X/OS operating system calls, those in section 4 are file formats, while section 5 contains miscellaneous facilities.

Some descriptions include more than one item. For example, the pages describing the *stat* system call also cover *lstat* and *fstat*. To locate an item such as *lstat* that is not listed under its own name in the Table of Contents, use the Permuted Index.

Each reference uses the same format, which is summarised below. Note that not all of the following sections are always present for each command.

NAME gives the name or names of the items described. A brief one-line description is given to indicate their purpose.

SYNOPSIS summarises the format in which the item is invoked, giving mandatory elements and options. The conventions used are as follows:

- **Boldface** text represents literal strings, and should be typed exactly as shown.
- Normal text identifies substitutable arguments. For example, the word "file" or "name" indicates that a filename should be given.
- [ ] Square brackets indicate optional arguments.
- ... Ellipses indicate that the previous argument may be repeated.

DESCRIPTION describes the nature and function of the item.

EXAMPLES gives one or more examples of usage.

FILES lists relevant filenames.

SEE ALSO gives references to related material.

DIAGNOSTICS illustrates the diagnostic systems, if any, built in to the item. Messages that are self-explanatory are not listed.

WARNINGS points to possible problem areas.

BUGS indicates some restrictions to the use of the item. A bypass may be indicated.

TABLE OF CONTENTS

The X/OS system provides all of the X/OPEN services. It also provides various additional services. The services that conform to the X/OPEN standard are marked with an asterisk in the table of contents.

PERMUTED INDEX

```
          /etc/mtab - mounted file system table.....        MTAB(4)
        fs, inode - format of file system volume.....          FS(4)
common object files..... filehdr - file header for....     FILEHDR(4)
      header for common object files.....filehdr - file    FILEHDR(4)
         specification in text files.....fspec - format      FSPEC(4)
   gets directory entries in a filesystem-independent/..../-GETDIRENTRIES(2)
    - static information about filesystems.....fstab, mntent     FSTAB(4)
                              fork - create a new process..      FORK(2)
   per-process accounting file format.............acct -       ACCT(4)
           - incremental dump format........dump, dumpdates     DUMP(4)
   in a filesystem-independent format...../directory entriesGETDIRENTRIES(2)
      object file symbol table format.....syms - common       SYMS(4)
       tar - tape archive file format.....                     TAR(4)
                     inode - format of an inode.....          INODE(4)
file.....term - format of compiled terminfo..                 TERM(4)
                      cpio - format of cpio archive.....      CPIO(4)
                       dir - format of directories.....       DIR(4)
volume.....fs, inode - format of file system.....             FS(4)
                      core - format of memory image file..    CORE(4)
library/.....shlib - format of the shared.....                SHLIB(4)
text files.....fspec - format specification in.....          FSPEC(4)
        - introduction to file formats.....intro              INTRO(4)
    wtmp - utmp and wtmp entry formats.....utmp,              UTMP(4)
system volume..... fs, inode - format of file...               FS(4)
specification in text/..... fspec - format.....              FSPEC(4)
information about/..... fstab, mntent - static.....         FSTAB(4)
                  stat, lstat, fstat - get file status.....   STAT(2)
        /- shut down part of a full-duplex connection.....  SHUTDOWN(2)
        prof - profile within a function.....                 PROF(5)
                  math - math functions and constants.....    MATH(5)
      getsockopt, setsockopt - get and set options on/.....  GETSOCKOPT(2)
                    ulimit - get and set user limits.....     ULIMIT(2)
              umask - set and get file creation mask.....     UMASK(2)
         stat, lstat, fstat - get file status.............     STAT(2)
                    statfs - get file system statistics...    STATFS(2)
                     ustat - get file system statistics...    USTAT(2)
resource/.....getrusage - get information about.....      GETRUSAGE(2)
                    msgget - get message queue.....          MSGGET(2)
                getpeername - get name of connected peer... GETPEERNAME(2)
operating system.....uname - get name of current.....         UNAME(2)
process times.....times - get process and child.....          TIMES(2)
```

wait3 - wait for child process to stop or/.....wait, WAIT(2)
ptrace - process trace..... PTRACE(2)
pause - suspend process until signal..... PAUSE(2)
to a process or a group of processes..../- send a signal KILL(2)
- reboot system or halt processor.....reboot REBOOT(2)
alarm - set a process's alarm clock..... ALARM(2)
function..... prof - profile within a..... PROF(5)
profile..... profil - execution time..... PROFIL(2)
profil - execution time profile..... PROFIL(2)
environment at login time..... profile - setting up an..... PROFILE(4)
prof - profile within a function.... PROF(5)
/setpriority - get/set program scheduling priority.. GETPRIORITY(2)
ptrace - process trace..... PTRACE(2)
msgget - get message queue..... MSGGET(2)
read, readv - read from file..... READ(2)
file..... read, readv - read from..... READ(2)
link.....readlink - read value of a symbolic..... READLINK(2)
open - open for reading or writing..... OPEN(2)
symbolic link..... readlink - read value of a... READLINK(2)
read, readv - read from file..... READ(2)
lseek - move read/write file pointer..... LSEEK(2)
setregid - set real and effective group ID.. SETREGID(2)
ID's.....setreuid - set real and effective user..... SETREUID(2)
/real user, effective user, real group, and effective/... GETUID(2)
real/.../getgid, getegid - get real user, effective user,... GETUID(2)
halt processor..... reboot - reboot system or.... REBOOT(2)
processor.....reboot - reboot system or halt..... REBOOT(2)
- specify what to do upon receipt of a signal....signal SIGNAL(2)
recv, recvfrom, recvmsg - receive a message from a/.... RECV(2)
receive a message from a/..... recv, recvfrom, recvmsg -.... RECV(2)
a message from a/.....recv, recvfrom, recvmsg - receive.. RECV(2)
from a/.....recv, recvfrom, recvmsg - receive a message.. RECV(2)
compile and match routines.... regexp - regular expression.. REGEXP(5)
and match/.....regexp - regular expression compile... REGEXP(5)
wait/....sigpause - atomically release blocked signals and.. SIGPAUSE(2)
information for a common/..... reloc - relocation..... RELOC(4)
a common object/.....reloc - relocation information for... RELOC(4)
rmdir - remove a directory file..... RMDIR(2)
unlink - remove directory entry..... UNLINK(2)
mount, umount - mount or remove file system..... MOUNT(2)
a file..... rename - change the name of.. RENAME(2)

```
IDs.....................setuid, setgid - set user and group..      SETUID(2)
identifier of/......gethostid, sethostid - get/set unique...   GETHOSTID(2)
of current/.......gethostname, sethostname - get/set name... GETHOSTNAME(2)
of interval/........getitimer, setitimer - get/set value....   GETITIMER(2)
ID.......................... setpgrp - set process group..      SETPGRP(2)
program/..........getpriority, setpriority - get/set........ GETPRIORITY(2)
effective group ID............ setregid - set real and......    SETREGID(2)
effective user ID's........... setreuid - set real and......    SETREUID(2)
system resource/....getrlimit, setrlimit - control maximum..   GETRLIMIT(2)
options on/........getsockopt, setsockopt - get and set.....  GETSOCKOPT(2)
and time.........gettimeofday, settimeofday - get/set date..GETTIMEOFDAY(2)
at login time........profile - setting up an environment....     PROFILE(4)
         /- speed and terminal settings used by getty.......   GETTYDEFS(4)
and group IDs................. setuid, setgid - set user....      SETUID(2)
file.....shlib - format of the shared library description...      SHLIB(4)
operations............shmctl - shared memory control........     SHMCTL(2)
                shmat, shmdt - shared memory operations.....      SHMOP(2)
                  shmget - get shared memory segment........     SHMGET(2)
shared library description/... shlib - format of the........      SHLIB(4)
memory operations............. shmat, shmdt - shared........      SHMOP(2)
control operations............ shmctl - shared memory.......     SHMCTL(2)
operations..............shmat, shmdt - shared memory........      SHMOP(2)
segment....................... shmget - get shared memory...     SHMGET(2)
full-duplex/........shutdown - shut down part of a.........   SHUTDOWN(2)
of a full-duplex/............. shutdown - shut down part....   SHUTDOWN(2)
                               sigblock - block signals.....   SIGBLOCK(2)
        - suspend process until signal..................pause      PAUSE(2)
        to do upon receipt of a signal......../- specify what     SIGNAL(2)
upon receipt of a signal...... signal - specify what to do..     SIGNAL(2)
              sigvec - software signal facilities.............    SIGVEC(2)
      sigsetmask - set current signal mask.................. SIGSETMASK(2)
     sigstack - set and/or get signal stack context.........   SIGSTACK(2)
group of/........kill - send a signal to a process or a.....       KILL(2)
              sigblock - block signals.......................   SIGBLOCK(2)
    atomically release blocked signals and wait for/....../-   SIGPAUSE(2)
release blocked signals/...... sigpause - atomically........   SIGPAUSE(2)
signal mask................... sigsetmask - set current.....  SIGSETMASK(2)
signal stack context.......... sigstack - set and/or get....   SIGSTACK(2)
facilities.................... sigvec - software signal.....    SIGVEC(2)
              - get system page size..............getpagesize GETPAGESIZE(2)
    - accept a connection on a socket................accept     ACCEPT(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; 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 description of a system call 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* or *ptrace*.

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 5,120 bytes is presented to a member of the *exec* family.

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
One of the following:

- a file descriptor refers to a file which is not open
- a *read* request is made to a file which is open only for writing
- a *write* request is made to a file which is open only for 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 processes
A *fork* failed because the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough space
During an *exec*, *brk*, or *sbrk*, a program asks for more space than the system is able to supply. This is not 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.

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*.

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*.

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*.

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*, or *kill*; reading or writing a file for which *lseek* has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.

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
The calling process has exceeded the system-defined limit on the number of files open at one time. When a record lock is being created with *fcntl*, there are too many files with record locks on them.

25 ENOTTY Not a character device
An attempt was made to *ioctl* 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 a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum file size (1,082,201,088 bytes) or ULIMIT; see *ulimit*(2).

28 ENOSPC No space left on device
During a write to an ordinary file, there is no free space left on the device. In *fcntl*, 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* 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 (1000) to a file was made.

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)].

45 EDEADLK Deadlock
A deadlock situation was detected and avoided.

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

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

101 EINPROGRESS Operation now in progress
An operation which takes a long time to complete (such as a *connect* (2)) was attempted on a non-blocking object (see *ioctl* (2)).

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

103 ENOTSOCK Socket operation on non-socket
An attempt has been made to call one of the functions which handle sockets, but specifying a non-existent or invalid socket.

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

105 EMSGSIZE Message too long
A message sent on a socket was larger than the maximum transmission unit size.

106 EPROTOTYPE Protocol wrong type for socket
A protocol was specified which does not support the semantics of the socket type requested. For example, it is not permitted to use the ARPA Internet UDP protocol with type SOCK_STREAM.

107 ENOPROTOOPT Bad protocol option
A bad option was specified in a *getsockopt*(2) or *setsockopt*(2) call.

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

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

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

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

112 EAFNOSUPPORT Address family not supported by protocol family
An address incompatible with the requested protocol was used. For example, one shouldn't necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

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

114 EADDRNOTAVAIL Can't assign requested address
This normally results from an attempt to create a socket with an address not on this machine.

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

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

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

118 ECONNABORTED Software caused connection abort
Your function call caused a connection abort on your host machine.

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

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

121 EISCONN Socket is already connected

A *connect* request was made on an already connected socket; or, a *sendto* or *sendmsg* request on a connected socket specified a destination other than the connected party.

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

123 ESHUTDOWN Can't send after socket shutdown
A request to send data was disallowed because the socket had already been shut down with a previous *shutdown*(2) call.

124 ETOOMANYREFS Too many references: can't splice
Too many references to the same port.

125 ETIMEDOUT Connection timed out
A *connect* request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

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

127 ELOOP Too many levels of symbolic links
A path name lookup involved more than 8 symbolic links.

128 ENAMETOOLONG File name too long
A component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.

129 ENOTEMPTY Directory not empty
An attempt was made to remove or rename a directory which contained entries other than "." and "..".

130 EHOSTDOWN Host is down
The host system is down. This happens during a connection to the host.

131 EHOSTUNREACH Host unreachable

No route to host. This happens if the host goes down during a data transfer.

132 *unused*

133 *unused*

134 EDQUOT Disc quota exceeded

176 ESTALE Stale NFS filehandle
The file handle given in the argument was invalid. That is, the file referred to by that file handle no longer exists, or access to it has been revoked.

177 EREMOTE Too many levels of remote in path
The remote file or directory being accessed by the client is not local to the server machine, but resides in a file system remotely mounted by the server.

DEFINITIONS

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

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.

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

Effective User ID

An active process has an effective user ID which is used to determine file access permissions (see below). The effective user ID is equal to the process's real user ID, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit set; see *exec*(2).

File Access Permissions

Read, write, and execute/search permissions on a file 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 the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process matches the group of the file and the appropriate access bit of the "group" portion (070) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied.

File Descriptor

A file descriptor is a small integer used to identify a file when issuing i/o requests. The value of a file descriptor is from 0 to a system-defined limit. A process may not have more than the specified limit of simultaneously open files. A file descriptor is returned by system calls such as *open*(2), or *pipe*(2). The file descriptor is used as an argument by calls such as *read*(2), *write*(2), *ioctl*(2), and *close*(2).

File Name

Names consisting of 1 to 255 characters may be used to name an

ordinary file, special file or directory. These characters may be selected from the set of all ASCII character values except 0 and 47 ("NUL" and 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, it is advisable to avoid the use of unprintable characters in file names.

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

00060 Read, Write by group

00006 Read, 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.[c]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 user ID of the process does not match msg_perm.[c]uid and the effective group ID of the process matches msg_perm.[c]gid and the appropriate bit of the "group" portion (060) of msg_perm.mode is set.

The effective user ID of the process does not match msg_perm.[c]uid and the effective group ID of the process does not match msg_perm.[c]gid and the appropriate bit of the "other" portion (06) of msg_perm.mode is set.

Otherwise, the corresponding permissions are denied.

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 includes the following members:

```
struct   ipc_perm msg_perm; /* operation permission struct */
ushort   msg_qnum;                  /* number of msgs on q */
ushort   msg_qbytes;           /* max number of bytes on q */
ushort   msg_lspid;        /* pid of last msgsnd operation */
ushort   msg_lrpid;        /* pid of last msgrcv operation */
time_t   msg_stime;                    /* last msgsnd time */
time_t   msg_rtime;                    /* last msgrcv time */
time_t   msg_ctime;                    /* last change time */
                           /* Times measured in secs since */
                              /* 00:00:00 GMT, Jan 1, 1970 */
```

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 */
```

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.

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.

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. More precisely, a pathname is a null-terminated character string constructed as follows:

<path-name>::=<file-name>|<path-prefix><file-name>|/ <path-prefix>::=<rtprefix>|/<rtprefix>
<rtprefix>::=<dirname>/|<rtprefix><dirname>/

where <file-name> is a string of 1 to 255 characters other than the ASCII slash and null, and <dirname> is a string of 1 to 255 characters (other than the ASCII slash and null) that names a directory. 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. Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

Process Group ID

Each active process is a member of a process group; this group 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).

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.

Real Group ID

Each user allowed on the system is a member of a group. The group is identified by a positive integer called the real group ID. Each active process has a real group ID that is set to the real group ID of the user responsible for its creation.

Real User ID

Each user allowed on the system is identified by a positive integer called a real user ID. Each active process has a real user ID which is set to the real user ID of the user responsible for its creation.

Root Directory and Current Working Directory

Each process has associated with it a root directory and a current working directory for the purpose of resolving pathname searches. The root directory of a process need not be the root directory of the root file system.

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 */
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  cuid;          /* creator user id */
ushort  cgid;         /* creator group id */
ushort  uid;                  /* user id */
ushort  gid;                 /* group id */
ushort  mode;    /* read/alter permission */
```

The value of 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, and 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 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. 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

00060 Read, Alter by group

00006 Read, 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 superuser.

The effective user ID of the process matches sem_perm.[c]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 user ID of the process does not match sem_perm.[c]uid and the effective group ID of the process matches sem_perm.[c]gid and the appropriate bit of the "group" portion (060) of sem_perm.mode is set.

The effective user ID of the process does not match sem_perm.[c]uid and the effective group ID of the process does not match sem_perm.[c]gid and the appropriate bit of the "other" portion (06) 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. The data structure is referred to as *shmid_ds* and includes the following members:

```
struct  ipc_perm  shm_perm; /* operation permission struct */
int     shm_segsz;                    /* size of segment */
ushort  shm_cpid;                        /* creator pid */
ushort  shm_lpid;              /* pid of last operation */
short   shm_nattch;         /* number of current attaches */
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 */
```

Shm_segsz specifies the size of the shared memory segment. It cannot exceed 256 Kbytes.

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 operation, shm_dtime is the time of the last *shmdt* operation, and 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

00060 Read, Write by group

00006 Read, 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 superuser.

The effective user ID of the process matches shm_perm.[c]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 user ID of the process does not match shm_perm.[c]uid and the effective group ID of the process matches shm_perm.[c]gid and the appropriate bit of the "group" portion (060) of shm_perm.mode is set.

The effective user ID of the process does not match shm_perm.[c]uid and the effective group ID of the process does not match shm_perm.[c]gid and the appropriate bit of the "other" portion (06) of shm_perm.mode is set.

Otherwise, the corresponding permissions are denied.

Special Processes

The processes with process ID's of 0, 1, or 2 are special. Process 0 is the scheduler. Process 1 is the initialization process *init*, and is the ancestor of every other process in the system. It is used to control the process structure. Process 2 is the paging daemon.

Super-user

A process is recognized as a super-user process and is granted special privileges if its effective user ID is 0.

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).

SEE ALSO

close(2), ioctl(2), open(2), pipe(2), read(2), write(2), intro(3).

NAME

accept - accept a connection on a socket

SYNOPSIS

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

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

DESCRIPTION

The argument *s* is a socket which 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 as *s* and allocates a new file descriptor, returned as functional value, 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, may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter which 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.

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 writeble part of the user address space.

[EWOULDBLOCK] The socket is marked non-blocking and no connections are present to be accepted.

RETURN VALUE

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

SEE ALSO

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

NAME

access - determine accessibility of a file

SYNOPSIS

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

DESCRIPTION

Path points to a pathname 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

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. A check of read, write, or execute (search) permission has been requested for a null or nonexistent pathname.

[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 filesystem.

[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. 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; all others have permissions checked with respect to the "other" mode bits.

[EFAULT] *Path* points outside the process's allocated address space.

RETURN VALUE

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

SEE ALSO

chmod(2), stat(2).

NAME

acct - enable or disable process accounting

SYNOPSIS

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

DESCRIPTION

Acct is used to enable or disable the system's process accounting routine. If the routine is enabled, an accounting record is written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a *signal*; see *exit*(2) and *signal*(2).

The effective user ID of the calling process must be superuser 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.

Accounting is automatically disabled when the filesystem the accounting file resides on runs out of space; it is enabled when space once again becomes available.

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

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

[EBUSY] An attempt is 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's pathname do not exist.

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

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

[EACCES] Mode permission is denied for the named accounting file.

[EISDIR] The named file is a directory.

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

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

RETURN VALUE

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

BUGS

No accounting is produced for programs running when a crash occurs. In particular, programs that do not terminate are never accounted for.

SEE ALSO

acct(4).

NAME

alarm - set a process's alarm clock

SYNOPSIS

unsigned alarm(sec)
unsigned sec;

DESCRIPTION

Alarm instructs the calling process's alarm clock 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 calling process's alarm clock.

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

RETURN VALUE

Alarm returns the amount of time previously remaining in the calling process's alarm clock

SEE ALSO

pause(2), signal(2).

NAME

bind - bind a name to a socket

SYNOPSIS

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

int 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.

Binding a name in the X/OS domain creates a socket in the file system which must be deleted by the caller when it is no longer needed (using *unlink*(2)). The file created is a side-effect of the current implementation, and will not be created in future versions of the X/OS ipc domain.

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

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.

[EACCESS] 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.

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*.

SEE ALSO

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

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. The newly allocated space is set to zero.

Brk sets the break value to *endds* and changes the allocated space accordingly.

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

Brk and *sbrk* fail without making any change in the allocated space if one or more of the following are true:

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

[ENOMEM] The requested change would result in the break value being greater than or equal to the start address of any attached shared memory segment (see *shmop*(2)).

RETURN VALUE

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

SEE ALSO

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

NAME

chdir - change working directory

SYNOPSIS

