Enhanced ONYX System V

PROGRAMMER REFERENCE MANUAL

Product Number: 805-02656-001

First Edition

INTRODUCTION

This manual describes the features of the UNIX system. It provides neither a general overview of the UNIX system nor details of the implementation of the system.

Not all commands, features, and facilities described in this manual are available in every UNIX system.

This manual is divided into two sections, some containing interfiled sub-classes:

2. System Calls
3. Subroutines:
   - 3C. C and Assembler Library Routines
   - 3S. Standard I/O Library Routines
   - 3M. Mathematical Library Routines
   - 3X. Miscellaneous Rouitnes
4. File Formats
5. Miscellaneous Facilities

Section 2: System Calls

This section describes the entries into the UNIX system kernel, including the C language interface.

Section 3: Subroutines

This section describes the available subroutines. Their binary versions reside in various system libraryies in the **/lib** and **/usr/lib** directories. See **intro(3)** for descriptions of these libraries and the files in which they are stored.

Section 4: File Formats

This section documents the structure of particular kinds of files; for example, the format of the outour file of the link editor is given in **a.out(4)**. Excluded are files used by only one command (for example, the assembler's intermediate files). In general, the C language **struct** declarations corresponding to these formats can be found in the directories **/usr/include** and **/usr/include/sys**.

Section 5: Miscellaneous Facilities

This section contains a variety of things. Included are descriptions of character sets, macro packages, etc.

Each section consists of a number of independent entries of a page or so in length. The name of the entry appears in the upper corners of its page. Entries within each section are alphabetized, with the exception of an introductory entry that begins each section. Some entries may describe several routines, commands, etc. In such cases, the entry appears only once, alphabetized under its "major" name. All entries are based on a common format, not all of whose parts always appear in the following manner.

The **NAME** part gives the name(s) of the entry and states briefly its purpose.

The **SYNOPSIS** summerizes the use of the program being described. A few conventions are used, particularly in Section 1 (Commands):

> **Boldface** strings are literals and are to be typed just as the appear.

Italic strings usually represent substitutable argument prototypes and program names found elsewhere in the manual. (They are underlined in the typed version of the entries.)

> Square brackets [] around an argument prototype indicate that the argument is optional. When an argument prototype is given as "name" or "file," it always refers to a file name.
>
> Ellipses ... are used to show that the previous argument prototype may be repeated.
>
> A final convention is used by the commands themselves. An argument beginning with a minus **-**, plus **+**, or equal sign **=** is often taken to be some sort of flag argument, even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with **-**, **+**, or **=**.

The **DESCRIPTION** part discusses the subject at hand.

The **EXAMPLE(S)** part gives example(s) or usage, where appropriate.

The **FILES** part gives the file names that are built into the program.

The **SEE ALSO** part gives pointers to related information.

The **DIAGNOSTICS** part discusses the diagnostic indications that may be produced. Messages that are intended to be self-explanatory are not listed.

The **WARNINGS** part points out potential pitfalls.

The **BUGS** part gives known bugs and sometimes deficiences. Occasionally, the suggested fix is also described.

A table of contents and a permuted index derived from that table precede Section 1. On each index line, the title of the entry to which that line refers is followed by the appropriate section number in parentheses. This is important because there is considerable duplication of names among the sections, arising principally from commands that exist only to exercise a particular system call.

On most systems, all entries are available on-line via the **man(1)** command (see Section 1 of the **Enhanced Onyx System V User Reference Manual**).

TABLE OF CONTENTS

3. Subroutines

PERMUTED INDEX

NAME

intro - introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all 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.

All the possible error numbers are not listed in each system call description because many errors are possible for most of the calls. 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 superuser. It is also returned for attempts by ordinary users to do things allowed only to the superuser.

2 ENOENT No such file or directory
This error occurs when a filename is specified and the file should exist but doesn't, or when one of the directories in a pathname does not exist.

3 ESRCH No such process
No process can be found corresponding to that specified by the process identifier (_pid_) in _kill_(2) or _ptrace_(2).

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

5 EIO I/O error
Some physical I/O error. 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 the I/O is beyond the limits of the

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

10 ECHILD No child processes
A *wait*(2) was executed by a process that had no existing or unwaited-for child processes.

11 EAGAIN No more processes
A *fork*(2) 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*(2), *brk*(2), or *sbrk*(2) call, a program asked 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*(2).

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

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

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

16 EBUSY Mount device 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). This error also occurs if an attempt is made to enable accounting when it is already enabled.

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

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

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

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

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

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

23 ENFILE File table overflow
The system's table of open files is full, and temporarily open(2) cannot be accepted.

24 EMFILE Too many open files
No process may have more than 20 file descriptors open at a time.

25 ENOTTY Not a typewriter

26 ETXTBSY Text file busy
An attempt was made to execute a pure-procedure program which is currently open for writing or reading. This error also indicates 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*(2) to an ordinary file, there is no free space left on the device.

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

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

31 EMLINK Too many links
An attempt was made to make more than the maximum number of links (1000) to a file.

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

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

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

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

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

37 ECHRNG Channel number out of range

38 EL2NSYNC Level 2 not synchronized

39 EL3HLT Level 3 halted

40 EL3RST Level 3 reset

41 ELNRNG Link number out of range

42 EUNATCH Protocol driver not attached

43 ENOCSI No CSI structure available

44 EL2HLT Level 2 halted

45 EDEADLOCK File locking deadlock situation

46 ENOSHP Not a binary shareable file

DEFINITIONS

Process ID

Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 0 to 30,000.

Parent Process ID

A new process is created by a currently active process; see _fork_(2). The parent process ID of a process is the process ID of its creator.

Process Group ID

Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes; see _kill_(2).

Tty Group ID

Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related processes upon termination of one of the processes in the group; see _exit_(2) and _signal_(2).

Real User ID and Real Group ID

Each user allowed on the system is identified by a positive integer called a real user ID.

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

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

Effective User ID and Effective Group ID

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

Superuser

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

Special Processes

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

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

Filename.

Names consisting of 1 to 14 characters may be used to name an ordinary file, special file, or directory.

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

Note that it is generally unwise to use *, ?, [, or] as part of filenames 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 filenames.

Pathname and Path Prefix

A pathname 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 filename.

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

```
<pathname>::=<filename>|<path-prefix><filename>|/
<path-prefix>::=<rtprefix>|/<rtprefix>
<rtprefix>::=<dirname>/|<rtprefix><dirname>/
```

where <*filename*> is a string of 1 to 14 characters other than the ASCII slash and null, and <*dirname*> is a string of 1 to 14 characters (other than the ASCII slash and null) that names a directory.

If a pathname 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 pathame is treated as if it named a non-existent file.

Directory.

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

Root Directory and Current Working Directory.

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

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

If none of these conditions exists, 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 contains 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 (see *msgop*(2)). *Msg_lrpid* is the process id of the last process that performed a *msgrcv* operation (see *msgop*(2)). *Msg_stime* is the time of the last *msgsnd* operation, *msg_rtime* is the time of the last *msgrcv* operation, and *msg_ctime* is the time of the last *msgctl*(2) operation that changed a member of the above structure.

Message Operation Permissions.

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

00400	Read by user
00200	Write by user
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 process's effective user ID is superuser.

The process's effective user ID 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 process's effective user ID does not match *msg_perm*.[*c*]*uid*, the process's effective group ID matches *msg_perm*.[*c*]*gid*, and the appropriate bit of the ``group'' portion (060) of *msg_perm*.*mode* is set.

The process's effective user ID does not match *msg_perm*.[*c*]*uid*, the process's effective group ID 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.

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;         /* r/a 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 until this semaphore's semval becomes greater than its current value. Semzcnt is a count of the number of processes that are currently suspended until this semaphore's semval becomes 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 process's effective user ID is superuser.

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

The process's effective user ID does not match *sem_perm.[c]uid*, the process's effective group ID 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 contains 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. 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 and shm_dtime is the time of the last shmdt operation; see shmop(2). 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 process's effective user ID is superuser.

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

The process's effective user ID does not match shm_perm.[c]uid,. the process's effective group ID 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.