```
int chdir(path)
char *path;
```

DESCRIPTION

Path points to the pathname of a directory. *Chdir* causes the named directory to become the current working directory. The starting point for path searches for pathnames that do not begin with /.

Chdir fails and the current working directory remains unchanged if one or more of the following are true:

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

[ENOENT] The named directory does not exist.

[EACCES] Search permission is denied for any component of the pathname.

[EFAULT] *Path* points outside the process's allocated address space.

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

RETURN VALUE

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

SEE ALSO

chroot(2).

NAME

chmod - change mode of file

SYNOPSIS

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

DESCRIPTION

Path points to a pathname naming a file. *Chmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

04000	Set user ID on execution.
02000	Set group ID on execution.
00400	Read by owner.
00200	Write by owner.
00100	Execute (or 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 superuser to change the mode of a file.

If the effective user ID of the process is not superuser or the effective group ID of the process does not match the group ID of the file, mode bits 06000 (set user ID and set group ID on execution) are cleared.

Chmod fails and the file mode remains 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 superuser.

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

[EFAULT] *Path* points outside the process's allocated address space.

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

RETURN VALUE

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

SEE ALSO

chown(2), mknod(2).

NAME

chown - change owner and group of a file

SYNOPSIS

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

DESCRIPTION

Path points to a pathname naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

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

If chown is invoked by other than the superuser, the set-user-ID and set-group-ID bits of the file mode are cleared (bits 04000 and 02000, respectively). See *chmod*(2) for a complete list of access permission bits.

Chown fails and the owner and group of the named file 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 superuser.

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

[EFAULT] *Path* points outside the process's allocated address space.

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

RETURN VALUE

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

SEE ALSO

chmod(2).

NAME

chroot - change root directory

SYNOPSIS

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

DESCRIPTION

Path points to a pathname naming a directory. *Chroot* causes the named directory to become the root directory, which is the starting point for path searches for pathnames that begin with /.

To change the root directory, the effective user ID of the process must be super-user.

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.

Chroot fails and the root directory remains unchanged if one or more of the following are true:

[ENOTDIR] Any component of the pathname is not a directory.

[ENOENT] The named directory does not exist.

[EPERM] The effective user ID is not superuser.

[EFAULT] *Path* points outside the process's allocated address space.

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

[EACCES] Search permission is denied for any component of the path name.

RETURN VALUE

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

SEE ALSO

chdir(2).

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.

Close will fail if:

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

RETURN VALUE

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

SEE ALSO

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

NAME

connect - initiate a connection on a socket

SYNOPSIS

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

int 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 permanently specifies the peer to which datagrams are to be sent; if it 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.

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] The connection could not be established before the timeout period expired.

[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.

[EWOULDBLOCK] The socket is non-blocking and the connection cannot be completed immediately. It is possible to *select*(2) the socket while it is connecting by selecting it for writing.

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*.

SEE ALSO

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

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 pathname 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 process's effective user ID, the file's group ID is set to the process's effective group ID, 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).

Upon successful completion, a non-negative integer, namely the 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). A new file may be created with a mode that forbids writing.

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.

[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] Too many file descriptors are currently open.

[EFAULT] *Path* points outside the process's allocated address space.

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

RETURN VALUE

Upon successful completion, a non-negative integer (i.e., the file descriptor) is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

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

NAME

dup, dup2 - duplicate an open file descriptor

SYNOPSIS

```
int dup(fildes)
int fildes;

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

DESCRIPTION

This function can be used, for example, to effectively open the same file twice, simultaneously - once for reading and once for writing. It is more efficient (in terms of cpu time used) to call *open* and then *dup* than to call *open* twice.

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.

In the second form of the call, the value of *newdes* desired is specified. If this descriptor is already in use, the descriptor is first deallocated as if a *close*(2) call had been done first.

Dup and *dup2* will fail if one or more of the following are true:

[EBADF] *Fildes* or *newdes* is not a valid active descriptor

[EMFILE] Too many descriptors are active.

RETURN VALUE

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

SEE ALSO

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

NAME

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. This file consists of a header (see *a.out*(4)), a text segment, and a data segment. The data segment contains an initialised portion and an uninitialised portion (bss).

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

An interpreter file begins with a line of the form:

#! interpreter

When an interpreter file is *exec*'ed, the system *exec*'s the specified *interpreter*, giving it the name of the originally *exec*'ed

file as an argument, shifting over the rest of the original arguments.

When a C program (which resides in the file specified by *path* or *file*) is *exec*uted, it is called as follows:

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

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string naming the file to be *exec*'ed.

Path points to a pathname 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)).

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 calling process's environment in the global cell:

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

This pointer is used to pass the calling process's environment 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 are set to terminate the new process. Signals set to be ignored by the calling process are set to be ignored by the new process. Signals set to be caught by the calling process are set to terminate the new process; see *signal*(2).

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 are not 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))

Exec fails and returns to the calling process if one or more of the following are true:

[ENOENT] One or more components of the new process file's pathname do not exist.

[ENOTDIR] A component of the new process file's path 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 10,240 bytes.

[EFAULT] The size of the new process file is less than that indicated by the size values in its header.

[EFAULT] *Path*, *argv*, or *envp* points to an illegal address.

RETURN VALUE

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

SEE ALSO

exit(2), fork(2), environ(5).

NAME

exit, _exit - terminate process

SYNOPSIS

```
void exit(status)
int status;

void _exit(status)
int status;
```

DESCRIPTION

Exit and *_exit* terminate the calling process with the following consequences:

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

- If the parent process of the calling process is executing a *wait* or is interested in the SIGCLD signal, it is notified of the calling process's termination and the low-order 8 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 in user space 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*(2).

- The parent process ID of all the calling process's 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; see *shmop*(2).

- For each semaphore for which the calling process has set a semaphore adjustment (*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, then the SIGHUP signal is sent to each process that has a process group ID equal to that of the calling process.

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

SEE ALSO

acct(2), plock(2), semop(2), shmop(2), signal(2), times(2), wait(2).

WARNING

See WARNING in signal(2).

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 permitted values for the *cmd* command argument 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_TRUNC Truncate the file associated with the file descriptor *fildes* to the length specified in *arg* (expressed as number of bytes).

F_SYNC Flush out all the buffers pertaining to the file specified via *fildes* which are open for writing. The caller is guaranteed to get control back only after data is actually on disk. (This command is only useful for files that have not been opened with the O_SYNC flag set; if the *writes* were synchronous, the buffers would have been flushed by the *write* calls themselves).

F_GETLK Get information about the first part of the file which is currently locked. The starting point for the search for locked areas is defined in the *flock* structure (pointed to by *arg*). The information is returned in the structure *flockd*, which is also pointed to by *arg*. Thus the information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found, 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 command *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 prevented 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*), and process id (*l_pid*) of the segment of the file to be affected. The process id field is only used with the *F_GETLK* command; it returns the id of the process which locked this segment of the file. 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_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 at each 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 affect. 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.

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

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

[EMFILE] *Cmd* is F_DUPFD and *arg* is negative or greater than the maximum allowable number of open file descriptors.

[EINVAL] *Cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.

[EINVAL] *Cmd* is F_GETLK, F_SETLK, or SETLKW and *fildes* is a file descriptor for a file or directory, which is on another machine or is not in System V format.

[EACCESS] *Cmd* is F_SETLK the type of lock (*l_type*) is a read (F_RDLCK) or write (F_WRLCK) lock, and the segment of a file to be locked is already write locked by another process; or, the type is a write lock, and the segment of a file to be locked is already read or write locked by another process.

[EMFILE] *Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more file locking headers available (too many files have segments locked).

[ENDSPC] *Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock and there are no more file locking headers available (too many files have segments locked) or there are no more record locks available (too many file segments locked).

[EDEADLK] *Cmd* is F_SETLKW; potential deadlock situation detected.

RETURN VALUE

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_TRUNC Value other than -1.

F_SYNC Value other than -1.

F_GETLK Value other that -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.

SEE ALSO

close(2), exec(2), open(2), fcntl(5).

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, 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) 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))

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.

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* (see *times*(2)) are set to 0.

Fork fails and no child process is 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.

RETURN VALUE

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the

parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

SEE ALSO

exec(2), times(2), wait(2).

NAME

getdirentries - gets directory entries in a filesystem-independent format

SYNOPSIS

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

int getdirentries(fd, buf, nbytes, basep)
int fd;
char *buf;
int nbytes;
long *basep
```

DESCRIPTION

Getdirentries attempts to put directory entries from the directory referenced by the file descriptor *fd* into the buffer pointed to by *buf*, in a filesystem-independent format. Up to *nbytes* of data will be transferred. *Nbytes* must be greater than or equal to the block size associated with the file, see *stat(2)*. Sizes less than this may cause errors on certain filesystems.

The data in the buffer is a series of *direct* structures each containing the following entries:

```
unsigned long  d_fileno;
unsigned short d_reclen;
unsigned short d_namlen;
char           d_name[MAXNAMELEN + 1];      /* see below */
```

The *d_fileno* entry is a number which is unique for each distinct file in the filesystem. Files that are linked by hard links (see *link(2)*) have the same *d_fileno*. The *d_reclen* entry is the length, in bytes, of the directory record. The *d_name* entry contains a null terminated file name. The *d_namlen* entry specifies the length of the file name. Thus the actual size of *d_name* may vary from 2 to MAXNAMELEN + 1.

The structures are not necessarily tightly packed. The *d_reclen* entry may be used as an offset from the beginning of a *direct* structure to the next structure, if any.

Upon return, the actual number of bytes transferred is returned. The current position pointer associated with *fd* is set to point to the next block of entries. The pointer is not necessarily incremented by the number of bytes returned by *getdirentries*. If the value returned is zero, the end of the directory has been reached. The current position pointer may be set and retrieved by *lseek(2)*. *Getdirentries* writes the position of the block read into the location pointed to by *basep*. It is not safe to set the current position pointer to any value other than a value previously returned by *lseek(2)* or a value previously returned in the location pointed to by *basep* or zero.

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

[EBADF] *fd* is not a valid file descriptor open for reading.

[EFAULT] Either *buf* or *basep* points outside the allocated address space.

[EINTR] A read from a slow device was interrupted by the delivery of a signal, before any data arrived.

[EIO] An I/O error occurred while reading from or writing to the file system.

RETURN VALUE

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

SEE ALSO

open(2), lseek(2)

NAME

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

SYNOPSIS

```
int gethostid()

int sethostid(hostid)
int hostid;
```

DESCRIPTION

Sethostid establishes a 32-bit identifier for the current processor which is intended to be unique among all X/OS 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)

NAME

gethostname, sethostname - get/set name of current host

SYNOPSIS

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

int 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.

Host names are limited to 255 characters.

The following errors may be returned by these calls:

[EFAULT] The *name* or *namelen* parameter gave an invalid address.

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

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*.

SEE ALSO

gethostid(2)

NAME

getitimer, setitimer - get/set value of interval timer

SYNOPSIS

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

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

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

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

DESCRIPTION

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

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

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

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

is non-zero).

Time values smaller than the resolution of the system clock (16.667 milliseconds) are rounded up to this resolution.

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.

Three macros for manipulating time values are defined in <*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).

Getitimer and *setitimer* will fail if:

[EFAULT] The *value* structure specified a bad address.

[EINVAL] A *value* structure specified a time too large to be handled.

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*.

SEE ALSO

sigvec(2), gettimeofday(2)

NAME

getpagesize - get system page size

SYNOPSIS

int getpagesize()

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 that of a system page; it is not necessarily the same as the underlying hardware page size.

SEE ALSO

sbrk(2)

NAME

getpeername - get name of connected peer

SYNOPSIS

```
int 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 initialised to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

Names bound to sockets in the X/OS domain are inaccessible; *getpeername* returns a zero length name.

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.

RETURN VALUE

A 0 is returned if the call succeeds, -1 if it fails. In the latter case, *errno* is set to show the error.

SEE ALSO

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

getpriority, setpriority - get/set program scheduling priority

SYNOPSIS

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

#define PRIO_PROCESS     0    /* process */
#define PRIO_PGRP        1    /* process group */
#define PRIO_USER        2    /* user id */

int getpriority(which, who)
int which, who;

int setpriority(which, who, prio)
int which, who, prio;
```

DESCRIPTION

The scheduling priority of the process, process group, or user, as indicated by *which* and *who* is obtained with the *getpriority* call and set with the *setpriority* call. *Which* is one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, and *who* is interpreted relative to *which* (a process identifier for PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user ID for PRIO_USER). *Prio* is a value in the range -20 to 20. The default priority is 0; lower priorities cause more favourable scheduling.

The *getpriority* call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The *setpriority* call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

Getpriority and *setpriority* will fail if:

[ESRCH] No process(es) were located using the *which* and *who* values specified.

[EINVAL] *Which* was not one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.

In addition to the errors indicated above, *setpriority* can return the following:

[EACCES] A process was located, but neither its effective nor real user ID matched the effective user ID of the caller.

[EACCES] A non super-user attempted to change a process priority to a negative value.

RETURN VALUE

Since *getpriority* can legitimately return the value -1, it is necessary to clear the external variable *errno* prior to the call, then check it afterward to determine if a -1 is an error or a legitimate value.

The *setpriority* call returns 0 if there is no error, or -1 if there is.

SEE ALSO

nice(1), fork(2), nice(2).

NAME

getrlimit, setrlimit - control maximum system resource consumption

SYNOPSIS

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

int getrlimit(resource, rlp)
int resource;
struct rlimit *rlp;

int setrlimit(resource, rlp)
int resource;
struct rlimit *rlp;
```

DESCRIPTION

Limits on the consumption of system resources by the current process and each process it creates may be obtained with the *getrlimit* call, and set with the *setrlimit* call.

The *resource* parameter is one of the following:

RLIMIT_CPU — the maximum amount of cpu time (in milliseconds) to be used by each process.

RLIMIT_FSIZE — the largest size, in bytes, of any single file which may be created.

RLIMIT_DATA — the maximum size, in bytes, of the data segment for a process; this defines how far a program may extend its break with the *sbrk*(2) system call.

RLIMIT_STACK — the maximum size, in bytes, of the stack segment for a process; this defines how far a program's stack segment may be extended, either automatically by the system, or explicitly by a user with the *sbrk*(2) system call.

RLIMIT_CORE the largest size, in bytes, of a *core* file which may be created.

RLIMIT_RSS the maximum size, in bytes, a process's resident set size may grow to. This imposes a limit on the amount of physical memory to be given to a process; if memory is tight, the system will prefer to take memory from processes which are exceeding their declared resident set size. A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded a process may receive a signal (for example, if the cpu time is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The *rlimit* structure is used to specify the hard and soft limits on a resource,

```
struct rlimit {
    int  rlim_cur; /* current (soft) limit */
    int  rlim_max; /* hard limit */
};
```

Only the super-user may raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*. An "infinite" value for a limit is defined as RLIMIT_INFINITY (0x7fffffff). Because this information is stored in the per-process information, this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to *csh*(1). The system refuses to extend the data or stack space when the limits would be exceeded in the normal way: a *break* call fails if the data space limit is reached, or the process is killed when the stack limit is reached (since the stack can not be extended, there is no way to send a signal!). A file i/o operation which would create a file which is too large will cause a signal SIGXFSZ to be generated, this normally terminates the process, but may be caught. When the soft cpu time

limit is exceeded, a signal SIGXCPU is sent to the offending process.

Getrlimit and *setrlimit* fail and the current limits are left unchanged if:

[EFAULT] The address specified for *rlp* is invalid.

[EPERM] The limit specified to *setrlimit* would have raised the maximum limit value, and the caller is not the super-user.

RETURN VALUE

A 0 return value indicates that the call succeeded, changing or returning the resource limit. A return value of -1 indicates that an error occurred, and an error code is stored in the global location *errno*.

SEE ALSO

ulimit(2).

NAME

getrusage - get information about resource utilisation

SYNOPSIS

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

#define RUSAGE_SELF       0      /* calling process */
#define RUSAGE_CHILDREN  -1      /* terminated child processes */

int getrusage(who, rusage)
int who;
struct rusage *rusage;
```

DESCRIPTION

Getrusage returns information describing the resources utilised by the current process, or all its terminated child processes. The *who* parameter is one of RUSAGE_SELF and RUSAGE_CHILDREN. If *rusage* is non-zero, the buffer it points to will be filled in with the following structure:

```
struct  rusage {
    struct timeval ru_utime;              /* user time used */
    struct timeval ru_stime;            /* system time used */
    int     ru_maxrss;
    int     ru_ixrss;        /* integral shared memory size */
    int     ru_idrss;        /* integral unshared data size */
    int     ru_isrss;       /* integral unshared stack size */
    int     ru_minflt;                    /* page reclaims */
    int     ru_majflt;                      /* page faults */
    int     ru_nswap;                             /* swaps */
    int     ru_inblock;          /* block input operations */
    int     ru_oublock;         /* block output operations */
    int     ru_msgsnd;                    /* messages sent */
    int     ru_msgrcv;                /* messages received */
    int     ru_nsignals;               /* signals received */
    int     ru_nvcsw;        /* voluntary context switches */
```

```
        int     ru_nivcsw;      /* involuntary context switches */
    };
```

The fields are interpreted as follows:

ru_utime — the total amount of time spent executing in user mode.

ru_stime — the total amount of time spent in the system executing on behalf of the process(es).

ru_maxrss — the maximum resident set size utilised (in kilobytes).

ru_ixrss — an "integral" value indicating the amount of memory used which was also shared among other processes. This value is expressed in units of kilobytes * seconds-of-execution and is calculated by summing the number of shared memory pages in use each time the internal system clock ticks and then averaging over 1-second intervals.

ru_idrss — an integral value of the amount of unshared memory residing in the data segment of a process (expressed in units of kilobytes * seconds-of-execution).

ru_isrss — an integral value of the amount of unshared memory residing in the stack segment of a process (expressed in units of kilobytes * seconds-of-execution).

ru_minflt — the number of page faults serviced without any i/o activity; here i/o activity is avoided by "reclaiming" a page frame from the list of pages awaiting reallocation.

ru_majflt — the number of page faults serviced which required i/o activity.

ru_nswap — the number of times a process was "swapped" out of main memory.

ru_inblock — the number of times the file system had to perform input.

ru_outblock — the number of times the file system had to perform output.

ru_msgsnd — the number of ipc messages sent.

ru_msgrcv — the number of ipc messages received.

ru_nsignals — the number of signals delivered.

ru_nvcsw — the number of times a context switch occurred because a process voluntarily gave up the processor before its time-slice was completed (usually to wait for the availability of a resource).

ru_nivcsw — the number of times a context switch occurred as a result of a higher priority process becoming runnable or because the current process exceeded its time-slice.

NOTES

The numbers *ru_inblock* and *ru_outblock* account only for real i/o; data supplied by the cacheing mechanism is charged only to the first process to read or write the data.

There is no way for a parent process to obtain information about a child process which has not yet terminated (though the child process can obtain information about itself).

SEE ALSO

gettimeofday(2), wait(2).

NAME

getsockname - get socket name

SYNOPSIS

```
int 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 initialised to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

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

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 which is not in a valid part of the process address space.

RETURN VALUES

A 0 is returned if the call succeeds, -1 if it fails. In this case, *errno* is set to show the error.

SEE ALSO

bind(2), socket(2).

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

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

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

int 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 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 *<sys/socket.h>* contains definitions for "socket" level options; see *socket*(2). Options at other protocol levels vary in format and name - consult the appropriate entries in the protocols file (see *protocols*(4)).

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.

[EFAULT] The options are not in a valid part of the process address space.

RETURN VALUE

A 0 is returned if the call succeeds, -1 if it fails. In the latter case *errno* is set to show the error.

SEE ALSO

socket(2), getprotoent(3N).