SEE ALSO

intro(3).

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:

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

Read, write, or execute (search) permission is requested for a null pathname. [ENOENT]

The named file does not exist. [ENOENT]

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

Write access is requested for a file on a read-only file system. [EROFS]

Write access is requested for a pure procedure (shared text) file that is being executed. [ETXTBSY]

Permission bits of the file mode do not permit the requested access. [EACCES]

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

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.

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.

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

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

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

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

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

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. [EACCES]

The named file is a directory. [EISDIR]

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

Path points to an illegal address. [EFAULT]

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

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

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:

> 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).

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:

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

The named directory does not exist. [ENOENT]

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

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

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.
01000	Save text image after 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, mode bit 01000 (save text image on execution) is cleared.

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 bit 02000 (set group ID on execution) is cleared.

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

Chmod fails and the file mode remains unchanged if one or more of the following are true:

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

The named file does not exist. [ENOENT]

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

The effective user ID does not match the owner of the file and the effective user ID is not superuser. [EPERM]

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

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

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:

> A component of the path prefix is not a directory. [ENOTDIR]
>
> The named file does not exist. [ENOENT]
>
> Search permission is denied on a component of the path prefix. [EACCES]
>
> The effective user ID does not match the owner of the file and the effective user ID is not superuser. [EPERM]
>
> The named file resides on a read-only file system. [EROFS]
>
> *Path* points outside the process's allocated address space. [EFAULT]

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. The starting point for *path* searches for pathnames that begin with /.

The effective user ID of the process must be superuser to change the root directory.

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

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

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

The named directory does not exist. [ENOENT]

The effective user ID is not superuser. [EPERM]

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

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*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call. *Close* closes the file descriptor indicated by *fildes*.

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

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

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).
>
> Mode bit 01000 (save text image after execution) is cleared; see *chmod*(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). No process may have more than 20 files open simultaneously. A new file may be created with a mode that forbids writing.

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

> A component of the path prefix is not a directory. [ENOTDIR]
>
> A component of the path prefix does not exist. [ENOENT]
>
> Search permission is denied on a component of the path prefix. [EACCES]
>
> The pathname is null. [ENOENT]
>
> The file does not exist and the directory in which the file is to be created does not permit writing. [EACCES]
>
> The named file resides or would reside on a read-only file system. [EROFS]

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

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

The named file is an existing directory. [EISDIR]

Twenty (20) file descriptors are currently open. [EMFILE]

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

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 - duplicate an open file descriptor

SYNOPSIS

```
int dup (fildes)
int fildes;
```

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) 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*(2) system calls; see *fcntl*(2).

The file descriptor returned is the lowest one available.

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

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

Twenty (20) file descriptors are currently open. [EMFILE]

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

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, 0)
char *path, *arg0, *arg1, ..., *argn;

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

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

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

int execlp (file, arg0, arg1, ..., argn, 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 initialized portion and an uninitialized portion (bss). There can be no return from a successful _exec_ because the calling process is overlaid by the new process.

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

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

where _argc_ is the argument count 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 containing the name of the file.

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:

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

A component of the new process file's path prefix is not a directory. [ENOTDIR]

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. [EACCES]

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. [ENOEXEC]

The new process file is a pure procedure (shared text) file that is currently open for writing by some process. [ETXTBSY]

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

The number of bytes in the new process's argument list is greater than the system-imposed limit of 5,120 bytes. [E2BIG]

The new process file is not as long as indicated by the size values in its header. [EFAULT]

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

RETURN VALUE

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

NAME

exit, _exit - terminate process

SYNOPSIS

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

DESCRIPTION

Exit terminates the calling process with the following consequences:

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

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low-order 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 either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by *times*.

The parent process ID of all of the calling 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, 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 control over open files. _Fildes_ is an open file descriptor obtained from a _creat_(2), _open_(2), _dup_(2), _fcntl_(2), or _pipe_(2) system call.

The _cmds_ available are:

F_DUPFD Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to _arg_.

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

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

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

Same file status flags (i.e., both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across _exec_(2) system calls.