NAME

gettimeofday, settimeofday - get/set date and time

SYNOPSIS

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

int gettimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;

int settimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;
```

DESCRIPTION

Gettimeofday returns the system's notion of the current Greenwich time and the current time zone. Time returned is expressed relative in seconds and microseconds since midnight January 1, 1970.

The structures pointed to by *tp* and *tzp* are defined in <*sys/time.h*> as:

```
struct timeval {
    u_long  tv_sec;         /* seconds since Jan.  1, 1970 */
    long    tv_usec;                    /* and microseconds */
};

struct timezone {
    int     tz_minuteswest;                 /* of Greenwich */
    int     tz_dsttime; /* type of dst correction to apply */
};
```

The *timezone* structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving Time (or its equivalent) applies locally during the appropriate part of the year.

Only the super-user may set the time of day.

Gettimeofday and *settimeofday* will fail if:

[EFAULT] An argument address references invalid memory.

[EPERM] A user other than the super-user attempts to set the time (*settimeofday*).

RETURN VALUE

A 0 return value indicates that the call succeeded. A -1 return value indicates that an error occurred; in this case an error code is set in the global variable *errno*.

SEE ALSO

date(1), time(2), ctime(3).

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

ioctl - control device

SYNOPSIS

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

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The descriptions of various devices in Section 7 of the "System Administration Utilities Reference Manual" discuss how *ioctl* applies to them.

The *request* argument gives the following information:

- whether *arg* is an "in" parameter or "out" parameter
- the size of the argument *arg* in bytes.

Macros and defines used in specifying an ioctl *request* are located in the file <*sys/ioctl.h*>.

Ioctl fails 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 character special device.

[EINVAL] *Request* or *arg* is not valid. See Section 7.

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

RETURN VALUE

If an error occurs, a value of -1 is returned and errno is set to indicate the error.

SEE ALSO

termio(7).

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 the process or group of processes specified by *pid*; the signal may or may not kill the process(es) specified.

The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), *sigvec*(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 superuser.

The processes with a process ID of 0, 1 and 2 are special processes (see *intro*(2)) and are referred to below as proc0, proc1 and proc2, respectively.

The table below shows how the combination of the *pid* argument and the effective user id of the caller determine which process(es) are to receive the signal.

pid arg.	Effective user id of caller	Recipient of signal	Excluded recipients
> 0	superuser	process with id *pid*	none
> 0	not superuser	process with id *pid*	those whose real or effective user id does not equal that of caller
0	superuser	processes with group id equal to the process group id of caller	proc0, proc1, proc2
0	not superuser	processes with group id equal to the process group id of group id of caller	proc0, proc1, proc2; and those whose real or effective user id does not equal that of caller
-1	superuser	all processes except 0, 1, 2	proc0, proc1, proc2
-1	not superuser	processes with real user id equal to effective user id of sender	proc0, proc1, proc2
-n, where *n* > 1	superuser	processes with process group id equal to *n*	none
-n, where *n* > 1	not superuser	processes with process group id equal to *n*	those whose real or effective user id does not equal that of caller

Kill fails and no signal is 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.

[ESRCH] No process can be found corresponding to that specified by *pid*.

[EPERM] The user ID of the sending process is not superuser, and its real or effective user ID does not match the real or effective user ID of the receiving process.

RETURN VALUE

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

SEE ALSO

kill(1), getpid(2), setpgrp(2), signal(2), sigvec(2).

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.

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).

[EACCES] The requested link requires writing in a directory 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.

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

RETURN VALUE

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

SEE ALSO

symlink(2), unlink(2).

NAME

listen - listen for connections on a socket

SYNOPSIS

```
int listen(s, backlog)
int s, backlog;
```

DESCRIPTION

To accept connections, a socket is first created with *socket*(2), a backlog for incoming connections is 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_PKTSTREAM.

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 will receive an error with an indication of ECONNREFUSED.

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

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*.

RETURN VALUE

A 0 return value indicates success; -1 indicates an error. In the latter case, *errno* is set to show the error.

SEE ALSO

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

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*(2), *open*(2), *dup*(2), or *fcntl*(2) 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*.

Seeking far beyond the end of a file, then writing, creates a gap or "hole", which occupies no physical space and reads as zeros.

Upon successful completion, the resulting pointer location, measured in bytes from the beginning of the file, is returned.

Lseek fails and the file pointer remains 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] *Whence* is not 0, 1, or 2 (SIGSYS signal is sent).

[EINVAL] 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.

RETURN VALUE

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

SEE ALSO

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

NAME

mkdir - make a directory file

SYNOPSIS

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

DESCRIPTION

Mkdir creates a new directory file with name *path*. The mode of the new file is initialised from *mode*. (The protection part of the mode 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 that of the parent directory in which it is created.

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).

Mkdir will fail and no directory will be created if:

[EPERM] The process's effective user ID is not super-user.

[EPERM] The *path* argument contains a byte with the high-order bit set.

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

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

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

[EEXIST] The named file exists.

[EFAULT] *Path* points outside the process's allocated address space.

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

[EIO] An I/O error occured while writing to the file system.

RETURN VALUE

A 0 return value indicates success. A -1 return value indicates an error, and an error code is stored in *errno*.

SEE ALSO

chmod(2), stat(2), umask(2).

NAME

mknod - make 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 initialised from *mode*, where the value of *mode* is interpreted as follows:

0170000	file type; one of the following:
	0010000 fifo special
	0020000 character special
	0060000 block special
	0100000 or 0000000 ordinary file
0004000	set-user-ID on execution
0002000	set-group-ID on 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. *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.

[EISDIR] The file type part of *mode* is "directory". Mkdir(2) should be used for making directories.

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

RETURN VALUE

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

SEE ALSO

mkdir(1), chmod(2), exec(2), mkdir(2), umask(2), fs(4).

NAME

mount, umount - mount or remove file system

SYNOPSIS

```
int mount(special, name, rwflag)
char *special, *name;
int rwflag;

int umount(special)
char *special;
```

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block-structured special file *special*; from now on, references to file *name* will refer to the root file on the newly mounted file system. *Special* and *name* are pointers to null-terminated strings containing the appropriate pathnames.

Name must exist already. *Name* must be a directory. Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done. Physically write-protected and magnetic tape file systems must be mounted read-only, or errors will occur when access times are updated, whether or not any explicit write is attempted.

Umount announces to the system that the *special* file is no longer to contain a removable file system. The associated file reverts to its ordinary interpretation.

Both *mount* and *umount* only work on the local machine and are retained for compatibility with previous versions of the system. The new system calls, only available with the nfs software kit, *vfsmount*(2) and *vfsunmount*(2), are capable of handling both local and remote file systems.

Mount will fail when one of the following occurs:

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

[NODEV] *Special* does not exist.

[ENOTBLK] *Special* is not a block device.

[ENXIO] The major device number of *special* is out of range (this indicates that no device driver exists for the associated hardware).

[EPERM] The pathname contains a character with the high-order bit set.

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

[EROFS] *Name* resides on a read-only file system.

[EBUSY] *Name* is not a directory, or another process currently holds a reference to it.

[EBUSY] No space remains in the mount table.

[EBUSY] The super block for the file system had a bad magic number or an out of range block size.

[EBUSY] Not enough memory was available to read the cylinder group information for the file system.

[EBUSY] An i/o error occurred while reading the super block or cylinder group information.

Umount will fail if one of the following occurs:

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

[NODEV] *Special* does not exist.

[ENOTBLK] *Special* is not a block device.

[ENXIO] The major device number of *special* is out of range (this indicates that no device driver exists for the associated hardware).

[EINVAL] The requested device is not in the mount table.

[EBUSY] A process is holding a reference to a file located on the file system.

RETURN VALUE

Mount returns 0 if the action occurred, -1 if *special* is inaccessible or not an appropriate file, if *name* does not exist, if *special* is already mounted, if *name* is in use, or if there are already too many file systems mounted.

Umount returns 0 if the action occurred; -1 if the special file is inaccessible or does not have a mounted filesystem, or if there are active files in the mounted filesystem.

SEE ALSO

fs(4).
vfsmount(2) and vfsunmount(2) in the *NFS User Manual.*

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 commands 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).

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 superuser or to the value of msg_perm.uid in the data structure associated with *msqid*. Only superuser 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 superuser or to the value of msg_perm.uid in the data structure associated with msqid.

Msgctl fails 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 and the effective user ID of the calling process is not equal to that of superuser and is not equal to the value of msg_perm.uid in the data structure associated with *msqid*.

[EPERM] *Cmd* is equal to IPC_SET, an attempt is being made to increase the value of msg_qbytes, and the effective user ID of the calling process is not equal to that of superuser.

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

RETURN VALUE

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

SEE ALSO

msgget(2), msgop(2).

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 is 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 initialised 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.

Msgget fails 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".

RETURN VALUE

Upon successful completion, a non-negative integer (i.e., a message queue identifier) is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

msgctl(2), msgop(2).

NAME

msgsnd, msgrcv - 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*.

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 is not sent and the calling process returns immediately.

If (*msgflg* & IPC_NOWAIT) is "false", the calling process suspends execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

Msqid is removed from the system (see *msgctl*(2)). When this occurs, *errno* is set equal to EIDRM and a value of -1 is returned.

The calling process receives a signal 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)).

Msgsnd fails and no message is 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*. 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 returns immediately with a return value of -1 and *errno* set to ENOMSG.

If (*msgflg* & IPC_NOWAIT) is "false", the calling process suspends execution until one of the following occurs:

A message of the desired type is placed on the queue.

Msqid is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of -1 is returned.

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(2)).

Msgrcv fails and no message is 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.

RETURN VALUES

If *msgsnd* or *msgrcv* returns prematurely due to the receipt of a signal (i.e. interrupt), 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.

SEE ALSO

msgctl(2), msgget(2).

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 positive number for which a more positive value results in lower CPU priority. A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit.

Nice fails and does not change the nice value if:

[EPERM] *Incr* is negative and the effective user ID of the calling process is not superuser.

RETURN VALUE

Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

nice(1), exec(2).

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 pathname naming a file. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by or-ing flags from the following list. The NOTES section describes the interactions between these flags, and permitted combinations.

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).

O_APPEND Append on each write. If O_APPEND is 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 the 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

- the low-order 12 bits of the file mode are set to the value of *z*. *z* is defined thus:

  z equals *x* logically and-ed with *y*.

  x is the logical "not" of the calling process's file mode creation mask (i.e. every set bit in the mask is cleared in *x*, every clear bit in the mask is set in *x*).

  y is the *mode* argument.

  For further details, see *creat*(2) and *umask*(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.

O_ORDERED If this flag is set, the file will be updated in order, asynchronously. This guarantees that in a series of asynchronous *write* calls, the data will reach the disk in the same sequence order as that of the *write* calls. (For example: If the 4th and 5th calls both update the same byte in the file, and no other calls access this byte, then at the end of execution the byte is guaranteed to contain the output from the 5th call.) This is not true in the general case, since a disk driver may reorder the *write* requests it receives, in order to minimize seek operations.

This mode of operation is only meaningful for files opened for writing.

If the *open* succeeds, 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:

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

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

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

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

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

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

[EMFILE] Too many file descriptors are currently open.

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

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

[EFAULT] *Path* points outside the allocated address space of the process.

[EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

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

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

[ENFILE] The system file table is full.

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

NOTES

When combining the use of several flags, note the following:

- Only one of the three flags O_RDONLY, O_WRONLY, and O_RDWR may be used.

- When opening a FIFO with O_RDONLY or O_WRONLY set, and O_NDELAY 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.

- When opening a FIFO with O_RDONLY or O_WRONLY set, and O_NDELAY 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.

- O_SYNC and O_ORDERED are mutually exclusive.

When opening a file associated with a communication line:

- if O_NDELAY is set:

  The *open* will return without waiting for the carrier.

- if O_NDELAY is clear:

  The *open* will block until the carrier is present.

RETURN VALUE

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

SEE ALSO

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

IAME

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* does 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.

RETURN VALUE

A value of -1 is returned from *pause* and *errno* is set to EINTR.

SEE ALSO

alarm(2), kill(2), signal(2), sigvec(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.

Writes of up to 5,120 bytes of data are buffered by the pipe before the writing process is blocked. A read on fildes[0] accesses the data written to fildes[1] on a first-in-first-out (FIFO) basis.

Pipe fails and no pipe is created if one or more of the following are true:

[EMFILE] Too many file descriptors are currently open.

[ENFILE] The system file table is full.

RETURN VALUE

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

SEE ALSO

sh(1), read(2), write(2).

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 and paging. *Plock* also allows these segments to be unlocked. The effective user ID of the calling process must be superuser to use this call.

Op specifies the following:

PROCLOCK	lock text & 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

Plock fails and does 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 superuser.

[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.

RETURN VALUE

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.

SEE ALSO

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

IAME

profil - execution time profile

SYNOPSIS

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

DESCRIPTION

Buff points to an area of memory whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (100 ms); *offset* is subtracted from it and the result is multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: either 0177777 or 0200000, octal (i.e., 0xffff and 0x10000 hexadecimal, respectively), give a 1-1 mapping of pc's to words in *buff*; either 077777 or 0100000, octal (i.e., 0x7fff and 0x8000 hexadecimal, respectively), map each pair of instruction words together. 02 (octal) maps all instructions onto the beginning of *buff* (producing a non-interrupting memory 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 both child and parent after a *fork*.

Profiling is turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

Not defined.

SEE ALSO

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

NAME

ptrace - process trace

SYNOPSIS

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

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging; see *sdb*(1). The child process behaves normally until it encounters a signal (see *signal*(2) for a list of signals), at which time it enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace*. The parent also can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

All the requests except 0 can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by the *func* argument of *signal*(2). The *pid*, *addr*, and *data* arguments are ignored and a return value is not defined for this request. Peculiar results ensue if the parent does not expect to trace the child.

1 The word at location *addr* in the address space of the child is returned to the parent process. The *data* argument is ignored.

The request fails if *addr* is not the start address of a word, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

2 This is the same as request 1 (on the LSX computer line).

3 With this request, the word at location *addr* in the child's USER area in the system's address space (see *<sys/user.h>*) is returned to the parent process. Addresses in this area range from 0 to 4096. The *data* argument is ignored. This request fails if *addr* is not the start address of a word or is outside the USER area, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

4 The value given by the *data* argument is written into the address space of the child at location *addr*. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests fail if *addr* is a location in a pure procedure space and another process is executing in that space, or if *addr* is not the start address of a word. Upon failure a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

5 Request 5 is identical to request 4 on the LSX computers.

6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The entries that can be written are:

- the data registers (d0-d7)
- the address registers (a0-a6)
- the user stack pointer (a7)
- the program counter
- bits 0-4 of the Processor Status Register (Condition Code Register).

7 This request causes the child to resume execution.

If the *data* argument is 0, all pending signals, including the one that caused the child to stop, are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal; any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the parent.

This request fails if *data* is not 0 or a valid signal number, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

8 This request causes the child to terminate with the same consequences as *exit*(2).

9 This request sets the trace bit in the Processor Status Register of the child (i.e., bit 15 on the M68020) and then executes the same steps as listed above for request 7. The trace bit causes an interrupt upon completion of one machine instruction. This effectively allows single stepping of the child. Note: the trace bit is turned off after an interrupt on the M68020.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec*(2) calls. If a traced process calls *exec*, it stops before executing the first instruction of the new image showing signal SIGTRAP. *Ptrace* in general fails if one or more of the following are true:

[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

sdb(1), exec(2), signal(2), wait(2).

NAME

read, readv - read from file

SYNOPSIS

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

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

int readv(fildes, iov, iovcnt)
int fildes;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

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

Readv performs the same action, but scatters the input data into the *iovcnt* buffers specified by the members of the *iovec* array: iov[0], iov[1], ..., iov[iovcnt - 1].

For *readv*, the *iovec* structure is defined as

```
struct iovec {
   caddr_t  iov_base;
   int      iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. *Readv* will always fill an

area completely before proceeding to the next.

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* and *readv* return 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.

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

If O_NDELAY is set, the read returns a 0.

If O_NDELAY is clear, the read blocks until data is written to the file or the file is no longer open for writing.

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

If O_NDELAY is set, the read returns a 0.

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

Read and *readv* 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.

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

In addition, *readv* may return one of the following errors:

[EINVAL] *Iovcnt* was less than or equal to 0, or greater than 16.

[EINVAL] One of the *iov_len* values in the *iov* array was negative.

[EINVAL] The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.

RETURN VALUE

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.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), open(2), pipe(2), termio(7).

NAME

readlink - read value of a symbolic link

SYNOPSIS

```
int readlink(path, buf, bufsiz)
char *path, *buf;
int bufsiz;
```

DESCRIPTION

Readlink places the contents of the symbolic link *path* in the buffer *buf* which has size *bufsiz*. The contents of the link are not null-terminated when returned.

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

[ENOENT] The pathname was too long.

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

[ENOENT] The named file does not exist.

[ENXIO] The named file is not a symbolic link.

[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 the super-user.

[EINVAL] The named file is not a symbolic link.

[EFAULT] *Buf* extends outside the process's allocated address space.

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

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*.

SEE ALSO

stat(2), lstat(2), symlink(2).

NAME

reboot - reboot system or halt processor

SYNOPSIS

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

int reboot(howto)
int howto;
```

DESCRIPTION

Reboot reboots the system, and is invoked automatically in the event of unrecoverable system failures. *Howto* is a mask of options passed to the bootstrap program. The system call interface permits only RB_HALT or RB_AUTOBOOT to be passed to the reboot program; the other flags are used in scripts stored on the console storage media, or used in manual bootstrap procedures. When none of these options (e.g. RB_AUTOBOOT) is given, the system is rebooted from the file specified within the configuration file "/conf" in the designated root file system (see *disconf*(1M) and *Installation Guide*). If a configuration file does not exist, or it does not specify the location of the kernel to be bootstrapped, the default choice is "/unix" in the designated root file system. An automatic consistency check of the disks is then normally performed.

The bits of *howto* are:

RB_HALT
: the processor is simply halted; no reboot takes place. RB_HALT should be used with caution.

RB_ASKNAME
: Interpreted by the bootstrap program itself, causing it to inquire as to what file should be booted. Normally, the system is booted from the file "unix" in the designated root file system, without asking.

RB_SINGLE

Normally, the reboot procedure involves an automatic disk consistency check and then multi-user operations. RB_SINGLE prevents the consistency check, rather simply booting the system with a single-user shell on the console. RB_SINGLE is interpreted by the *init*(1M) program in the newly booted system. This switch is not available from the system call interface.

Only the super-user may *reboot* a machine.

Reboot can fail with the following error code:

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

RETURN VALUES

If successful, this call never returns. Otherwise, a -1 is returned and an error is returned in the global variable *errno*.

SEE ALSO

crash(1M), disconf(1M), halt(1M), init(1M), reboot(1M).
Installation Guide.

NAME

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

SYNOPSIS

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

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

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

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

DESCRIPTION

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

The *recv* call may be 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, initialised 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 as the functional value. 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 non-blocking (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 send call is formed by or-ing one or more of the values:

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

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

```
struct msghdr {
    caddr_t  msg_name;                  /* optional address */
    int      msg_namelen;                /* size of address */
    struct   iov      *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). Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*.

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.

RETURN VALUE

These calls return the number of bytes received, or -1 if an error occurred. In the latter case *errno* is set to show the error.

SEE ALSO

read(2), send(2), socket(2).

NAME

rename - change the name of a file

SYNOPSIS

```
int 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. 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.

Rename will fail and neither of the argument files will be affected if any 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 *from* does not exist.

[EPERM] The file named by *from* is a directory and the effective user ID is not super-user.

[EXDEV] The link named by *to* and the file named by *from* are on different logical devices (file systems).

[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 process's allocated address space.

[EINVAL] *From* is a parent directory of *to*.

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.

SEE ALSO

open(2).

NAME

rmdir - remove a directory file

SYNOPSIS

```
int rmdir(path)
char *path;
```

DESCRIPTION

Rmdir removes a directory file whose name is given by *path*. The directory must not have any entries other than "." and "..".

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

[ENOTEMPTY] The named directory contains entries other than "." and "..".

[EPERM] The pathname contains a character with the high-order bit set.

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

[ENOENT] The named file does not exist.

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

[EACCES] Write permission is denied on the directory containing the link to be removed.

[EBUSY] The directory to be removed is the mount point for a mounted file system.

[EROFS] The directory entry to be removed resides on a read-only file system.

[EFAULT] *Path* points outside the process's allocated address space.

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

RETURN VALUE

A 0 is returned if the remove succeeds; otherwise a -1 is returned and an error code is stored in the global location *errno*.

SEE ALSO

mkdir(2), unlink(2).

NAME

select - synchronous i/o multiplexing

SYNOPSIS

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

int select(nfds, readfds, writefds, execptfds, timeout)
int nfound, nfds, *readfds, *writefds, *execptfds;
struct timeval *timeout;
```

DESCRIPTION

Select examines the i/o descriptors specified by the bit masks *readfds*, *writefds*, and *execptfds* to see if they are ready for reading, writing, or have an exceptional condition pending, respectively. File descriptor *f* is represented by the bit "1<<f" in the mask. *Nfds* desciptors are checked, i.e. the bits from 0 through *nfds*-1 in the masks are examined.