F_GETFD Get the close-on-exec flag associated with the file descriptor _fildes_. If the low-order bit is 0, the file remains open across _exec_; otherwise the file is 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).

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

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

Cmd is F_DUPFD and 20 file descriptors are currently open. [EMFILE]

Cmd is F_DUPFD and *arg* is negative or greater than 20. [EINVAL]

Refer to *fcntl*(5) for a list of the flag values contained in <fcntl.h>.

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.

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 (i.e., the process ID of the parent process).

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

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

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

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0.

Fork fails and no child process is created if one or more of the following are true:

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

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

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

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

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

SYNOPSIS

int getuid ()

int geteuid ()

int getgid ()

int 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

ioctl (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 *Administrator's Manual* discuss how *ioctl* applies to them.

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

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

Fildes is not associated with a character special device. [ENOTTY]

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

RETURN VALUE

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

SEE ALSO

termio(7) in the *Administrator's Manual*.

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 that is to be sent is specified by *sig* and is either one from the list given in *signal*(2) or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process unless the effective user ID of the sending process is superuser.

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

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

If *pid* is 0, *sig* is sent to all processes, excluding *proc0* and *proc1*, whose process group ID is equal to the process group ID of the sender.

If *pid* is -1 and the effective user ID of the sender is not superuser, *sig* is sent to all processes, excluding *proc0* and *proc1*, whose real user ID is equal to the effective user ID of the sender.

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

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

Kill fails and no signal is sent if one or more of the following are true:

> *Sig* is not a valid signal number. [EINVAL]
>
> No process can be found corresponding to that specified by *pid*. [ESRCH]
>
> 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. [EPERM]

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).

NAME
link - link to a file

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

DESCRIPTION
Path1 points to a pathname naming an existing file. *Path2* points to a pathname naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

Link fails and no link is created if one or more of the following are true:

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

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

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

The file named by *path1* does not exist. [ENOENT]

The link named by *path2* exists. [EEXIST]

The file named by *path1* is a directory and the effective user ID is not superuser. [EPERM]

The link named by *path2* and the file named by *path1* are on different logical devices (file systems). [EXDEV]

Path2 points to a null pathname. [ENOENT]

The requested link requires writing in a directory with a mode that denies write permission. [EACCES]

The requested link requires writing in a directory on a read-only file system. [EROFS]

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

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
unlink(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_.

Upon successful completion, the resulting pointer location as 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:

Fildes is not an open file descriptor. [EBADF]

Fildes is associated with a pipe or fifo. [ESPIPE]

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

The resulting file pointer would be negative. [EINVAL]

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 indicating 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

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

SYNOPSIS

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

DESCRIPTION

Mknod creates a new file named by the pathname pointed to by _path_. The mode of the new file is initialized from _mode_, where the value of _mode_ is interpreted as follows:

```
0170000 file type; one of the following:
     0010000 fifo special
     0020000 character special
     0040000 directory
     0060000 block special
     0100000 or 0000000 ordinary file
0004000 set user ID on execution
0002000 set group ID on execution
0001000 save text image after execution
0000777 access permissions; constructed from the fol-
lowing:
     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 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.

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 superuser for file types other than FIFO special.

Mknod fails and the new file is not created if one or more of the following are true:

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

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

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

The directory in which the file is to be created is located on a read-only file system. [EROFS]

The named file exists. [EEXIST]

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

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), umask(2), fs(4).

NAME

mount - mount a file system

SYNOPSIS

```
int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;
```

DESCRIPTION

Mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *Spec* and *dir* are pointers to pathnames.

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

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

Mount may be invoked only by the superuser.

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

The effective user ID is not superuser. [EPERM]

Any of the named files does not exist. [ENOENT]

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

Spec is not a block special device. [ENOTBLK]

The device associated with *spec* does not exist. [ENXIO]

Dir is not a directory. [ENOTDIR]

Spec or *dir* points outside the process's allocated address space. [EFAULT]

Dir is currently mounted on, is someone's current working directory, or is otherwise busy. [EBUSY]

The device associated with *spec* is currently mounted. [EBUSY]

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.

NAME

msgctl - message control operations

SYNOPSIS

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

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

DESCRIPTION

Msgctl provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of 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:

Msqid is not a valid message queue identifier. [EINVAL]

Cmd is not a valid command. [EINVAL]

Cmd is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)). [EACCES]

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 to the value of *msg_qbytes*, and the effective user ID of the calling process is not equal to that of superuser. [EPERM]

Buf points to an illegal address. [EFAULT]

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 initialized as follows:

Msg_perm.cuid, *msg_perm.uid*, *msg_perm.cgid*, and *msg_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of *msg_perm.mode* are set equal to the low-order 9 bits of *msgflg*.

Msg_qnum, *msg_lspid*, *msg_lrpid*, *msg_stime*, and *msg_rtime* are set equal to 0.

Msg_ctime is set equal to the current time.

Msg_qbytes is set equal to the system limit.

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

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. [EACCES]

A message queue identifier does not exist for *key* and (*msgflg* & IPC_CREAT) is ``false''. [ENOENT]

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. [ENOSPC]

A message queue identifier exists for *key* but ((*msgflg* & IPC_CREAT) **&** (*msgflg* & **IPC_EXCL**)) is ``true''. [EEXIST]

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_.{WRITE} _Msgp_ points to a structure containing the message. This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is a positive integer that can be used by the receiving process for message selection (see _msgrcv_ below). _Mtext_ is any text of length _msgsz_ bytes. _Msgsz_ can range from 0 to a system imposed maximum.

Msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to _msg_qbytes_ (see _intro_(2)).

The total number of messages on all queues system-wide is equal to the system imposed limit.

These actions are as follows:

If (_msgflg_ & **IPC_NOWAIT**) is ``true'', the message 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:

Msqid is not a valid message queue identifier. [EINVAL]

Operation permission is denied to the calling process (see _intro_(2)). [EACCES]

Mtype is less than 1. [EINVAL]

The message cannot be sent for one of the reasons cited above and (_msgflg_ & IPC_NOWAIT) is ``true''. [EAGAIN]

Msgsz is less than zero or greater than the system imposed limit. [EINVAL]

Msgp points to an illegal address. [EFAULT]

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

Msg qnum is incremented by 1.

Msg lspid is set equal to the process ID of the calling process.

Msg stime is set equal to the current time.

Msgrcv reads a message from the queue associated with the message queue identifier specified by _msqid_ and places it in the structure pointed to by _msgp_.{READ} This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is the received message's type, as specified by the sending process. *Mtext* is the text of the message. *Msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & **MSG_NOERROR**) is ``true''. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

Msgtyp specifies the type of message requested as follows:

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

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

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

Msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg* & **IPC_NOWAIT**) is ``true'', the calling process 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:

Msqid is not a valid message queue identifier. [EINVAL]

Operation permission is denied to the calling process. [EACCES]

Msgsz is less than 0. [EINVAL]

Mtext is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is ``false''. [E2BIG]

The queue does not contain a message of the desired type and (*msgtyp* & **IPC_NOWAIT**) is ``true''. [ENOMSG]

Msgp points to an illegal address. [EFAULT]

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

Upon successful completion, the return value is as follows:

Msgsnd returns a value of 0.

Msgrcv returns a value equal to the number of bytes actually placed into *mtext*.

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

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

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

onyx - Onyx 6810 special system service

SYNOPSIS