Select returns, in place, a mask of those descriptors which are ready. The total number of ready descriptors is returned as the functional value.

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 *execptfds* may be given as 0 if no descriptors are of interest.

The descriptor masks are always modified on return, even if the call returns as the result of the timeout.

An error return from *select* indicates:

[EBADF] One of the bit masks specified an invalid descriptor.

[EINTR] An signal was delivered before any of the selected events occurred, or the time limit expired.

RETURN VALUE

Select returns the number of descriptors which are contained in the bit masks, or -1 if an error occurred. If the time limit expires then *select* returns 0.

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* (see *intro*(2) for definitions of values and permissions):

GETVAL — Return the value of semval. {READ}

SETVAL — Set the value of semval to *arg.val*. {ALTER} When this *cmd* is successfully executed, the *semadj* value (see *exit*(2)) 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:

IP_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.[c]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 superuser or to the value of sem_perm.[c]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 superuser or to the value of sem_perm.[c]uid in the data structure associated with *semid*.

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 superuser and is not equal to the value of sem_perm.[c]uid in the data structure associated with *semid*.

[EFAULT] *Arg.buf* points to an illegal address.

RETURN VALUE

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.

When *semctl* is unsuccessful, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

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

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 initialised 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.

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 semaphores system wide would be exceeded.

[EEXIST] A semaphore identifier exists for *key* but (*semflg* & IPC_CREAT & IPC_EXCL) is "true".

RETURN VALUE

Upon successful completion, a non-negative integer (i.e., a semaphore identifier) is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

semctl(2), semop(2).

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[];
int nsops;
```

DESCRIPTION

Semop is used to atomically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *Sops* is a pointer to the array of semaphore-operation structures. *Nsops* is the number of such structures in the array. 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 (see semaphore data structure in *intro*(2)):

1. *sem_op* is a negative integer

   One of the following occurs: {ALTER}

   If *semval* 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* returns immediately.

If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is "false", *semop* increments the semncnt associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

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).

2. *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)

3. *sem_op* is zero

One of the following occurs: (READ)

If semval is zero, *semop* returns immediately.

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

If semval is not equal to zero and (*sem_flg* & IPC_NOWAIT) is "false", *semop* increments the semzcnt associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

Semval becomes zero, at which time the value of semzcnt associated with the specified semaphore is decremented.

The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, *errno* is set equal to EIDRM and a value of -1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented and the calling process resumes execution in the manner prescribed in *signal*(2).

Semop fails 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 a 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.

RETURN VALUE

If *semop* returns due to the receipt of a signal, a value of -1 is returned to the calling process and *errno* is set to EINTR. If it returns due to the removal of a *semid* from the system, a value of -1 is returned and *errno* is set to EIDRM.

Upon successful completion, the value of semval at the time of the call for the last operation in the array pointed to by *sops* is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

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

NAME

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

SYNOPSIS

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

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

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

int sendmsg(s, msg, flags)
int 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 be set to SOF_OOB to send "out-of-band" data on sockets which support this notion (e.g. SOCK_STREAM).

See *recv*(2) for a description of the *msghdr* structure. The system calls may fail if:

[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 the message be sent atomically, but the size of the message to be sent made this impossible.

[EWOULDBLOCK] The socket is marked non-blocking and the requested operation would block.

RETURN VALUE

The call returns the number of characters sent, or -1 if an error occurred. In this case *errno* is set to show the error.

SEE ALSO

recv(2), socket(2).

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.

RETURN VALUE

Setpgrp returns the value of the new process group ID.

SEE ALSO

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

NAME

setregid - set real and effective group ID

SYNOPSIS

int setregid(rgid, egid**)**
int rgid, egid;

DESCRIPTION

The real and effective group ID's of the current process are set to the arguments. Only the super-user may change the real group ID of a process. Unprivileged users may change the effective group ID to the real group ID, but to no other.

Supplying a value of -1 for either the real or effective group ID forces the system to substitute the current ID in place of the -1 parameter. *Setregid* will fail if:

[EPERM] The current process is not the super-user and a change other than changing the effective group-id to the real group-id was specified.

RETURN VALUE

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

SEE ALSO

getgid(2), setreuid(2), setgid(3).

NAME

setreuid - set real and effective user ID's

SYNOPSIS

int setreuid(ruid, euid)
int ruid, euid;

DESCRIPTION

The real and effective user ID's of the current process are set according to the arguments. If *ruid* or *euid* is -1, the current uid is filled in by the system. Only the super-user may modify the real uid of a process. Users other than the super-user may change the effective uid of a process only to the real uid.

Setreuid will fail if:

[EPERM] The current process is not the super-user and a change other than changing the effective user-id to the real user-id was specified.

RETURN VALUE

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

SEE ALSO

getuid(2), setregid(2), setuid(2).

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 superuser, 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 superuser 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 superuser, 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*) fails if one or more of the following are true:

[EPERM] The real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not superuser.

[EINVAL] The *uid* is out of range.

RETURN VALUE

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

SEE ALSO

getuid(2), intro(2).

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 *cmds* 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 either that of super-user or to the value of shm_perm.[c]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 either that of super-user or to the value of shm_perm.[c]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.

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 effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of shm_perm.[c]uid in the data structure associated with *shmid*.

[EPERM] *Cmd* is equal to SHM_LOCK or effective user ID of the calling process is not equal to that of super-user.

[EINVAL] *Cmd* is equal to SHM_UNLOCK and the shared memory segment specified by *shmid* is not locked in memory.

[EFAULT] Buf points to an illegal address.

RETURN VALUE

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

SEE ALSO

shmget(2), shmop(2).

NAME

shmget - get shared memory segment

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 *size* bytes (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 shared memory identifier associated with it, and (*shmflg* & IPC_CREAT) is "true".

The *size* of a shared memory segment is limited to 256 Kbytes.

Upon creation, the data structure associated with the new shared memory identifier is initialised 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.

Shmget fails 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 (256 Kbytes).

[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 physical memory is not sufficient to fill the request.

[EEXIST] A shared memory identifier exists for *key* but (*shmflg* & IPC_CREAT & IPC_EXCL) is "true".

RETURN VALUE

Upon successful completion a non-negative integer, i.e., a shared memory identifier, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

shmctl(2), shmop(2).

NAME

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 SHML8A)).

If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is "false", the segment is attached at the address given 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}.

Shmat fails and does 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.

Shmdt detaches the shared memory segment located at the address specified by *shmaddr* from the calling process's data segment.

Shmdt fails and does not detach the shared memory segment if:

[EINVAL] *Shmaddr* is not the data segment start address of a shared memory segment.

RETURN VALUES

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.

SEE ALSO

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

NAME

shutdown - shut down part of a full-duplex connection

SYNOPSIS

```
int shutdown(s, how)
int s, how;
```

DESCRIPTION

The *shutdown* call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

The call succeeds unless:

[EBADF] *S* is not a valid descriptor.

[ENOTSOCK] *S* is a file, not a socket.

[ENOTCONN] The specified socket is not connected.

RETURN VALUE

A 0 is returned if the call succeeds, -1 if it fails. In the latter case, *errno* is set to show the error.

SEE ALSO

connect(2), socket(2).

NAME

sigblock - block signals

SYNOPSIS

```
int sigblock(mask);
int mask;
```

DESCRIPTION

Sigblock causes the signals specified in *mask* to be added to the set of signals currently being blocked from delivery. Signal *i* is blocked if the *i*-th bit in *mask* is a 1.

It is not possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

RETURN VALUE

The previous set of masked signals is returned.

SEE ALSO

kill(2), sigvec(2), sigsetmask(2),.

NAME

signal - specify what to do upon receipt of a signal

SYNOPSIS

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

int (*signal(sig, func))()
int sig;
int (*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* quit
SIGILL	04* illegal instruction (not reset when caught)
SIGTRAP	05* trace trap (not reset when caught)
SIGIOT	06* IOT instruction
SIGEMT	07* EMT instruction
SIGFPE	08* floating point exception
SIGKILL	09 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 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 death of a child (see WARNING below)

SIGPWR 19 power fail (see WARNING below)

See below for the significance of the asterisk (*) in the above list.

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 - 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); a "core image" is 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
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* is passed as the first argument to the signal-catching function. Additional arguments can be passed to the signal-catching function for hardware-generated signals. Before entering the signal-catching function, the value of *func* for the caught signal is set to SIG_DFL unless the signal is SIGILL, SIGTRAP, or SIGPWR.

Upon return from the signal-catching function, the receiving process resumes execution at the point it was interrupted. See the WARNINGS section below.

When a signal that is to be caught occurs during a *read*(2), *write*(2), *open*(2), or *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 is executed; then the interrupted system call returns a -1 to the calling process with *errno* set to EINTR.

Note: the signal SIGKILL cannot be caught.

A call to signal cancels a pending signal *sig* except for a pending SIGKILL signal. *Signal* fails if one or more of the following are true:

[EINVAL] *Sig* is an illegal signal number, including SIGKILL.

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

RETURN VALUE

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), kill(2), pause(2), ptrace(2), wait(2), setjmp(3C).

WARNING

Two other signals that behave differently than the signals described above exist in this release of the system. They are:

SIGCLD 18 death of a child (reset when caught)

SIGPWR 19 power fail (not reset when caught)

There is no guarantee that, in future releases of the X/OS System, these signals will continue to behave as described below; they are included only for compatibility with other versions of the UNIX System. Their use in new programs is strongly discouraged.

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. If *sig* is SIGCLD, the calling process's child processes do 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, except that, while

the process is executing the signal-catching function, any received SIGCLD signals are queued and the signal-catching function is continually reentered until the queue is empty.

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* blocks until all of the calling process's child processes terminate; it then returns 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 does not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding 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.

The ability to resume execution upon return from the signal-catching function is machine-dependent.

NAME

sigpause - atomically release blocked signals and wait for interrupt

SYNOPSIS

```
int sigpause(sigmask)
int sigmask;
```

DESCRIPTION

Sigpause assigns *sigmask* to the set of masked signals and then waits for a signal to arrive; on return the set of masked signals is restored. *Sigmask* is usually 0 to indicate that no signals are now to be blocked. *Sigpause* always terminates by being interrupted, returning EINTR.

In normal usage, a signal is blocked using *sigblock*(2), to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using *sigpause* with the mask returned by *sigblock*.

SEE ALSO

sigblock(2), sigvec(2).

NAME

sigsetmask - set current signal mask

SYNOPSIS

```
int sigsetmask(mask);
int mask;
```

DESCRIPTION

Sigsetmask sets the current signal mask (those signals which are blocked from delivery). Signal *i* is blocked if the *i*-th bit in *mask* is a 1.

The system quietly disallows SIGKILL, SIGSTOP, or SIGCONT to be blocked.

RETURN VALUE

The previous set of masked signals is returned.

SEE ALSO

kill(2), sigvec(2), sigblock(2), sigpause(2).

NAME

sigstack - set and/or get signal stack context

SYNOPSIS

```
#include <signal.h>

struct sigstack {
        caddr_t   ss_sp;
        int       ss_onstack;
};

int sigstack(ss, oss);
struct sigstack *ss, *oss;
```

DESCRIPTION

Sigstack allows users to define an alternate stack on which signals are to be processed. If *ss* is non-zero, it specifies a *signal stack* on which to deliver signals and tells the system if the process is currently executing on that stack. When a signal's action indicates that its handler should execute on the signal stack (specified with a *sigvec*(2) call), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution. If *oss* is non-zero, the current signal stack state is returned.

NOTES

Signal stacks are not "grown" automatically, as is done for the normal stack. If the stack overflows, unpredictable results may occur.

RETURN VALUE

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

Sigstack will fail and the signal stack context will remain

unchanged if:

[EFAULT] Either *ss* or *oss* points to memory which is not a valid part of the process address space.

SEE ALSO

sigvec(2), setjmp(3).

NAME

sigvec - software signal facilities

SYNOPSIS

```
#include <signal.h>

struct sigvec {
   int    (*sv_handler)();
   int    sv_mask;
   int    sv_onstack;
};

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

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *blocked* or *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special *signal stack*.

All signals have the same *priority*. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process.

The signal mask for a process is initilised from that of its parent (normally 0). It may be changed with a *sigblock*(2) or *sigsetmask*(2) call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process, a new signal mask is installed for the duration of the process's signal handler (or until a *sigblock* or *sigsetmask* call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and or-ing in the signal mask associated with the handler to be invoked.

Sigvec assigns a handler for a specific signal. If *vec* is non-zero, it specifies a handler routine and mask to be used when delivering the specified signal. Further, if *sv_onstack* is 1, the system will deliver the signal to the process on a *signal stack*, specified with *sigstack*(2). If *ovec* is non-zero, the previous handling information for the signal is returned to the user.

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, blocked, 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 one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user defined signal usr1
SIGUSR2	17	user defined signal usr2
SIGCLD	18	to parent on child stop or exit
SIGPWR	19	power fail
SIGURG	20@	urgent condition present on socket
SIGSTOP	21+	stop (cannot be caught, blocked, or ignored)
SIGTSTP	22+	stop signal generated from keyboard
SIGCONT	23@	continue after stop (cannot be blocked)
SIGCHLD	18@	child status has changed
SIGTTIN	24+	background read attempted from control terminal
SIGTTOU	25+	background write attempted to control terminal
SIGIO	26@	i/o is possible on a descriptor (see *fcntl*(2))
SIGXCPU	27	cpu time limit exceeded (see *setrlimit*(2))
SIGXFSZ	28	file size limit exceeded (see *setrlimit*(2))
SIGVTALRM	29	virtual time alarm (see *setitimer*(2))
SIGPROF	30	profiling timer alarm (see *setitimer*(2))

The starred signals in the list above cause a core image if not caught or ignored.

Once a signal handler is installed, it remains installed until another *sigvec* call is made, or an *execve*(2) is performed. This differs from the way *signal*(2) works. The default action for a signal may be reinstated by setting *sv_handler* to SIG_DFL; 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 *sv_handler* is SIG_IGN the signal is subsequently ignored, and pending instances of the signal are discarded.

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* or *write*(2) on a slow device (such as a terminal; but not a file) and during a *wait*(2).

After a *fork*(2) the child inherits all signals, the signal mask, and the signal stack.

Execve(2) resets all caught signals to default action; ignored signals remain ignored; the signal mask remains the same; the signal stack state is reset.

NOTES

The mask specified in *vec* is not allowed to block SIGKILL, SIGSTOP, or SIGCONT. This is done silently by the system. *Sigvec* will fail and no new signal handler will be installed if one of the following occurs:

[EFAULT] Either *vec* or *ovec* points to memory which is not a valid part of the process address space.

[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).

RETURN VALUE

A value of 0 indicates that the call succeeded. A -1 return value indicates an error occurred and *errno* is set to indicate the reason.

SEE ALSO

kill(1), ptrace(2), kill(2), sigblock(2), signal(2), sigsetmask(2), sigpause(2) sigstack(2), sigvec(2), setjmp(3).

NAME

socket - create an endpoint for communication

SYNOPSIS

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

int socket(af, type, protocol)
int af, type, protocol;
```

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor. The *af* parameter specifies an address format with which addresses specified in later operations using the socket should be interpreted. These formats are defined in the include file *<sys/socket.h>*. The currently understood formats are:

```
AF_UNIX (UNIX/X/OS path names)
AF_INET (ARPA Internet addresses)
AF_PUP (Xerox PUP-I Internet addresses)
AF_IMPLINK (IMP "host at IMP" addresses)
```

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 with an out-of-band data transmission mechanism. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network interfaces. The types SOCK_RAW, which is available only to the super-user, and

SOCK_SEQPACKET and SOCK_RDM, which are 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 using a given address format. 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 *services*(3N) and *protocols*(3N). 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_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2) calls. It is also possible to receive datagrams at such a socket with *recv*(2).

An *fcntl*(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives.

The operation of sockets is controlled by socket level *options*.

These options are defined in the file *<sys/socket.h>* and explained below. *Setsockopt* and *getsockopt*(2) are used to set and get options, respectively. The options are:

SO_DEBUG	turn on recording of debugging information
SO_REUSEADDR	allow local address reuse
SO_KEEPALIVE	keep connections alive
SO_DONTROUTE	do no apply routing on outgoing messages
SO_LINGER	linger on close if data present
SO_DONTLINGER	do not linger on close

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 and SO_DONTLINGER control the actions taken when unsent messages are queued on a 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_DONTLINGER is specified and a *close* is issued, the system will process the close in a manner which allows the process to continue as quickly as possible. The *socket* call fails if:

[EAFNOSUPPORT] The specified address family is not supported in this version of the system.

[ESOCKTNOSUPPORT] The specified socket type is not supported in this address family.

[EPROTONOSUPPORT] The specified protocol is not supported.

[EMFILE] The per-process descriptor table is full.

[ENOBUFS] No buffer space is available. The socket cannot be created.

RETURN VALUE

A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), get sockopt(2), ioctl(2), listen(2), recv(2), select(2), send(2), shutdown(2), socketpair(2).
TCP/IP User Manual.

NAME

socketpair - create a pair of connected sockets

SYNOPSIS

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

int socketpair(d, type, protocol, sv)
int d, type, protocol;
int sv[2];
```

DESCRIPTION

The *socketpair* call creates an unnamed pair of connected sockets in the specified domain *d*, of the specified *type*, and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv*[0] and *sv*[1]. The two sockets are indistinguishable.

This call is currently implemented only for the X/OS domain.

The call succeeds unless:

[EMFILE]	Too many descriptors are in use by this process.
[EAFNOSUPPORT]	The specified address family is not supported on this machine.
[EPROTONOSUPPORT]	The specified protocol is not supported on this machine.
[EOPNOSUPPORT]	The specified protocol does not support creation of socket pairs.
[EFAULT]	The address *sv* does not specify a valid part of the process address space.

RETURN VALUE

A 0 is returned if the call succeeds, -1 if it fails. In the latter case, *errno* is set to show the error.

SEE ALSO

read(2), write(2), pipe(2).

NAME

stat, lstat, fstat - get file status

SYNOPSIS

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

int stat(path, buf)
char *path;
struct stat *buf;

int lstat(path, buf)
char *path;
struct stat *buf;

int fstat(fd, buf)
int fd;
struct stat *buf;
```

DESCRIPTION

Stat obtains information about the file *path.* Read, write or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be reachable.

Lstat is like *stat* except in the case where the named file is a symbolic link, in which case *lstat* returns information about the link, while *stat* returns information about the file the link references. *Fstat* obtains the same information about an open file referenced by the argument descriptor, such as would be obtained by an *open* 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:

```
dev_t     st_dev;                    /* device inode resides on */
ino_t     st_ino;                       /* this inode's number */
ushort    st_mode;                              /* protection */
short     st_nlink;          /* number or hard links to the file */
short     st_uid;                             /* user-id of owner */
short     st_gid;                            /* group-id of owner */
dev_t     st_rdev;                             /* the device type */
                          /* (only for inode that is device) */
off_t     st_size;                           /* total size of file */
time_t    st_atime;                    /* file last access time */
time_t    st_mtime;                    /* file last modify time */
time_t    st_ctime;               /* file last status change time */
long      st_blksize;   /* optimal blocksize for file system i/o */
long      st_blocks;        /* actual number of blocks allocated */
```

The three *time_t* items have the following properties:

st_atime Time when file data was last read or modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), *utimes*(2), *read*(2), and *write*(2).

st_mtime Time when data was last modified. It is not set by changes of owner, group, link count, or mode. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), *utimes*(2), *write*(2).

st_ctime Time when file status was last changed. It is set both by writing and changing the inode. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), *utimes*(2), *write*(2).

The status information word *st_mode* has bits:

```
#define    S_IFMT   0170000                  /* type of file */
#define    S_IFIFO  0010000                     /* FIFO file */
#define    S_IFCHR  0020000           /* character special */
#define    S_IFDIR  0040000                    /* directory *
#define    S_IFBLK  0060000                 /* block special *
#define    S_IFREG  0100000                       /* regular *,
#define    S_IFLNK  0120000                 /* symbolic link */
```

```
#define    S_IFSOCK 0140000                          /* socket */
#define    S_ISUID  0004000        /* set user id on execution */
#define    S_ISGID  0002000       /* set group id on execution */
#define    S_ISVTX  0001000 /*save swapped text even after use */
#define    S_IREAD  0000400          /* read permission, owner */
#define    S_IWRITE 0000200         /* write permission, owner */
#define    S_IEXEC  0000100 /* execute/search permission, owner */
```

The mode bits 0000070 and 0000007 encode "group" and "others" permissions respectively (see *chmod*(2)).

When *fd* is associated with a pipe, *fstat* reports an ordinary file with an inode number, restricted permissions, and a not necessarily meaningful length.

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 pathname was too long.

[ENOENT] The named file does not exist.

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

[EFAULT] *Buf* or *name* points to an invalid address.

Fstat will fail if one or both of the following are true:

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

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

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

CAVEAT

The fields in the stat structure currently marked *st_spare1*, *st_spare2*, and *st_spare3* are reserved for future use.

RETURN VALUE

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

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), time(2), unlink(2), utime(2), utimes(2), stat(5).

NAME

statfs - get file system statistics

SYNOPSIS

```
#include <sys/vfs.h>
int statfs(path, buf)
char *path;
struct statfs *buf;
int fstatfs(fd, buf)
int fd;
struct statfs *buf;
```

DESCRIPTION

Statfs returns information about a mounted file system.

Path is the pathname of any file within the mounted filesystem.

Buf is a pointer to a *statfs* structure defined as follows:

```
typedef struct {
   long    val[2];
} fsid_t;
struct statfs {
   long    f_type;     /* type of info, zero for now */
   long    f_bsize;    /* fundamental file system block size */
   long    f_blocks;   /* total blocks in file system */
   long    f_bfree;    /* free blocks */
   long    f_bavail;   /* free blocks available to non-superuser */
   long    f_files;    /* total file nodes in file system */
   long    f_ffree;    /* free file nodes in fs */
   fsid_t  f_fsid;     /* file system id */
   long    f_spare[7]; /* spare for later */
   };
```

Fields that are undefined for a particular file system are set to -1. *Fstatfs* returns the same information about an open file referenced by descriptor *fd*.

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

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

[EPERM] The pathname contains a character with the high-order bit set.

[ENAMETOOLONG] The pathname was too long.

[ENOENT] The named file does not exist.

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

[EFAULT] *Buf* or *name* points to an invalid address.

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

[EIO] An I/O error occurred while reading from or writing to the file system.

Fstatfs fails if one or both of the following are true:

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

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

[EIO] An I/O error occurred while reading from or writing to the file system.

RETURN VALUE

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

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.

Stime fails if:

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

RETURN VALUE

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

SEE ALSO

time(2).

NAME

swapon - add a swap device for interleaved paging/swapping

SYNOPSIS

```
int swapon(special)
char *special;
```

DESCRIPTION

Swapon makes the block device *special* available to the system for allocation for paging and swapping. The names of potentially available devices are known to the system and defined at system configuration time. The size of the swap area on *special* is calculated at the time the device is first made available for swapping.

There is no way to stop swapping on a disk so that the pack may be dismounted.

SEE ALSO

swapon(8), config(8).

NAME

symlink - make symbolic link to a file

SYNOPSIS

```
int symlink(name1, name2)
char *name1, *name2;
```

DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the 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. The symbolic link is made unless one or more of the following are true:

[ENOENT] One of the pathnames specified was too long.

[ENOTDIR] A component of the *name2* prefix is not a directory.

[EEXIST] *Name2* already exists.

[EACCES] A component of the *name2* path prefix denies search permission.

[EROFS] The file *name2* would reside on a read-only file system.

[EFAULT] *Name1* or *name2* points outside the process's allocated address space.

[ELOOP] Too may symbolic links were encountered in translating the pathname.

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.

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 inodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*(1M) and *df*(1M). It is mandatory before a boot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

SEE ALSO

System Administration Utilities Reference Manual

NAME

time - get time

SYNOPSIS

long time((long *) 0)

long time(tloc**)**
long *tloc;

DESCRIPTION

Time returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* (taken as an integer) is non-zero, the return value is also stored in the location to which *tloc* points.

Time fails if:

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

RETURN VALUE

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

SEE ALSO

stime(2).

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. Contents of the structure are:

```
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 in 60ths of a second.

Tms_utime is the CPU time used while executing instructions in the user space of the calling process. *Tms_stime* is the CPU time used by the system on behalf of the calling process. *Tms_cutime* is the sum of the *tms_utimes* and *tms_cutimes* of the child processes.

Tms_cstime is the sum of the *tms_stimes* and *tms_cstimes* of the child processes.

Times fails if:

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

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in 60ths of a second, since system start-up time. If *times* fails, a -1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

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:

1 Get the process's file size limit. 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 process's file size limit to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of superuser may increase the limit. *Ulimit* fails and the limit remains unchanged if a process with an effective user ID other than superuser attempts to increase its file size limit ([EPERM]).

3 Get the maximum possible break value. See *brk*(2).

RETURN VALUE

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

SEE ALSO

brk(2), write(2).

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.

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

mkdir(1), sh(1), chmod(2), creat(2), mkdir(2), mknod(2), open(2).

NAME

uname - get name of current operating system

SYNOPSIS

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

int uname(name)
struct utsname *name;
```

DESCRIPTION

Uname stores information identifying the current system in the structure pointed to by *name*.

Uname uses the structure defined in <sys/utsname.h> whose members are:

```
char sysname[9];
char nodename[9];
char release[9];
char version[9];
char machine[9];
```

Uname returns a null-terminated character string naming the current system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system. *Machine* contains a standard name that identifies the hardware that the system is running on.

Uname fails if:

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

RETURN VALUE

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

SEE ALSO

uname(1).

NAME

unlink - remove directory entry

SYNOPSIS

```
int unlink(path)
char *path;
```

DESCRIPTION

Unlink removes the directory entry named by the pathname pointed to by *path*.

The named file is unlinked unless 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.

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

RETURN VALUE

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

SEE ALSO

rm(1), close(2), link(2), open(2), rmdir(2).

NAME

ustat - get file system statistics

SYNOPSIS

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

int ustat(dev, buf)
int 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 */
```

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.

RETURN VALUE

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

SEE ALSO

stat(2), fs(4).

NAME

utime, utimes - set file access and modification times

SYNOPSIS

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

int utime(path, times)
char *path;
struct utimbuf *times;

#include <sys/time.h>

int utimes(path, tvp)
char *path;
struct timeval *tvp[2];
```

DESCRIPTION

Utime and *utimes* set the access and modification times of the file *path*.

Utime sets the access and modification times of the file according to the contents of the structure *times* points to.

If *times* is NULL, *utime* sets the access and modification times of the file to the current time.

If *times* is not NULL, it must point to a structure defined as follows:

```
struct utimbuf {
   time_t actime;         /* access time */
   time_t modtime;  /* modification time */
);
```

The times in the structure are measured in seconds since 00:00:00 GMT, 1 Jan 1970.

The *utimes* call uses the "accessed" and "updated" times, in that order, from the *tvp* vector to set the corresponding recorded times for *path*.

The times are specified in seconds and microseconds, according to the following structure:

```
struct timeval {
    u_long tv_sec;    /* seconds since Jan 1, 1970 */
    long tv_usec;             /* and microseconds */
};
```

A process must be the super-user or the owner of the file or have write permission to use *utime* or *utimes*.

Utime and *Utimes* will fail if one or more of the following are true:

[ENOENT] The pathname was too long.

[ENOENT] The named file does not exist.

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

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

[EPERM] The process is not super-user and not the owner of the file.

[EACCES] The effective user ID is not super-user and not the owner of the file, *times* is NULL and write access is denied.

[EROFS] The file system containing the file is mounted read-only.

[EFAULT] *Tvp* or *times* points outside the process's allocated address space.

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

RETURN VALUE

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

SEE ALSO

stat(2), gettimeofday(2).

NAME

wait, wait3 - wait for child process to stop or terminate

SYNOPSIS

```
int wait(stat_loc)
int *stat_loc;

int wait((int *)0)

#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

int wait3(status, options, rusage)
union wait *status;
int options;
struct rusage *rusage;
```

DESCRIPTION

wait

Wait suspends the calling process until it receives a signal that is to be caught (see *signal*(2)), or until any one of the calling process's child processes stops in a trace mode (see *ptrace*(2)) or terminates. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) 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. 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 contain the number of the signal that caused the process to stop and the low-order 8 bits are set equal to 0177.

If the child process terminated due to an *exit* call, the low-order 8 bits of status are zero and the high-order 8 bits 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 are zero and the low-order 8 bits contain the number of the signal that caused the termination. In addition, if the low-order seventh bit (i.e., octal bit 200) is set, a "core image" will have been produced; see *signal*(2).

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 initialisation process inherits the child processes; see *intro*(2).

A precise definition of the status word *stat_loc* points to (effectively a union) is given in <*sys/wait.h*>.

wait3

Wait3 provides an alternate interface for programs which must not block when collecting the status of child processes.

The *status* parameter is defined in <*sys/wait.h*>.

The *options* parameter takes the following values:

WNOHANG This is used to indicate that the call should not block if there are no processes which wish to report status.

WUNTRACED This is used to indicate that only children of the current process which are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal should have their status reported.

When the WNOHANG option is specified and no processes wish to report status, *wait3* returns a *pid* of 0. The WNOHANG and WUNTRACED options may be combined by or-ing the two values.

If the *rusage* parameter is non-zero, a summary of the resources used by the terminated process and all its children is returned

(this information is currently not available for stopped processes).

NOTES

See *sigvec*(2) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted; see *ptrace*(2). If the 0200 bit of the termination status is set, a core image of the process is produced by the system.

If the parent process terminates without waiting for its children, the initialisation process (process ID = 1) inherits the children.

Wait and *wait3* are automatically restarted when a process receives a signal while awaiting termination of a child process, provided it is using the new style signal facilities (see *sigvec*(2)) rather than the old style ones (see *signal*(2)).

Wait will fail and return immediately if one or more of the following are true:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EFAULT] The *status* or *rusage* arguments point to an illegal address.

RETURN VALUE

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.

Wait3 returns -1 if there are no children not previously waited for; 0 is returned if WNOHANG is specified and there are no stopped or exited children.

SEE ALSO

exec(2), exit(2), fork(2), pause(2), signal(2), sigvec(2).

NAME

write, writev - write to a file

SYNOPSIS

```
int write(fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

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

int writev(fildes, iov, ioveclen)
int fildes;
struct iovec *iov;
int ioveclen;
```

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*. *Writev* performs the same action, but gathers the output data from the *iovlen* buffers specified by the members of the *iovec* array: iov[0], iov[1], etc. (see *readv*(2)).

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 file status flag is set, the file pointer is set to

the end of the file prior to each write.

Write and *writev* fail and the file pointer remains unchanged if one or more of the following are true:

[EBADF] *Fildes* is not a valid file descriptor open for writing.

[EPIPE] An attempt is made to write to a pipe that is not open for reading by any process. (SIGPIPE signal)

[EFBIG] An attempt is made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2).

[EFAULT] The data to be written lies outside the process's allocated address space.

[EINTR] A signal was caught during the *write* system call. If a *write* requests that more bytes be written than there is room for (e.g., it requests more than the ulimit (see *ulimit*(2)) or it tries to write off the physical end of a medium), then only as many bytes as there is room for are written. For example, if there is space for 20 bytes more in a file before reaching a limit, a *write* of 512 bytes actually writes 20. The next *write* of a non-zero number of bytes gives a failure return (except as noted below).

If the file being written is a pipe (or FIFO), no partial writes are permitted. Thus, *write* attempts, which would exceed a limit if carried out, will fail. The exact nature of such failures depends on the value of the O_NDELAY bit of the file flag word. If this bit is set, then write to a full pipe (or FIFO) returns a count of 0. If it is clear, *write* attempts to a full pipe (or FIFO) will block until space becomes available.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), lseek(2), open(2), pipe(2), readv(2), ulimit(2).

NAME

intro - introduction to file formats

DESCRIPTION

This section outlines the format of various files. The C struct declarations for the file formats are given where appliceble. Usually, these structures can be found in the directories /usr/include or /usr/include/sys.

References of the type *name*(1M) refer to entries found in Section 1 of the *System Administration Reference Manual*.

NAME

a.out - common assembler and link editor output

DESCRIPTION

The file name **a.out** is the output file from the assembler *as*(1) and the link editor *ld*(1). Both programs will make **a.out** executable if there were no errors in assembling or linking and no unresolved external references.

A common object file consists of a file header, an X/OS system header, a table of section headers, data sections, relocation information, (optional) line numbers, a symbol table, and a string table. The order is given below.

```
File header.
X/OS system header.
Section 1 header.
...
Section n header.
Section 1 data.
...
Section n data.
Section 1 relocation.
...
Section n relocation.
Section 1 line numbers.
...
Section n line numbers.
Symbol table.
String table.
```

The last three parts of an object file (line numbers, symbol table and string table) may be missing if the program was linked with the **-s** option of *ld*(1) or if they were removed by *strip*(1). Also note that the relocation information will be absent if there were no unresolved external references after linking. The string table exists only if the symbol table contains symbols with names longer

than eight characters.

The sizes of each section (contained in the header, discusse below) are in bytes and are even.

When an **a.out** file is loaded into memory for execution, three logical segments are set up: the text segment (executable code), the data segment (initialised data followed by uninitialised, the latter actually being initialised to zeroes), and a stack. On LSX computers the text segment starts at location 0x80000000.

The **a.out** file produced by *ld*(1) by default contains two *magic numbers*, in the file header and the X/OS system header. The default values of these numbers are 0x150 and 0x108 respectively. The headers (file header, X/OS system header, and section headers) are loaded at the beginning of the text segment and the text immediately follows the headers in the user address space. The first text address will equal the size of the headers, and will vary depending upon the number of section headers in the **a.out** file.

In an **a.out** file with three sections (**.text**, **.data**, and **.bss**), th first text address is at 0x80000200. The text segment is not writable by the program; if other processes are executing the same **a.out** file, the processes will share a single text segment.

The data segment starts at 0x80c00000.

The stack begins at 0x80ffffc and grows downwards (towards lower addresses). The stack is automatically extended as required; its maximum size is 512 kilobytes. The data segment is extended only as requested by the *brk*(2) system call.

The value of a word in the text or data portions that is not a reference to an undefined external symbol is exactly the value that will appear in memory when the file is executed. If a word in the text involves a reference to an undefined external symbol, th storage class of the symbol-table entry for that word will b marked as an "external symbol", and the section number will be se to zero. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be

added to the word in the file.

File Header

The format of the **filehdr** header is:

```
struct filehdr
{
    unsigned short f_magic;       /* magic number */
    unsigned short f_nscns;       /* number of sections */
    long       f_timdat;          /* time and date stamp */
    long       f_symptr;          /* file ptr to symtab */
    long       f_nsyms;           /* # symtab entries */
    unsigned short f_opthdr;      /* sizeof(opt hdr) */
    unsigned short f_flags;       /* flags */
};
```

X/OS System Header

The format of the X/OS system header is:

```
typedef struct aouthdr
{
    short  magic;        /* magic number */
    short  vstamp;       /* version stamp */
    long   tsize;        /* text size in bytes, padded */
    long   dsize;        /* initialised data (.data) */
    long   bsize;        /* uninitialised data (.bss) */
    long   entry;        /* entry point */
    long   text_start;   /* base of text used for this file */
    long   data_start;   /* base of data used for this file */
} AOUTHDR;
```

Section Header

The format of the **section** header is:

```
struct scnhdr
{
    char        s_name[8];          /* section name */
    long        s_paddr;            /* physical address */
    long        s_vaddr;            /* virtual address */
    long        s_size;             /* section size */
    long        s_scnptr;           /* file ptr to raw data */
    long        s_relptr;           /* file ptr to relocation */
    long        s_lnnoptr;          /* file ptr to line numbers */
    unsigned short s_nreloc;        /* # reloc entries */
    unsigned short s_nlnno;         /* # line number entries */
    long        s_flags;            /* flags */
};
```

Relocation

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 */
     unsigned short r_type;   /* relocation type */
};
```

The start of the relocation information is *s_relptr* from the section header. If there is no relocation information, *s_relptr* is zero.

Symbol Table

The format of each symbol in the symbol table is:

```
#define SYMNMLEN 8
#define FILNMLEN 14
#define SYMESZ  18              /* the size of a SYMENT */

struct syment
{
  union                         /* get a symbol name */
  {
    char    _n_name[SYMNMLEN]; /* name of symbol */
    struct
    {
      long  _n_zeroes;      /* == 0L if in string table */
      long  _n_offset;      /* location in string table */
    } _n_n;
    char    *_n_nptr[2];    /* allows overlaying */
  } _n;
  long       n_value;       /* value of symbol */
  short      n_scnum;       /* section number */
  unsigned short n_type;    /* type and derived type */
  char      n_sclass;       /* storage class */
  char      n_numaux;       /* number of aux entries */
};

#define n_name  _n._n_name
#define n_zeroes _n._n_n._n_zeroes
#define n_offset _n._n_n._n_offset
#define n_nptr  _n._n_nptr[1]
```

Some symbols require more information than a single entry; they are followed by *auxiliary entries* that are the same size as a symbol entry. The format follows:

```
union auxent {
    struct {
       long  x_tagndx;
       union {
           struct {
               unsigned short x_lnno;
```

```
                unsigned short x_size;
            } x_lnsz;
            long   x_fsize;
        } x_misc;
        union {
            struct {
                long  x_lnnoptr;
                long  x_endndx;
            } x_fcn;
            struct {
                unsigned short x_dimen[DIMNUM];
            } x_ary;
        } x_fcnary;
        unsigned short x_tvndx;
    } x_sym;

    struct {
        char  x_fname[FILNMLEN];
    } x_file;

    struct {
        long    x_scnlen;
        unsigned short x_nreloc;
        unsigned short x_nlinno;
    } x_scn;

    struct {
        long      x_tvfill;
        unsigned short x_tvlen;
        unsigned short x_tvran[2];
    } x_tv;
};
```

Indexes of symbol table entries begin at *zero*. The start of the symbol table is *f_symptr* (from the file header) bytes from the beginning of the file. If the symbol table is stripped, *f_symptr* is 0. The string table (if one exists) begins at *f_symptr* + (*f_nsyms* * SYMESZ) bytes from the beginning of the file.

SEE ALSO

as(1), cc(1), ld(1), brk(2), filehdr(4), ldfcn(4), linenum(4), reloc(4), scnhdr(4), syms(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  u_short comp_t; /* "floating point" */
        /* 13-bit fraction, 3-bit exponent */
struct acct
{
    char    ac_comm[10];    /* Accounting command name */
    comp_t  ac_utime;       /* Accounting user time */
    comp_t  ac_stime;       /* Accounting system time */
    comp_t  ac_etime;       /* Accounting elapsed time */
    time_t  ac_btime;       /* Beginning time */
    short   ac_uid;         /* Accounting user ID */
    short   ac_gid;         /* Accounting group ID */
    short   ac_mem;         /* average memory usage */
    comp_t  ac_io;          /* number of disk IO blocks */
    dev_t   ac_tty;         /* control type writer */
    char    ac_flag;        /* Accounting flag */
    char    ac_stat;        /* KEXT Exit status */
    comp_t  ac_rw;          /* KEXT blocks read or written */
};


#define AFORK 01        /* has executed fork, but no exec */
#define ASU   02        /* used superuser privileges */
#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 usin text)

The value of *ac_mem/(ac_stime+ac_utime)* can be viewed as an approximation of the mean process size, as modified by text-sharing.

The structure stored in the file **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(1M), acctcom(1), acct(2).

BUGS

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

aouthdr - optional aout header

SYNOPSIS

```
#include <aouthdr.h>
```

DESCRIPTION

An object file may contain an optional header, following the file header described in *filehdr*(4). Object files that have been completely linked by *ld*(1) contain this header; others do not. The format of the optional header is:

```
typedef struct aouthdr {
    short  magic;       /* magic number */
    short  vstamp;      /* version stamp */
    long   tsize;       /* text size in bytes, padded (.text) */
    long   dsize;       /* initialised data (.data) */
    long   bsize;       /* uninitialised data (.bss) */
    long   entry;       /* entry point */
    long   text_start;  /* base of text used for this file */
    long   data_start;  /* base of data used for this file */
} AOUTHDR;
```

SEE ALSO

a.out(4), filehdr(4).

NAME

core - format of memory image file

SYNOPSIS

#include <sys/param.m68.h>
#include <sys/user.h>

DESCRIPTION

The X/OS 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 constant in the *<sys/param.m68.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.