```
#include <onyx.h>
int onyx (request, arg1, arg2)
int request, arg2;
char *arg1;
```

DESCRIPTION

onyx is used to provide some special operating system services for the ONYX 6810 system. Currently, the request argument can be one of the following:

ONYX_CONF — The arg2 is ignored by this call and the system configuration information is stored into arg1.

ONYX_UCP — With this request 3K bytes of information are copied into arg1. First, 1K bytes of the user stack are copied, followed by the user area of the the process.

arg2 specifies the table entry in the **proc** table for the process desired.

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

There is not enough memory allocated for the information to be copied. [EFAULT]

Request code is invalid. [EINVAL]

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.

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 (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes. See _read_(2) and _write_(2).

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An _open_ for reading-only returns without delay. An _open_ for writing-only returns an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An _open_ for reading-only blocks until a process opens the file for writing. An _open_ for writing-only blocks until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The _open_ returns without waiting for carrier.

If O_NDELAY is clear:

The _open_ blocks until carrier is present.

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

O_CREAT If the file exists, this flag has no effect. 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 (see *creat*(2)):

> All bits set in the process's file mode creation mask are cleared. See *umask*(2).
>
> Mode bit 01000 (save text image after execution) is cleared. See *chmod*(2).

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* fails if the file exists.

Upon successful completion a non-negative integer, the file descriptor, is returned.

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).

No process may have more than 20 file descriptors open simultaneously.

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

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

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

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

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

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

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

20 file descriptors are currently open. [EMFILE]

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

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

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

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

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

RETURN VALUE

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

Refer to *fcntl*(5) for a list of the flag values contained in **<fcntl.h>**.

SEE ALSO

close(2), creat(2), dup(2), fcntl(2), lseek(2), read(2), write(2), fcntl(5).

NAME

pause - suspend process until signal

SYNOPSIS