In general the debugger *sdb*(1) is sufficient to deal with core images.

SEE ALSO

sdb(1), sigvec(2), setrlimit(2)

NAME

cpio - format of cpio archive

DESCRIPTION

When the **-c** option of *cpio*(1) is not used, the file header structure 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%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 items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated pathname *h_name*, including the null byte, is given by *h_namesize* (maximum length 256).

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

cpio(1), find(1), stat(2).

NAME

dir - format of directories

SYNOPSIS

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

DESCRIPTION

A directory behaves exactly like an ordinary file, except 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 inode entry; see *fs*(4). The structure of a directory entry as given in the include file is:

```
/*
 * 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.
 */
#ifdef KERNEL
#define DIRBLKSIZ DEV_BSIZE
#else
#define DIRBLKSIZ 256
#endif

#define MAXNAMLEN 127

/*
 * The 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 DIRSIZ
#define DIRSIZ(dp) \
 ((sizeof (struct direct) - (MAXNAMLEN+1)) +
 (((dp)->d_namlen+1 + 3) &~ 3))

struct direct {
    u_long  d_ino;
    short   d_reclen;
    short   d_namlen;
    char    d_name[MAXNAMLEN + 1];
    /* typically shorter */
};

/* definitions for library routines operating on directories */

typedef struct _dirdesc {
    int     dd_fd;
    long    dd_loc;
    long    dd_size;
    long    dd_bbase:
    long    dd_entno;
```

```
        long    dd_bsize;
        char    *dd_buf;
    } DIR;
```

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(4)

NAME

dkinfo - structure of a disk partition map

SYNOPSIS

#include <sys/dkinfo.h>

DESCRIPTION

A partition map is normally available at a specified offset within each disk drive. This map specifies the characteristics of each disk partition. Generally, a file system is created for each partition (the only partitions without the file system are the swap area and "a").

A partition map can be created or modified by means of the *disconf*(1M) utility.

Disk device drivers read the partition map when a device is first opened. If the partition map is absent, each driver uses a default.

The partition map can be recognised at the specified position from a "magic number". The structure of the map is defined in *<sys/dkinfo.h>*, and is equivalent to the following:

```
#define DK_MAGIC 0x1f397441                          /* magic number */
#define DK_LEN   256                    /* length of dkinfo record */

#define FD_OFF   (daddr_t)24   /* dkinfo disk block begin offset */
#define FD_MBOOTOFF (daddr_t)23          /* begin offset of mboot */
#define MBOOTSIZE (143-FD_MBOOTOFF) /* max size of mboot program */

#define MINBOOTPSIZE (MBOOTSIZE+FD_MBOOTOFF)
                                    /* min size of boot partition */

#define SIZE_20Mb  (20000 * 1024 / DEV_BSIZE)

#define PBOOT 0
```

```
#define PROOT 1
#define PSWAP 2

#define DK_NAME            16    /* length of name label */
#define DK_MIN              8    /* minor numbers available */
#define DK_SPARE          128    /* size of spare area */
#define DK_S24OFF (daddr_t)7     /* "Std. #24" disk block offset */

struct partition {
    long p_cyl;           /* cylinder begin offset */
    long p_size;          /* size of partition in disk blocks */
    long p_block;         /* block begin offset */
};

#define DK_MAPSIZE  (sizeof (struct partition) * DK_MIN)

#define DK_L (sizeof (long) * 3 + DK_MAPSIZE + DK_NAME + DK_SPARE)
#define DK_PAD (DK_LEN - DK_L)    /* padding constant */

struct dkinfo {
    long dk_magic;                  /* magic number */
    char dk_name[DK_NAME];          /* name label of disk */
    long dk_id;                     /* id number of disk */
    long dk_root;                   /* partition containing */
                                    /* ROOT FS */
    struct partition dk_p[DK_MIN]; /* partition table */
    char dk_spare[DK_SPARE];        /* free for user use */
    char dk_pad[DK_PAD];            /* padding to 256 bytes - */
                                    /* reserved */
};

/*  Cyl 0 - Track 0 - Sector 7   */
/*  "Standard 24" format.        */

#define S24_HEOFF     71   /* heads # (surfaces) */
#define S24_CYLOFF    81   /* cylinders on disk */
#define S24_TRKOFF    85   /* sectors per track */
#define S24_SLOFF     88   /* sector length */
#define S24_PCYLOFF 130    /* pre-compensation cylinder */
#define S24_FIRSTST   91   /* first sector # */
```

Note that the *dk_root* field of the *dkinfo* structure is used to flag the partition that the bootstrapper will use as the root file system.

FILES

/usr/include/sys/dkinfo.h

SEE ALSO

disconf(1M)

NAME

dump, dumpdates - incremental dump format

SYNOPSIS

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

DESCRIPTION

Tapes used by *dump* and *restore*(1M) contain:

a header record
two groups of bit map records
a group of records describing directories
a group of records describing files

The format of the header record and of the first record of each description as given in the include file <*dumprestor.h*> is:

```
#define NTREC     10
#define MLEN      16
#define MSIZ      4096

#define TS_TAPE   1
#define TS_INODE  2
#define TS_BITS   3
#define TS_ADDR   4
#define TS_END    5
#define TS_CLRI   6
#define MAGIC     (int) 60011
#define CHECKSUM  (int) 84446

struct  spcl {
   int        c_type;
   time_t     c_date;
   time_t     c_ddate;
   int        c_volume;
```

```
    daddr_t     c_tapea;
    ino_t       c_inumber;
    int         c_magic;
    int         c_checksum;
    struct      dinode      c_dinode;
    int         c_count;
    char        c_addr[BSIZE];
} spcl;
```

NTREC is the number of 1024 byte records in a physical tape block. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS_ entries are used in the *c_type* field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TAPE	Tape volume label
TS_INODE	A file or directory follows. The *c_dinode* field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS	A bit map follows. This bit map has a one bit for each inode that was dumped.
TS_ADDR	A subrecord of a file description. See *c_addr* below.
TS_END	End of tape record.
TS_CLRI	A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped.
MAGIC	All header records have this number in *c_magic*.
CHECKSUM	Header records checksum to this value.

The fields of the header structure are as follows:

c_type — The type of the header.

c_date — The date the dump was taken.

c_ddate — The date the file system was dumped from.

c_volume — The current volume number of the dump.

c_tapea — The current number of this (1024-byte) record.

c_inumber — The number of the inode being dumped if this is of type TS_INODE.

c_magic — This contains the value MAGIC above, truncated as needed.

c_checksum — This contains whatever value is needed to make the record sum to CHECKSUM.

c_dinode — This is a copy of the inode as it appears on the file system; see *fs*(4).

c_count — The count of characters in *c_addr*.

c_addr — An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is non-zero. If the block was not present on the file system, no block was dumped; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, TS_ADDR records will be scattered through the file, each one picking up where the last left off.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a TS_END record and then the tapemark.

FILES

/etc/dumpdates

SEE ALSO

dump(1M), restore(1M), fs(4), types(5)

NAME

filehdr - file header for common object files

SYNOPSIS

```
#include <filehdr.h>
```

DESCRIPTION

Every common 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 ptr to symtab */
     long        f_nsyms ;      /* # symtab entries */
     unsigned short  f_opthdr ; /* sizeof(opt hdr) */
     unsigned short  f_flags ;   /* flags */
} ;
```

F_symptr is the byte offset into the file at which the symbol table can be found. Its value can be used as the offset in *fseek*(3S) to position an I/O stream to the symbol table. See *aouthdr*(4) for the structure of the optional aout header. The valid magic number is:

```
#define MC68MAGIC 0520  /* X/OS magic number */
```

The value in *f_timdat* is obtained from the *time*(2) system call. Flag bits currently defined are:

```
#define F_RELFLG  00001  /* relocation entries stripped */
#define F_EXEC    00002  /* file is executable */
#define F_LNNO    00004  /* line numbers stripped */
#define F_LSYMS   00010  /* local symbols stripped */
#define F_MINMAL  00020  /* minimal object file */
#define F_UPDATE  00040  /* update file, ogen produced */
```

```
#define F_SWABD   00100  /* file is "pre-swabbed" */
#define F_AR16WR  00200  /* 16-bit DEC host */
#define F_AR32WR  00400  /* 32-bit DEC host */
#define F_AR32W   01000  /* non-DEC host */
#define F_PATCH   02000  /* "patch" list in opt hdr */
```

SEE ALSO

time(2), fseek(3S), a.out(4), aouthdr(4).

NAME

fs, inode - format of file system volume

SYNOPSIS

```
#include <sys/types.h>
#include <ufs/fs.h>
#include <ufs/inode.h>
```

DESCRIPTION

Every 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 0 to 15 are reserved.

The layout of the super block as defined by the include file *<ufs/fs.h>* is:

```
#define  FS_MAGIC 0x011954
struct  fs {
   struct  fs *fs_link;      /* linked list of file systems */
   struct  fs *fs_rlink;     /* used for in-core 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_id[2]           /* file system id */
    long fs_sparecon[4];    /* 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[MAXMNTLEN];/* name mounted on */
/* these fields retain the current block allocation info */
   long fs_cgrotor;         /* last cg searched */
   struct  csum *fs_csp[MAXCSBUFS];
                            /* list of fs_cs info buffers */
   long fs_cpc;             /* cyl per cycle in postbl */
   short   fs_postbl[MAXCPG][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. The main superblock on disk is updated at each "sync". But no critical data is changed. Therefore the copies are not updated. They are only referenced and updated after a disaster (when using the **b** option with the command *fsck*).

Addresses stored in inodes are capable of addressing fragments of "blocks". File system blocks of at most size MAXBSIZE can be optionally broken into 2, 4, or 8 pieces, each of which is addressable; these pieces may be DEV_BSIZE, or some multiple of a 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 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 the availability of space 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, so we are stuck with it). 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 which 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.

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 NBPI bytes of disk space. The inode allocation strategy is extremely conservative.

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.: MAXIPG must be a multiple of INOPB(fs).

MINBSIZE is the smallest allowable block size. With a MINBSIZE of 4096 it is possible to create files of size "2 to the power 32" with only two levels of indirection. MINBSIZE must be big enough to hold a cylinder group block, thus changes to (struct cg) must keep its size within MINBSIZE. 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 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*. 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 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: MAXBPC bounds the size of the rotational layout tables and is limited by the fact that the super block is of size 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).

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 X/OS file system.

SEE ALSO

inode(4) <*ufs/inode.h*>

NAME

fspec - format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the X/OS operating system with non-standard tabs (that is, tabs that 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 operating 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 key letter, possibly followed immediately by a value. The following parameters are recognised:

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 hyphen followed immediately by an integer *n*, indicating tabs at intervals of *n* columns

3. a hyphen 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 that are recognised 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), newform(1), tabs(1).

NAME

fstab, mntent - 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 / 4.2 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;    /* type */
   char *mnt_opts;    /* options */
   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* field is the full path name of the directory on which the filesystem is to be mounted.

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:

4.2 *mnt_fsname* must be a block special device.

nfs *mnt_fsname* is the path on the server of the directory to be served (only valid if NFS is installed).

swap *mnt_fsname* must be a block special device swap partition.

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.

mnt_opts specific to 4.2 file systems (the default is **noquota**):

quota usage limits enforced.

noquota usage limits not enforced.

mnt_opts are specific to nfs (NFS) file systems and are only valid if NFS is installed.

bg if the first attempt fails, retry in the background.

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 defaults are:

fg,retry=1,timeo=7,retrans=4,port=NFS_PORT,hard

with defaults for **rsize** and **wsize** set by the kernel.

The **bg** option causes *mount* to run in the background if the server's *mountd*(1M) does not respond. *mount* attempts each request *retry* times before giving up. Once the filesystem is mounted, each nfs request made in the kernel waits *timeo* tenths of a second for a response. If no response arrives, the timeout is multiplied by 2 and the request is retransmitted. When *retrans* 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.

The field *mnt_freq* indicates how often each partition should be dumped by the *dump*(1M) command (and triggers that command's *w* option, which determines what filesystems should be dumped). Most systems set the *mnt_freq* field to 1, indicating that filesystems are dumped each day.

The final field, *mnt_passno*, is used by the consistency checking program *fsck*(1M) to allow overlapped checking of filesystems during

a reboot. All filesystems with *mnt_passno* of 1 are checked first simultaneously, then all filesystems with *mnt_passno* of 2, and so on. It is usual to make the *mnt_passno* of the root filesystem have the value 1, and then check one filesystem on each available disk drive in each subsequent pass, until all filesystem partitions are checked.

The */etc/fstab* file is read only by programs and never written; the system administrator must maintain it manually. The order of records in */etc/fstab* is important because *fsck*, *mount*, and *umount* process the file sequentially; filesystems must appear *after* filesystems within which they are mounted.

FILES

/etc/fstab

SEE ALSO

getmntent(3), fsck(1M), mount(1M), umount(1M)

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 that 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. Lines that begin with **#** are ignored and may be used to comment the file. The format 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 fields are:

label This is the string against which *getty*(1M) tries to match its second argument. It is often the speed at which the terminal is supposed to run, e.g., 1200, but it needn't be. If *getty*(1M) 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. The first entry is also used if *getty*(1M) can't 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.

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*(1M). *Getty*(1M) understands the symbolic names specified in **/usr/include/sys/termio.h** (see *termio*(7)). Normally

only the speed flag is required in the *initial-flags* field. *Getty*(1M) 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*(1M) executes *login*(1).

final-flags These flags take the same values as the *initial-flags* and are set just before *getty*(1M) executes *login*(1). 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 communicate in a rational fashion. The other two commonly specified *final-flags are* **TAB3** (tabs are sent to the terminal as spaces) and **HUPCL** (the line is hung up on the final close).

login-prompt This entire field is printed as the *login-prompt*. White-space characters (space, tab, and new-line) are included in this field, unlike the other fields, in which white space is ignored.

next-label This field indicates the next entry *label* in the table that *getty*(1M) should use if the user types a <*break*> or the input cannot be read. Usually, a series of speeds are linked together in a closed set. No matter where the set is entered, the correct speed can be obtained. For example, **2400** is linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

After making or modifying **/etc/gettydefs**, it is strongly recommended that the file be run through *getty*(1M) with the check option to be sure there are no errors.

FILES

/etc/gettydefs

SEE ALSO

getty(1M), termio(7), login(1), ioctl(2).

NAME

group - group file

SYNOPSIS

/etc/group

DESCRIPTION

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 override 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
/etc/yp/group

SEE ALSO

setgroups(2), initgroups(3), crypt(3), passwd(1), passwd(4)

BUGS

The *passwd*(1) command won't change group passwords.

NAME

inittab - script for the init process

DESCRIPTION

The **/etc/inittab** file supplies the script for *init*(1M) to perform as a general process dispatcher. The process that constitutes the majority of *init*'s process dispatching activities is the line process */etc/getty*, which initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

NOTE: Within this section, the term *init* always refers to the program described in *init*(1M).

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 new-line; however, a backslash (\) preceding a new-line 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 *gettys* 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 field is 1 to 4 characters used to uniquely identify an entry.

rstate This field 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, *action* will be taken on this *process for 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 the system is only in these states for as long as it takes to execute all the entries associated with the states. 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, *init* is to start the process, not wait for its termination (continue scanning the *inittab* file), and, when it dies, restart the process. If the process currently exists *init* is to do nothing and continue scanning the *inittab* file.

wait When *init* enters the *run-level* that matches the entry's *rstate*, it is to 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 When *init* enters a *run-level* that matches the entry's *rstate*, it is to start the process, not wait for its termination and, when it dies, not restart the process. If a new *run-level* is entered when the process is still running, 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, either 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 only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, wait for its termination, and, when it dies, not restart the process.

powerfail *Init* is to execute the process associated with this entry only when it receives a powerfail signal (**SIGPWR**; see *signal*(2)).

powerwait *Init* is to execute the process associated with this entry only when it receives a powerfail signal (**SIGPWR**) and is to wait until the process terminates before continuing any processing of *inittab*.

off If the process associated with this entry is currently running, *init* is to send the

warning signal (**SIGTERM**) and wait 20 seconds before forcibly terminating the process via the kill signal (**SIGKILL**). If the process is nonexistent, *init* is to 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 scanned only when *init* is 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 *init* will enter *run-level* **6**. If the **initdefault** entry is **s**, *init* will start in the *SINGLE USER* state. If *init* doesn't find an **initdefault** entry in **/etc/inittab**, 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. It is expected that this entry will be only used to initialise 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 the *process* field. Comments can be inserted with the syntax:

```
; #comment
```

FILES

/etc/inittab

SEE ALSO

getty(1M), init(1M), sh(1), who(1), exec(2), open(2), signal(2).

NAME

inode - format of an inode

SYNOPSIS

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

#include <ufs/inode.h>
```

DESCRIPTION

An *inode* has the following structure.

```
/* Inode structure as it appears on a disk block. */

#define  NDADDR  12       /* direct addresses in inode */
#define  NIADDR  3        /* indirect addresses in inode */

struct  dinode
{
  ushort  di_mode;        /* 0: mode and type of file */
  short   di_nlink;       /* 2: number of links to file */
  short   di_uid;         /* 4: owner's user id */
  short   di_gid;         /* 6: owner's group id */
  quad di_size;           /* 8: number of bytes in file */
  time_t  di_atime;       /* 16: time last accessed */
  long di_atspare;
  time_t  di_mtime;       /* 24: time last modified */
  long di_mtspare;
  time_t  di_ctime;       /* 32: last time inode changed */
  long di_ctspare;
  daddr_t  di_db[NDADDR]; /* 40: disk block addresses */
  daddr_t  ic_ib[NIADDR]; /* 88: indirect blocks */
  long di_flags;          /* 100: status, currently unused */
  long di_blocks;         /* 104: blocks actually held */
  long di_gen;            /* 108: generation number */
  union {
          long  di_spare[4];
          short di_ff[8];       /* Used when type is FIFO */
```

```
    } i_spfifo;
};

#define  di_rdev  di_db[0]
```

For the meaning of the defined types *ushort*, *quad*, *daddr_t* and *time_t*, see *types*(5).

There is a unique inode allocated in memory 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.

FILES

/usr/include/ufs/inode.h

SEE ALSO

stat(2), fs(4), types(5).

NAME

ldfcn - common object file access routines

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.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>**. The primary purpose of this structure is to provide uniform access to both simple object files and object files that are members of an archive file.

The function *ldopen*(3X) allocates and initialises the **LDFILE** structure and returns a pointer to the structure to the calling program. The fields of the **LDFILE** structure may be accessed individually through macros defined in **<ldfcn.h>** and 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; the offset is non-zero if the object file is a member of an archive file.

HEADER(ldptr) The file header structure of the object file.

The object file access functions may be divided into four 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 read 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

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. The function *ldtbindex*(3X) which returns the index of a particular common object file symbol table entry.

These functions are described in detail in the manual pages identified for each function. All the functions except *ldopen*, *ldaopen*, and *ldtbindex* return either **SUCCESS** or **FAILURE**, which are constants defined in **<ldfcn.h>**. *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, translating 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 string table in a X/OS 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 the *C Language Reference Manual*).

The program must be loaded with the object file access routine library **libld.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 may not be the same as the end of one of its object file members.

SEE ALSO

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).

Common Object File Format, by I. S. Law.

NAME

linenum - line number entries in a common object file

SYNOPSIS

```
#include <linenum.h>
```

DESCRIPTION

The C compiler 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 *sdb*(1)). The structure of these line number entries appears below.

```
struct lineno
{
    union
    {
       long   l_symndx ;
       long   l_paddr ;
    }         l_addr ;
    unsigned short l_lnno ;
} ;
```

Numbering starts with one for each function. The initial line number entry for a function has *l_lnno* equal to zero, and the symbol table index of the function's entry is in *l_symndx*. Otherwise, *l_lnno* is non-zero, and *l_paddr* is the physical address of the code for the referenced line. Thus the overall structure is the following:

```
l_addr          l_lnno

function symtab index  0
physical address     line
physical address     line
...
```

```
        function symtab index  0
        physical address     line
        physical address     line
        ...
```

SEE ALSO

cc(1), sdb(1), a.out(4).

.IAME

/etc/mtab - mounted file system table

SYNOPSIS

#include <mntent.h>

DESCRIPTION

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 / 4.2 rw,noquota 1 2

The file is accessed by programs using *getmntent*(3), and by the system administrator using a text editor.

FILES

/etc/mtab

SEE ALSO

getmntent(3), fstab(4), mount(1M)

NAME

passwd - password file

SYNOPSIS

/etc/passwd

DESCRIPTION

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 (s)he logs 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 lines beginning with a plus (+), which means "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 non-null password, directory, gecos, or shell fields, they will override 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:God:/:/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* and *tut*, in case the yellow pages are out of order. The user 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)

NAME

profile - setting up an environment at login time

DESCRIPTION

If a user's login directory contains a file named **.profile**, that file will be executed (via the shell's **exec .profile**) before the user's session begins; **.profile** is handy for setting exported environment variables and terminal modes. If the file **/etc/profile** exists, it will be executed for every user before the **.profile**. The following example is typical:

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 22
# Tell me when new mail comes in
MAIL=/usr/mail/myname
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
# Set terminal type
echo "terminal: \c"
read TERM
case $TERM in
   300)       stty cr2 nl0 tabs; tabs;;
   300s)      stty cr2 nl0 tabs; tabs;;
   450)       stty cr2 nl0 tabs; tabs;;
   hp)        stty cr0 nl0 tabs; tabs;;
   745|735)   stty cr1 nl1 -tabs; TERM=745;;
   43)        stty cr1 nl0 -tabs;;
   4014|tek)  stty cr0 nl0 -tabs ff1; TERM=4014; echo "\33;";;
   *)         echo "$TERM unknown";;
esac
```

FILES

$HOME/.profile
/etc/profile

SEE ALSO

env(1), login(1), mail(1), sh(1), stty(1), su(1), environ(5), term(5).

NAME

reloc - relocation information for a common 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 */
    unsigned short  r_type ;   /* relocation type */
} ;


/*
* All generics
*    reloc. already performed to symbol in the same section
*/
#define R_ABS          0

/*
* LSX Processors
*
*/
#define R_RELBYTE 017
#define R_RELWORD 020
#define R_RELLONG 021
#define R_PCRBYTE 022
#define R_PCRWORD 023
#define R_PCRLONG 024
```

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

R_ABS The reference is absolute, and no relocation is necessary. The entry will be ignored.

R_RELBYTE A direct 8-bit reference to a symbol's virtual address.

R_RELWORD A direct 16-bit reference to a symbol's virtual address.

R_RELLONG A direct 32-bit reference to a symbol's virtual address.

R_PCRBYTE A "PC-relative" 8-bit reference to a symbol's virtual address.

R_PCRWORD A "PC-relative" 16-bit reference to a symbol's virtual address.

R_PCRLONG A "PC-relative" 32-bit reference to a symbol's virtual address.

Other relocation types will be defined as they are needed.

Relocation entries are generated automatically by the assembler and automatically utilized by the link editor. A link editor option exists for removing the relocation entries from an object file.

SEE ALSO

ld(1), strip(1), a.out(4), syms(4).

NAME

scnhdr - section header for a common object file

SYNOPSIS

```
#include <scnhdr.h>
```

DESCRIPTION

Every common 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 */
    long        s_vaddr;        /* virtual address */
    long        s_size;         /* section size */
    long        s_scnptr;       /* file ptr to raw data */
    long        s_relptr;       /* file ptr to relocation */
    long        s_lnnoptr;      /* file ptr to line numbers */
    unsigned short  s_nreloc;   /* # reloc entries */
    unsigned short  s_nlnno;    /* # line number 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*(3S). If a section is initialised, the file contains the actual bytes. An uninitialised 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, line numbers, or data. Consequently, an uninitialised section has no raw data in the object file, and the values for *s_scnptr*, *s_relptr*, *s_lnnoptr*, *s_nreloc*, and *s_nlnno* are zero.

SEE ALSO

ld(1), fseek(3S), a.out(4).

NAME

shlib - format of the shared library description file

DESCRIPTION

The "/etc/shlib" file is used to specify the binding between each numeric library code identifier available and the path name of the file that contains the corresponding shared library module within the machine.

The file is a regular text file. Its format is very straightforward. It is made of non-empty lines, separated by a new-line character. Each line in the file contains the ordered pair:

numeric-code pathname

The numeric-code field is a decimal string that specifies the unique identifier for the library. The *pathname* is the absolute pathname of the shared library module within the file system. The two fields are separated by one or more blanks or tabs.

Note that the numeric code specified for the library must be consistent with the one specified in the library description file when the library is generated (see *LSX X/OS Programmer Guide*).

FILES

/etc/shlib

SEE ALSO

LSX X/OS Programmer Guide.

NAME

syms - common object file symbol table format

SYNOPSIS

#include <syms.h>

DESCRIPTION

Common object files contain information to support *symbolic* software testing (see *sdb*(1)). Line number entries, *linenum*(4), and extensive symbolic information permit testing at the C *source* level. Every object file's symbol table is organized as shown below.

```
Filename 1.
  Function 1.
     Local symbols for function 1.
  Function 2
     Local symbols for function 2.
  ...
  Static externs for file 1.

Filename 2.
  Function 1.
     Local symbols for function 1.
  Function 2.
     Local symbols for function 2.
  ...
  Static externs for file 2.
...

Defined global symbols.
Undefined global symbols.
```

The entry for a symbol is a fixed-length structure. The members of the structure hold the name (null padded), its value, and other information. The C structure is given below.

```
#define   SYMNMLEN     8
#define   FILNMLEN     14

struct   syment
{
 union                              /* ways to get a symbol name*/
 {
      char       _n_name[SYMNMLEN] ; /* names less than 8 chars. */
      struct                        /* names 8 char or more*/
      {
        long     _n_zeroes;        /* == 0L when in string table*/
        long     _n_offset;        /* location of name in table */
      }         _n_n;
      char       *_n_nptr[2];      /* allows overlaying */
 } _n;
      long       n_value ;         /* value of symbol */
      short      n_scnum ;         /* section number */
      unsigned short  n_type ;     /* type and derived type */
      char       n_sclass ;        /* storage class */
      char       n_numaux ;        /* number of aux entries */
} ;
#define   n_name       _n._n_name
#define   n_zeroes     _n._n_n._n_zeroes
#define   n_offset     _n.n_n._n_offset
#define   n_nptr       _n._n_nptr[1]
```

Meaningful values and explanations for them are given in both **syms.h** and *Common Object File Format*. Anyone who needs to interpret the entries should seek more information in these sources. Some symbols require more information than a single entry; they are followed by *auxiliary entries* that are the same size as a symbol entry. The format follows.

```
union auxent
{
   struct
   {
       long      x_tagndx;
       union
       {
          struct
```

```
        {
             unsigned short  x_lnno;
             unsigned short  x_size;
        } x_lnsz;
        long   x_fsize;
    } x_misc;
    union
    {
        struct
        {
             long  x_lnnoptr;
             long  x_endndx;
        }    x_fcn;
        struct
        {
             unsigned short x_dimen[DIMNUM];
        }    x_ary;
    }        x_fcnary;
    unsigned short x_tvndx;
}   x_sym;
struct
{
    char  x_fname[FILNMLEN];
}   x_file;
 struct
 {
     long  x_scnlen;
     unsigned short x_nreloc;
     unsigned short x_nlinno;
 }    x_scn;

struct
{
    long x_tvfill;
    unsigned short x_tvlen;
    unsigned short x_tvran[2];
}   x_tv;
};
```

Indexes of symbol table entries begin at *zero*.

SEE ALSO

sdb(1), a.out(4), linenum(4).
Common Object File Format by I. S. Law.

WARNING

In the LSX machines, and others in which *long*s are equivalent to *int*s (such as the VAX), the *long*s are converted to *int*s in the compiler to minimise the complexity of the compiler code generator. Thus it is not possible to determine from the symbol table which symbols were declared as *long*s and which as *int*s.

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 1 block, maximum 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. *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 a decimal 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 ASCII "0" if the file is "normal" or a special file, ASCII "1" if it is a hard link, and ASCII "2" 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 inode 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 terminfo file

SYNOPSIS

term
/usr/lib/terminfo/*/*

DESCRIPTION

Compiled *terminfo* descriptions are placed under the directory **/usr/lib/terminfo**. To avoid a linear search of a huge operating 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, *act4* can be found in the file **/usr/lib/terminfo/a/act4**. 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 or more bit byte is assumed, but no assumptions about byte ordering or sign extension are made.

The compiled file is created with the *tim*(1M) program, 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 description, listing the various names for the terminal, separated by the vertical bar (|) character. 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. All the capabilities are in the file **<term.h>**. They are stored in the following order: boolean flags, number flags and string flags (see also *terminfo*(4)).

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 flags section. Each capability takes up two bytes and is stored as a short integer. If the value represented is -1, the capability is taken to be missing.

The strings section is also similar to the flags section. Each capability is stored as a short integer, in the format above. A value of -1 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.

FILES

/usr/lib/terminfo/*/* compiled terminal capability data base

SEE ALSO

curses(3X), terminfo(4), tic(1M).

NAME

terminfo - terminal capability data base

SYNOPSIS

/usr/lib/terminfo/*/*

DESCRIPTION

Terminfo is a data base describing terminals used, for example, by *vi*(1) and *curses*(3X). Terminals are described in *terminfo* by giving a set of capabilities which they have and by describing how operations are performed. Padding requirements and initialisation sequences are included in *terminfo*.

Entries in *terminfo* consist of a number of comma-separated fields. White space after each comma is ignored. The first entry for each terminal gives the names which are known for the terminal, separated by vertical bar (|) characters. The first name given is the most common abbreviation for the terminal. The last name given should be a long name fully identifying the terminal; all others are understood as synonyms for the terminal name. All names but the last should be lowercase characters and contain no blanks; the last name may well contain uppercase characters and 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, thus "hp2621". This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. Thus, a vt100 in 132 column mode would be vt100-w. The following suffixes should be used where possible:

Suffix	Meaning	Example
-w	Wide mode (more than 80 columns)	vt100-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
-*n*p	Number of pages of memory	c100-4p
-rv	Reverse video	c100-rv

CAPABILITIES

The variable is the name by which the programmer (at the *terminfo* level) accesses the capability. The "capability name" is the short name used in the text of the database, and is used by a person updating the database. The "i.code" is the two-letter internal code used in the compiled database, and always 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 and to allow the tabs in the source file **caps** to line up nicely. 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. In the list of capabilities below, the following symbols apply:

(P) indicates that padding may be specified

(G) indicates that the string is passed through *tparm* with params as given (*#i*).

(*) indicates that padding may be based on the number of lines affected

(*#i*) indicates the *i*th parameter.

Variable	Cap. name	I. Code	Description
Booleans:			
auto_left_margin,	bw	bw	cubl wraps from column 0 to last column
auto_right_margin,	am	am	Terminal has automatic margins
beehive_glitch,	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
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
over_strike,	os	os	Terminal overstrikes
status_line_esc_ok,	eslok	es	Escape can be used on the status line
teleray_glitch,	xt	xt	Tabs ruin, magic so char (Teleray 1061)
tilde_glitch,	hz	hz	Hazeltine; cannot print ~'s
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
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 of blank chars left by smso or rmso
padding_baud_rate,	pb	pb	Lowest baud where cr/nl padding is needed
virtual_terminal,	vt	vt	Virtual terminal number (X/OS system)
width_status_line,	wsl	ws	No. columns in status line

Strings:

back_tab,	cbt	bt	Back tab (P)
bell,	bel	bl	Audible signal (bell) (P)
carriage_return,	cr	cr	Carriage return (P*)
change_scroll_region,	csr	cs	change to lines #1 through #2 (vt100) (PG)
clear_all_tabs,	tbc	ct	Clear all tab stops (P)
clear_screen,	clear	cl	Clear screen and home cursor (P*)
clr_eol,	el	ce	Clear to end of line (P)
clr_eos,	ed	cd	Clear to end of display (P*)
column_address,	hpa	ch	Set cursor column (PG)
command_character,	cmdch	CC	Term. settable cmd char in prototype
cursor_address,	cup	cm	Screen rel. cursor motion row #1 col #2 (PG)
cursor_down,	cudl	do	Down one line
cursor_home,	home	ho	Home cursor (if no cup)
cursor_invisible,	civis	vi	Make cursor invisible
cursor_left,	cubl	le	Move cursor left one space

cursor_mem_address,	mrcup	CM	Memory relative cursor addressing
cursor_normal,	cnorm	ve	Make cursor appear normal (undo vs/vi)
cursor_right,	cufl	nd	Non-destructive space (cursor right)
cursor_to_ll,	ll	ll	Last line, first column (if no cup)
cursor_up,	cuul	up	Upline (cursor up)
cursor_visible,	cvvis	vs	Make cursor very visible
delete_character,	dchl	dc	Delete character (P*)
delete_line,	dll	dl	Delete line (P*)
dis_status_line,	dsl	ds	Disable status line
down_half_line,	hd	hd	Half-line down (forward 1/2 linefeed)
enter_alt_charset_mode,	smacs	as	Start alternate character set (P)
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 stand out mode
enter_underline_mode,	smul	us	Start underscore mode
erase_chars	ech	ec	Erase #1 characters (PG)
exit_alt_charset_mode,	rmacs	ae	End alternate character set (P)
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 stand out mode
exit_underline_mode,	rmul	ue	End underscore mode

flash_screen,	flash	vb	Visible bell (may not move cursor)
form_feed,	ff	ff	Hardcopy terminal page eject (P*)
from_status_line,	fsl	fs	Return from status line
init_1string,	is1	i1	Terminal initialisation string
init_2string,	is2	i2	Terminal initialisation string
init_3string,	is3	i3	Terminal initialisation string
init_file,	if	if	Name of file containing is
init_prog,	iprog	iP	Path name of program for init
insert_character,	ichl	ic	Insert character (P)
insert_line,	ill	al	Add new blank line (P*)
insert_padding,	ip	ip	Insert pad after character inserted (p*)
key_a1,	ka1	K1	Upper left of keypad
key_a3,	ka3	K3	Upper right of keypad
key_b2,	kb2	K2	Center of keypad
key_backspace,	kbs	kb	Sent by backspace key
key_c1,	kc1	K4	Lower left of keypad
key_c3,	kc3	K5	Lower right of keypad
key_catab,	ktbc	ka	Sent by clear-all-tabs key
key_clear,	kclr	kC	Sent by clear screen or erase key
key_ctab,	kctab	kt	Sent by clear-tab key
key_dc,	kdchl	kD	Sent by delete character key
key_dl,	kdll	kL	Sent by delete line key
key_down,	kcudl	kd	Sent by terminal down arrow key
key_eic,	krmir	kM	Sent by rmir or smir in insert mode
key_eol,	kel	kE	Sent by clear-to-end-of-line key
key_eos,	ked	kS	Sent by clear-to-end-of-screen key
key_f0,	kf0	k0	Sent by function key f0
key_f1,	kf1	k1	Sent by function key f1
key_f10,	kf10	k;	Sent by function key f10

key_f2,	kf2	k2	Sent by function key f2
key_f3,	kf3	k3	Sent by function key f3
key_f4,	kf4	k4	Sent by function key f4
key_f5,	kf5	k5	Sent by function key f5
key_f6,	kf6	k6	Sent by function key f6
key_f7,	kf7	k7	Sent by function key f7
key_f8,	kf8	k8	Sent by function key f8
key_f9,	kf9	k9	Sent by function key f9
key_home,	khome	kh	Sent by home key
key_ic,	kichl	kI	Sent by ins char/enter ins mode key
key_il,	kill	kA	Sent by insert line
key_left,	kcubl	kl	Sent by terminal left arrow key
key_ll,	kll	kH	Sent by home-down key
key_npage,	knp	kN	Sent by next-page key
key_ppage,	kpp	kP	Sent by previous-page key
key_right,	kcufl	kr	Sent by terminal right arrow key
key_sf,	kind	kF	Sent by scroll-forward/down key
key_sr,	kri	kR	Sent by scroll-backward/up key
key_stab,	khts	kT	Sent by set-tab key
key_up,	kcuul	ku	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	10	Labels on function key f0 if not f0
lab_f1,	lf1	11	Labels on function key f1 if not f1
lab_f10,	lf10	la	Labels on function key f10 if not f10
lab_f2,	lf2	12	Labels on function key f2 if not f2
lab_f3,	lf3	13	Labels on function key f3 if not f3
lab_f4,	lf4	14	Labels on function key f4 if not f4
lab_f5,	lf5	15	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
meta_on,	smm	mm	Turn on "meta mode" (8th bit)
meta_off,	rmm	mo	Turn off "meta mode"
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 (PG*)
parm_delete_line,	dl	DL	Delete #1 lines (PG*)
parm_down_cursor,	cud	DO	Move cursor down #1 lines (PG*)
parm_ich,	ich	IC	Insert #1 blank chars (PG*)
parm_index,	indn	SF	Scroll forward #1 lines (PG)
parm_insert_line,	il	AL	Add #1 new blank lines (PG*)
parm_left_cursor,	cub	LE	Move cursor left #1 spaces (PG)
parm_right_cursor,	cuf	RI	Move cursor right #1 spaces (PG*)
parm_rindex,	rin	SR	Scroll backward #1 lines (PG)
parm_up_cursor,	cuu	UP	Move cursor up #1 lines (PG*)
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
print_screen,	mc0	ps	Print contents of the screen
prtr_non,	mc5p	p0	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. (PG*)
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 (set row) (PG)
save_cursor,	sc	sc	Save cursor position (P)
scroll_forward,	ind	sf	Scroll text up (P)
scroll_reverse,	ri	sr	Scroll text down (P)
set_attributes,	sgr	sa	Define the video attributes (PG9)
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
tab,	ht	ta	Tab to next 8 space hardware tab stop
to_status_line,	tsl	ts	Go to status line, column #1
underline_char,	uc	uc	Underscore one char and move past it
up_half_line,	hu	hu	Half-line up (reverse 1/2 linefeed)

A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the *terminfo* file as of this writing.

```
concept100|c100|concept|c104|c100-4p|concept 100,
 am, bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>, cnorm=\Ew,
 cols#80, cr=^M$<9>, cubl=^H, cudl=^J, cufl=\E=,
 cup=\Ea%p1%' '%+%c%p2%' '%+%c,
 cuul=\E;, cvvis=\EW, db, dchl=\E^A$<16*>, dim=\EE, dll=\E^B$<3*>,
 ed=\E^C$<16*>, el=\E^U$<16>, eo, flash=\Ek$<20>\EK, ht=\t$<8>,
 ill=\E^R$<3*>, in, ind=^J, .ind=^J$<9>, ip=$<16*>,
 is2=\EU\Ef\E7\E5\E8\E1\ENH\EK\E\200\Eo&\200\Eo\47\E,
 kbs=^h, kcubl=\E>, kcudl=\E<, kcufl=\E=, kcuul=\E;,
```

```
kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
lines#24, mir, pb#9600, prot=\EI, rep=\Er%p1%c%p2%' '%+%c$<.2*>,
rev=\ED, rmcup=\Ev  $<6>\Ep\r\n, rmir=\E\200, rmkx=\Ex,
rmso=\Ed\Ee, rmul=\Eg, rmul=\Eg, sgr0=\EN\200,
smcup=\EU\Ev 8p\Ep\r, smir=\E^P, smkx=\EX, smso=\EE\ED,
smul=\EG, tabs, ul, vt#8, xenl,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Comments may be included on lines beginning with "#". 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 the size of particular delays, and string capabilities, which give a sequence that can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For example, 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**. The description of the Concept therefore 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.

Finally, string-valued capabilities, such as **el** (clear to end of line sequence) are given by the two-character code, an "=", and then a string ending at the next following ",". A delay in milliseconds may appear anywhere in such a capability, enclosed in $<..> brackets, as in **el**=\K$<3>, and padding characters are supplied by *tputs* to provide this delay. The delay can be either a number, e.g., "20", or a number followed by an "*", i.e., "3*". 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 **xenl** 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 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. Other escapes include **\^** for ^, **** for \, **\,** for comma, **\:** for :, and **\0** for null. (**\0** will 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 ****.

Sometimes individual capabilities must be commented out. To do this, put a dot before the capability name. For example, see the second **ind** in the example above.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. 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* 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 bugs in *vi*. To easily test a new terminal description, set the environment variable TERMINFO to a pathname of a directory containing the compiled description being worked on and programs will look there rather than in **/usr/lib/terminfo**. To get the padding for insert line right (if the terminal manufacturer did not document it) a severe test is to edit **/etc/passwd** at 9600 baud, delete 16 or so lines from the middle of the screen, then hit the "u" key several times quickly. If the terminal messes up, 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 is a VDU, 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 (e.g., bell, beep), give this as **bel**.

If there is a code to move the cursor one position to the left (such as backspace), that capability should be given as **cubl**. Similarly, codes to move to the right, up, and down should be given as **cufl**, **cuul**, and **cudl**. These local cursor motions should not alter the text they pass over; for example, "**cufl**= " would not normally be used because the space would erase the character moved over.

A very important point is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a CRT terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and 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.

Parameterised 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 **cufl** from the last column. Local motion from the left edge is defined only if **bw** is given; then a **cubl** 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 that 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 glass-tty terminals. Thus the model 33 teletype is described as

```
33|tty33|tty|model 33 teletype, bel=^G, cols#72, cr=^M, cudl=^J,
hc, ind=^J, os,
```

while the Lear Siegler ADM-3 is described as

```
adm3|3|lsi adm3, am, bel=^G, clear=^Z, cols#80, cr=^M, cubl=^H,
cudl=^J, ind=^J, lines#24,
```

Parameterised Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterised string capability, with *printf*(3S)-like escapes such as **%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. 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.

The **%** encodings have the following meanings:

```
%%      outputs "%"
%d      print pop() as in printf
%2d     print pop() like %2d
%3d     print pop() like %3d
```

```
%02d
%03d      as in printf
%c        print pop() gives %c
%s        print pop() gives %s

%p[1-9]   push ith parm
%P[a-z]   set variable [a-z] to pop()
%g[a-z]   get variable [a-z] and push it
%'c'      char constant c
%{nn}     integer constant nn

%+ %- %* %/ %m
          arithmetic (%m is mod): push(pop() op pop())
%& %| %^ bit operations: push(pop() op pop())
%= %> %< logical operations: push(pop() op pop())
%! %~     unary operations push(op pop())
%i        add 1 to first two parms (for ANSI terminals)

%? expr %t thenpart %e elsepart %;
          if-then-else, %e elsepart is optional.
          else-if's are possible (as in Algol 68) like this:
          %? c1 %t b1 %e c2 %t b2 %e c3 %t b3 %e c4 %t b4 %e %;
          ci are conditions, bi are bodies.
```

Binary operations are in postfix form with the operands in the usual order. That is, to get x-5 one would use "%gx%{5}%-".

Consider the HP2645, 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 printed as two digits. Thus its **cup** capability is cup=6\E&%p2%2dc%p1%2dY.

The Microterm 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 that use %c need to be able to backspace the cursor (**cubl**), and to move the cursor up one line on the screen (**cuul**). 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 it is safe to send **\t**. 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%' '%+%c%p2%' '%+%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.

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 hp2645) and can be used in preference to **cup**. If there are parameterised 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.

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** (letter l's); this may involve going up with **cuul** 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 HP terminals cannot be used for **home**.)

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 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 **ill**; 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 **dll**; this is done only from the first position on the line to be deleted. Versions of **ill** and **dll** that take a single parameter and insert or delete that many lines can be given as **il** and **dl**. If the terminal has a settable 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, however, is 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.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterised 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 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 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. The terminal type can be determined 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 the terminal does not distinguish between blanks and untyped positions. If the abc shifts over to the def and then the six characters move together around the end of the current line and onto the next during insert, this means the terminal is is the second type; give the capability **in**, which stands for insert null. While these are two logically separate attributes (one line vs. 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 that have an insert mode and terminals that 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 **ichl** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ichl**; terminals that send a sequence to open a screen position should give it here. (If the terminal has both, insert mode is usually preferable to **ichl**. 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 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 the terminal needs both to be placed into an "insert mode" and a special code to precede each inserted character, then both **smir/rmir** and **ichl** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ichl** *n* times.

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 the terminal allows motion while in insert mode, 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, the user can specify **dchl** 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 **dchl** 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 the terminal has one or more kinds of display attributes, these can be represented in a number of different ways. One display form can be chosen as *standout mode*, representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If there is a choice, reverse video plus half-bright is good, or reverse video alone.) 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 Microterm 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 there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking 9 parameters. Each parameter is either 0 or 1, as the corresponding attribute is on or off. The 9 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.

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 HP 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 is present, asserting that it is safe to move in standout mode. 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.

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**. 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 both 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 terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then give the capability **ul**. If overstrikes are erasable with a blank, then indicate this by giving **eo**.

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 HP 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 **kcubl**, **kcufl**, **kcuul**, **kcudl**, and **khome** respectively. If there are function keys such as f0, f1, ..., f10, the codes they send can be given as **kf0**, **kf1**, ..., **kf10**. If these 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), **kdchl** (delete character), **kdll** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kichl** (insert character or enter insert mode), **kill** (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.

Tabs and Initialisation

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 the *tset* command 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.

Other capabilities include **is1**, **is2**, and **is3**, initialisation strings for the terminal, **iprog**, the pathname of a program to be run to initialise the terminal, and **if**, the name of a file containing long initialisation strings. These strings are expected to set the terminal into modes consistent with the rest of the

terminfo description. They are normally sent to the terminal by the *tset* program each time the user logs in. They will be printed in the following order: **isl**; **is2**; setting tabs using **tbc** and **hts**; **if**; running the program **iprog**; **is3**. Most initialisation 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 **isl** and **is3**. A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rsl**, **rs2**, **rf**, and **rs3**, analogous to **is2** and **if**. These strings are output by the *reset* program, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs2** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the vtl00 into 80-column mode would normally be part of **is2**, but it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80 column mode.

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). If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in **is2** or **if**.

Delays

Certain capabilities control padding in the teletype driver. These are primarily needed by hard copy terminals, and are used by the *tset* program to set teletype modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cubl**, **ff**, and **tab** will cause the appropriate delay bits to be set in the teletype driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

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 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 that is set to a 23-line scrolling region), the capability **hs** should be given. Special strings to go to the beginning of the status line and to 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 parameter **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 that 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**.

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 character a given number of times (to save time transmitting a large number of identical characters), this can be indicated with the parameterised 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 that is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some systems: The environment is to be searched for a **CC** variable, and if found, all occurrences of the prototype character are replaced with the character in the environment variable.

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 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.

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.

If the terminal is one of those supported by the X/OS virtual terminal protocol, the terminal number can be given as **vt**.

Media copy strings that 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. It is undefined whether the text is also displayed on the terminal screen when the printer is on. 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. All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Strings to program function keys can be given as **pfkey**, **pfloc**, and **pfx**. 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; and **pfx** causes the string to be transmitted to the computer.

Specific Terminal Problems

Hazeltine terminals, which do not allow "~" characters to be displayed, should indicate **hz**.

Terminals that ignore a linefeed immediately after an **am** wrap, such as the Concept and vt100, should indicate **xenl**.

If **el** is required to get rid of standout (instead of merely writing normal text on top of it), **xhp** should be given.

Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This glitch is also taken to mean that it is not possible to position the cursor on top of a "magic cookie", that to erase standout mode it is instead necessary to use delete and insert line.

The Beehive Superbee, which is unable to correctly transmit the escape or control C characters, has **xsb**, indicating that the f1 key is used for escape and f2 for control C. (Only certain Superbees have this problem, depending on the ROM.)

Other specific terminal problems may be corrected by adding more capabilities of the form **xx**.

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 cancelled by placing **xx@** to the left of the capability definition, where xx is the capability. For example, the entry

```
2621-nl, smkx@, rmkx@, use=2621,
```

defines a 2621-nl that does not have the **smkx** or **rmkx** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/usr/lib/terminfo/?/* files containing terminal descriptions

SEE ALSO

curses(3X), printf(3S), term(5).

NAME

utmp, wtmp - utmp and wtmp entry formats

SYNOPSIS

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

DESCRIPTION

These files hold user and accounting information for commands such as *who*(1), *write*(1), and *login*(1). They have the following structure, as defined by **<utmp.h>**:

```
#define  UTMP_FILE  "/etc/utmp"  /* utmp: list of all current logins */
#define  WTMP_FILE  "/etc/wtmp"  /* wtmp: history of all logins */
                                 /*       since file was created */
#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 */
   char    ut_host[16]        /* host name if remote */
};

/* 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

/usr/include/utmp.h /etc/utmp /etc/wtmp

SEE ALSO

login(1), who(1), write(1), getut(3C).

NAME

intro - introduction to miscellaneous facilities

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

NAME

ascii - map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, to be printed as needed. It contains:

Table with octal codes

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

Table with hexadecimal codes

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

FILES

/usr/pub/ascii

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 commands such as *sh*(1), *time*(1), *nice*(1), and *nohup*(1) apply in searching for a file known by an incomplete pathname. 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).

TERM The kind of terminal for which output is to be prepared. This information is used by commands such as *mm*(1), and *vi*(1), which may exploit special capabilities of the terminal.

TZ Time zone information. The format is **xxxnzzz** where **xxx** is the 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, e.g., **MAIL, PS1, PS2, IFS.**

SEE ALSO

env(1), login(1), nice(1), nohup(1), sh(1), time(1), exec(2), getenv(3C), profile(4), term(5).

NAME

fcntl - file control options

SYNOPSIS

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

DESCRIPTION

The *fcntl*(2) function provides for control over open files.

The include file describes *requests* and *arguments* to *fcntl* and *open*(2),

```
/* Flag values accessible to open(2) and fcntl(2) */

#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 */
#define O_ORDERED       04000
        /* asynchronous, ordered write option */

/* 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_GETOWN        8       /* Get owner */
#define F_SETOWN        9       /* Set owner */
#define F_CHKFL         10
        /* Check legality of file flag changes */
#define F_TRUNC         11
        /* Truncate the file at arg size */
#define F_SYNC          12      /* Sync out buffered data */

/* file segment locking set data type -
   information passed to system by user */

struct flock {
     short   l_type;
     short   l_whence;
     long    l_start;
     long    l_len;
             /* len = 0 means until end of file */
     int     l_pid;
     };

/* file segment locking types */

#define F_RDLCK         01      /* Read lock */
#define F_WRLCK         02      /* Write lock */
#define F_UNLCK         03      /* Remove lock(s) */
```

SEE ALSO

fcntl(2), open(2).

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 structure and constants used by the *matherr*(3M) error-handling mechanisms, including the following constant used as an error-return value:

HUGE The maximum value of a single-precision floating-point number.

The following mathematical constants are some of the constants 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 The ratio of the circumference of a circle to its diameter. (There are also several fractions of its reciprocal and its square root.)

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.

FILES

/usr/include/math.h

SEE ALSO

intro(3), matherr(3M), values(5).

NAME

prof - profile within a function

SYNOPSIS

```
#define MARK
#include <prof.h>

void MARK (name)
```

DESCRIPTION

MARK will introduce a mark called *name* that will be treated the same as a function entry point. Execution of the mark will add to a counter for that mark, and program counter time spent will be accounted to the immediately preceding mark or to the function if there are no preceding marks within the active function.

Name may be any combination of up to six letters, numbers, or underscores. Each *name* in a single compilation must be unique, but may be the same as any ordinary program symbol.

For marks to be effective, the symbol MARK must be defined before the header file **<prof.h>** is included. This may be defined by a preprocessor directive as in the synopsis, or by a command line argument, like this:

```
cc -p -DMARK foo.c
```

If MARK is not defined, the **MARK**(*name*) statements may be left in the source files containing them and will be ignored.

EXAMPLE

In this example, marks can be used to determine how much time is spent in each loop. Unless this example is compiled with *MARK* defined on the command line, the marks are ignored.

```
#include <prof.h>
```

```
foo( )
{
   int Bi, j;

   .
   .
   .
   MARK(loop1);
   for (i = 0; i < 2000; i++) {
        . . .
   }
   MARK(loop2);
   for (j = 0; j < 2000; j++) {
        . . .
   }
 }
```

SEE ALSO

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

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 step(string, expbuf)
char *string, *expbuf;
```

DESCRIPTION

The following paragraphs describe general purpose regular expression matching routines in the form of *ed*(1), defined in **/usr/include/regexp.h**. Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), and *expr*(1), which perform regular expression matching, use this source file. Therefore, only the *regexp* 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 5 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 **\(**'s.
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 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 that 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 initialisations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for **GETC()**, **PEEKC()**, and **UNGETC()**. Otherwise it can be used to declare external variables that might be used by **GETC()**, **PEEKC()**, and **UNGETC()**. See the example below of the declarations taken from *grep*(1).

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, *step* will only try to match the regular expression to the beginning of the string. 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*; *step* continues until *advance* returns a one 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); for example, expressions like **s/y*//g** do not loop forever.

The routines *ecmp* and *getrange* are trivial and are called by the routines previously mentioned.

EXAMPLES

The following is an example, taken from the code for *grep*(1), showing the use of these regular expression macros and calls:

```
#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], ' ');
...
   if(step(linebuf, expbuf))
            succeed();
```

FILES

/usr/include/regexp.h

SEE ALSO

ed(1), grep(1), sed(1).

BUGS

The variable *circf* is not handled well.

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;
     ino_t     st_ino;
     ushort    st_mode;
     short     st_nlink;
     short     st_uid;
     short     st_gid;
     dev_t     st_rdev;
     off_t     st_size;
     time_t    st_atime;
     int       st_spare1;
     time_t    st_mtime;
     int       st_spare2;
     time_t    st_ctime;
     int       st_spare3;
     long      st_blksize;
     long      st_blocks;
     long      st_spare4[2];
};
```

```
#define   S_IFMT    0170000   /* type of file */
#define   S_IFIFO   0010000   /* fifo special */
#define   S_IFCHR   0020000   /* character special */
#define   S_IFDIR   0040000   /* directory */
#define   S_IFBLK   0060000   /* block special */
#define   S_IFREG   0100000   /* regular */
#define   S_IFLNK   0120000   /* symbolic link */
#define   S_IFSOCK  0140000   /* socket */
#define   S_ISUID   0004000   /* set user id on execution */
#define   S_ISGID   0002000   /* set group id on execution */
#define   S_ISVTX   0001000   /* save swapped text even after use */
#define   S_IREAD   0000400   /* read permission, owner */
#define   S_IWRITE  0000200   /* write permission, owner */
#define   S_IEXEC   0000100   /* execute/search permission,
                                 owner */
```

FILES

/usr/include/sys/types.h
/usr/include/sys/stat.h

SEE ALSO

stat(2), types(5).

NAME

term - conventional names for terminals

DESCRIPTION

The names in this file are used by certain commands (e.g., *nroff*, *mm*(1), *man*(1), *tabs*(1)) and are maintained as part of the shell environment (see *sh*(1), *profile*(4), and *environ*(5)) in the variable **TERM.**

3220	vt220|vision2|v2|v3220|3220|Lanpar Vision II 3220
WSL1	WSL1|wsl1|Olivetti WSL1 Workstation
arpanet	arpanet|network
bussiplexer	bussiplexer
decwriter	dw2|decwriter|dw|decwriter II
dialup	dialup
dumbd	dumbd
dw	dw2|decwriter|dw|decwriter II
dw1	dw1|decwriter I
dw2	dw2|decwriter|dw|decwriter II
dw3	dw3|la120|decwriter III
dw4	dw4|decwriter IV
ethernet	ethernet|network
gigi	gigi|vk100|dec gigi graphics terminal
gt40	gt40|dec gt40
gt42	gt42|dec gt42
la120	dw3|la120|decwriter III
network	arpanet|network
patch	plugboard|patch|patchboard
patchboard	plugboard|patch|patchboard
plugboard	plugboard|patch|patchboard
qvt109	wy50|qvt109|qume qvt109 80-column| wyse 50 80-column
switch	switch|intelligent switch
unknown	unknown
v2	vt220|vision2|v2|v3220|Lanpar Vision II 3220
v2-em	vt220-em|vision2-em|Vt220 emulator (no delays)
v3220	vt220|vision2|v2|v3220|Lanpar Vision II 3220

```
vision2         vt220|vision2|v2|v3220|Lanpar Vision II 3220
vision2-em      vt220-em|vision2-em|Vt220 emulator (no delays)
vk100           gigi|vk100|dec gigi graphics terminal
vt100           vt100|vt100-nam|vt100 w/no am
vt100-am        vt100|vt100-nam|vt100 w/no am
vt100-bot-s     vt100-s-bot|vt100-bot-s|vt100 for use with sysl
vt100-nam       vt100|vt100-nam|vt100 w/no am
vt100-nam-w     vt100-w-nam|vt100-nam-w|dec vt100 132 cols
                (w/advanced video)
vt100-nav       vt100-nav|vt100 withou advanced video option
vt100-nav-w     vt100-nav-w|vt100-w-nav|dec vt100 132 cols 14
                lines (no advanced video option)
vt100-np        vt100-np
vt100-s         vt100-s|vt100-s-top|vt100-top-s|vt100 for use
                with sysline
vt100-s-bot     vt100-s-bot|vt100-bot-s|vt100 for use with sysl
vt100-s-top     vt100-s|vt100-s-top|vt100-top-s|vt100 for use
                with sysline
vt100-top-s     vt100-s|vt100-s-top|vt100-top-s|vt100 for use
                with sysline
vt100-w         vt100-w|vt100-w-am|dec vt100 132 cols
                (w/advanced video)
vt100-w-am      vt100-w|vt100-w-am|dec vt100 132 cols
                (w/advanced video)
vt100-w-nam     vt100-w-nam|vt100-nam-w|dec vt100 132 cols
                (w/advanced video)
vt100-w-nav     vt100-nav-w|vt100-w-nav|dec vt100 132 cols 14
                lines (no advanced video option)
vt100am         vt100am
vt100nam        vt100nam
vt100s          vt100s
vt100w          vt100w
vt125           vt125|vt125 graphics terminal
vt132           vt132|vt132
vt220           vt220|vision|v2|v3220|3220|Lanpar Vision II 322
vt220-em        vt220-em|vision2-em|Vt220 emulator (no delays)
vt50            vt50|dec vt50
vt50h           vt50h|dec vt50h
vt52            vt52|dev vt52
ws1530          ws1530|ws1530-am|Olivetti Work Station 1530
ws1530-am       ws1530|ws1530-am|Olivetti Work Station 1530
```

wsli	WSL1|wsl1|Olivetti WSL1 Workstation
wy100	wy100|100|wyse 100
wy50	wy50|qvt109|qume qvt109 80-column|wyse 50 80-column
wy75	wy75|wyse 75 80-column

Local changes to this list are common. Refer to **/usr/lib/terminfo.directory** for information on terminals supported by your system.

Up to 8 characters - lowercase letters, minus sign, and/or digits - make up a basic terminal name. Terminal sub-models and operational modes are distinguished by suffixes beginning with a -. Names should be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name.

Commands whose behaviour 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*.

SEE ALSO

mm(1), nroff(1), tplot(1G), sh(1), stty(1), tabs(1), profile(4), environ(5).

BUGS

Programs that should make use of this file do not adhere to the nomenclature in a consistent manner.

NAME

types - primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in system code; some data of these types are accessible to user code:

```
typedef   unsigned char  u_char;
typedef   unsigned short u_short;
typedef   unsigned int   u_int;
typedef   unsigned long  u_long;
typedef   unsigned short ushort;
typedef   struct _physadr { int r[1]; } *physadr;
typedef   struct label_t {int  val[14]; } label_t;
typedef   struct _quad { long val[2]; } quad;
typedef   long daddr_t;
typedef   char * caddr_t;
typedef   u_long ino_t;
typedef   long swblk_t;
typedef   int size_t;
typedef   int time_t;
typedef   short dev_t;
typedef   int off_t;
typedef   long key_t;
#define   FD_SETSIZE       256

typedef   long fd_mask;
#define   NFDBITS (sizeof(fd_mask)*NBBY    /* bits per mask */
#ifndef   howmany
#define   howmany(x, y) (((x)+((y)-1))/(y))
#endif
typedef   struct fd_set {
          fd_mask fds_bits[howmany(FD_SETSIZE, NFDBITS)];
} fd_set;
```

The form *daddr_t* is used for disk addresses except in an inode 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

varargs - handle variable argument list

SYNOPSIS

```
#include <varargs.h>

typedef char *va_list

#define va_dcl int va_alist;

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 initialise *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 routin to know what type of argument is expected, as it canno 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

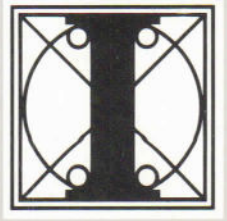
exec(2), printf(3S).

BUGS

It is up to the calling routine to somehow 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.

Printed in Italy



olivetti