```
pause ()
```

DESCRIPTION

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

If the signal causes termination of the calling process, *pause* 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. A value of -1 is returned from *pause* and *errno* is set to EINTR.

SEE ALSO

alarm(2), kill(2), signal(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 up to 5,120 bytes of data are buffered by the pipe before the writing process is blocked. A read on file descriptor _fildes_[0] accesses the data written to _fildes_[1] on a first-in-first-out basis.

No process may have more than 20 file descriptors open simultaneously.

Pipe fails if 19 or more file descriptors are currently open. [EMFILE]

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. *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:

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

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. [EINVAL]

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.

NAME

profil - execution time profile

SYNOPSIS

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

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (60th second); *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: 0177777 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words together. 02(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling is turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

Not defined.

SEE ALSO

prof(1), 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.

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.

The remainder of the requests 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.

1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If I and D space are separated, request 1 returns a word from I space, and request 2 returns a word from D space. If I and D space are not separated, either request 1 or request 2 may be used with equal results. The *data* argument is ignored. These two requests fail if *addr* is not the start address of a word, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

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 1024 on the PDP-11s and 0 to 2048 on the 3B20S, VAX, and M68000. 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, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated, request 4 writes a word into I space and request 5 writes a word into D space. If I and D space are not separated, either request 4 or request 5 may be used with equal results. 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.

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 few entries that can be written are:

> the general registers (i.e., registers 0-11 on the 3B20S, registers 0-7 on PDP-11s, and registers 0-15 on the VAX and M68000)
>
> certain bits of the Processor Status Word on the M68000 and M68010 (i.e., bits 0-4 and 15)
>
> the condition codes of the Processor Status Word on the 3B20S .
>
> the floating point status register and six floating point registers on PDP-11s
>
> certain bits of the Processor Status Word on PDP-11s (i.e, bits 0-4, and 8-11)
>
> certain bits of the Processor Status Longword on the VAX (i.e., bits 0-7, 16-20, and 30-31)

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 Word of the child (i.e., bit 4 on PDP-11s; bit 30 on the VAX; bit 15 on the M68000) 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. On the 3B20S there is no trace bit and this request returns an error.
Note: the trace bit remains set after an interrupt on PDP-11s but is turned off after an interrupt on the VAX and M68000.

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**.

GENERAL ERRORS

Ptrace in general fails if one or more of the following are true:

Request is an illegal number. [EIO]

Pid identifies a child that does not exist or has not executed a *ptrace* with request **0**. [ESRCH]

SEE ALSO

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

NAME

read - read from file

SYNOPSIS

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

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *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*.

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

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

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see *ioctl*(2) and *termio*(7)), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

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 fails if one or more of the following are true:

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

Buf points outside the allocated address space. [EFAULT]

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

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 *cmds* 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 *cmds* 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 *cmds* are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in *intro*(2).{READ}

IPC_SET Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */

This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of *sem_perm.uid* in the data structure associated with *semid*.

IPC_RMID Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of *sem_perm.uid* in the data structure associated with *semid*.

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

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. [EINVAL]

Operation permission is denied to the calling process (see *intro*(2)). [EACCES]

Cmd is **SETVAL** or **SETALL** and the value to which *semval* is to be set is greater than the system imposed maximum. [ERANGE]

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.uid* in the data structure associated with *semid*. [EPERM]

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

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 initialized as follows:

Sem_perm.cuid, *sem_perm.uid*, *sem_perm.cgid*, and *sem_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of *sem_perm.mode* are set equal to the low-order 9 bits of *semflg*.

Sem_nsems is set equal to the value of *nsems*.

Sem_otime is set equal to 0 and *sem_ctime* is set equal to the current time.

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

Nsems is either less than or equal to zero or greater than the system imposed limit. [EINVAL]

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. [EACCES]

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. [EINVAL]

A semaphore identifier does not exist for *key* and (*semflg* & **IPC_CREAT**) is ``false''. [ENOENT]

A semaphore identifier is to be created but the system imposed limit on the maximum number of allowed semaphores system wide would be exceeded. [ENOSPC]

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

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)):

If *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).

If *sem_op* is a positive integer, the value of *sem_op* is added to *semval* and, if (*sem_flg* & SEM_UNDO) is ``true'', the value of *sem_op* is subtracted from the calling process's semadj value for the specified semaphore. {ALTER}

If *sem_op* is zero, one of the following 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*:

Semid is not a valid semaphore identifier. [EINVAL]

Sem_num is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*. [EFBIG]

Nsops is greater than the system imposed maximum. [E2BIG]

Operation permission is denied to the calling process (see *intro*(2)). [EACCES]

The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is ``true''. [EAGAIN]

The limit on the number of individual processes requesting a **SEM_UNDO** would be exceeded. [ENOSPC]

The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit. [EINVAL]

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. [ERANGE]

Sops points to an illegal address. [EFAULT]

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

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

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_).

Setuid (_setgid_) fails if the real user (group) ID of the calling process is not equal to _uid_ (_gid_) and its effective user ID is not superuser. [EPERM]

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 superuser or to the value of shm_perm.uid in the data structure associated with shmid.

IPC_RMID Remove the shared memory identifier specified by shmid from the system and destroy the shared memory segment and data structure associated with it. This cmd can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of shm_perm.uid in the data structure associated with shmid.

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

Shmid is not a valid shared memory identifier. [EINVAL]

Cmd is not a valid command. [EINVAL]

Cmd is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)). [EACCES]

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 *shm_perm.uid* in the data structure associated with *shmid*. [EPERM]

Buf points to an illegal address. [EFAULT]

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''.

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

Shm_perm.cuid, *shm_perm.uid*, *shm_perm.cgid*, and *shm_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of *shm_perm.mode* are set equal to the low-order 9 bits of *shmflg*. *Shm_segsz* is set equal to the value of *size*.

Shm_lpid, *shm_nattch*, *shm_atime*, and *shm_dtime* are set equal to 0.

Shm_ctime is set equal to the current time.

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

Size is less than the system imposed minimum or greater than the system imposed maximum. [EINVAL]

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. [EACCES]

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. [EINVAL]

A shared memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is ``false''. [ENOENT]

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. [ENOSPC]

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. [ENOMEM]

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

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 **SHMLBA**)).
>
> 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:

> _Shmid_ is not a valid shared memory identifier. [EINVAL]
>
> Operation permission is denied to the calling process (see _intro_(2)). [EACCES]
>
> The available data space is not large enough to accommodate the shared memory segment. [ENOMEM]
>
> _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. [EINVAL]

The number of shared memory segments attached to the calling process would exceed the system imposed limit. [EMFILE]

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

Shmdt fails and does not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment. [EINVAL]

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

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. A second argument, *sig code*, is also passed to the function. *Sig code* has various contents, according to the value of *sig*. These values are provided in the table below. 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:

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

Func points to an illegal address. [EFAULT]

The table below shows how SIGTRAP handles M68000 traps. Most traps result in signals being sent to the user process that caused the trap. All other traps are considered to be STRAYFT, spurious interrupts.

The following meanings apply to information in the "SIGNAL CODE" column of the table:

code == address means the address causing the fault

code == pc means the program counter value at the time of the trap

code == (%d0) means the user parameter to the TRAP instruction

The definitions of KINTDIV, KINTOVF, and KSUBRNG are provided in the include file **<sys/signal.h>**.

TRAP TYPE	TRAP NO.	ASSIGNMENT	SIGNAL	SIGNAL CODE
BUSERR	2	bus error	SIGBUS	address
ADDRERR	3	address error	SIGILL	address
INSTERR	4	illegal instruction	SIGILL	pc
ZDVDERR	5	zero divide fault	SIGFPE	KINTDIV
CHKTRAP	6	CHK instruction fault	SIGFPE	KSUBRNG
TRAPVFT	7	TRAPV instruction fault	SIGFPE	KINTOVF
PRIVFLT	8	privileged instruction fault	SIGILL	pc
TRCTRAP	9	trace trap	SIGTRAP	pc
L1010FT	10	line 1010 emulator	SIGILL	pc
L1111FT	11	line 1111 emulator	SIGILL	pc
STRAYFT	24	spurious interrupt	n/a	n/a
SYSCALL	32	TRAP 0 - system call	n/a	(%d0)
BPTFLT	33	TRAP 1 - breakpoint	SIGTRAP	pc
IOTTRAP	34	TRAP 2 - simulate DEC IOT instruction	SIGIOT	(%d0)
EMTTRAP	35	TRAP 3 - simulate DEC EMT instruction	SIGEMT	(%d0)
FPETRAP	36	TRAP 4 - floating point exception	SIGFPE	(%d0)

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).

WARNINGS

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 UNIX 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. For the M68000, resumption cannot occur after faults requiring instruction recovery. These faults are bus errors and address errors.

NAME

stat, fstat - get file status

SYNOPSIS

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

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

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

DESCRIPTION

Path points to a pathname naming a file. 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 searchable. _Stat_ obtains information about the named file.

Similarly, _fstat_ obtains information about an open file known by the file descriptor _fildes_, obtained from a successful _open_(2), _creat_(2), _dup_(2), _fcntl_(2), or _pipe_(2) system call.

Buf is a pointer to a _stat_ structure into which information is placed concerning the file.

The contents of the structure pointed to by _buf_ include the following members:

```
ushort  st_mode;   /* File mode; see mknod(2) */
ino_t   st_ino;    /* Inode number */
dev_t   st_dev;    /* ID of device containing */
                   /* a directory entry for this file */
dev_t   st_rdev;   /* ID of device */
                   /* This entry is defined only for */
                   /* character special or block */
                   /* special files */
short   st_nlink;  /* Number of links */
ushort  st_uid;    /* User ID of the file's owner */
ushort  st_gid;    /* Group ID of the file's group */
off_t   st_size;   /* File size in bytes */
time_t  st_atime;  /* Time of last access */
time_t  st_mtime;  /* Time of last data modification */
time_t  st_ctime;  /* Time of last file status change */
                   /* Times measured in seconds since */
                   /* 00:00:00 GMT, Jan. 1, 1970 */
```

_St_atime_, _st_mtime_, and _st_ctime_ are changed by system calls as stated below.

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

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

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

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

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

The named file does not exist. [ENOENT]

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

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

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

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

Buf points to an invalid address. [EFAULT]

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), time(2), unlink(2).

NAME

utime - set file access and modification times

SYNOPSIS

```
#include <sys/types.h>
int utime (path, times)
char *path;
struct utimbuf *times;
```

DESCRIPTION

Path points to a pathname naming a file. _Utime_ sets the access and modification times of the named file.

If _times_ is **NULL**, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use _utime_ in this manner.

If _times_ is not **NULL**, _times_ is interpreted as a pointer to a _utimbuf_ structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the superuser may use _utime_ this way.

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

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

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

The named file does not exist. [ENOENT]

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

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

The effective user ID is not superuser and not the owner of the file and _times_ is not **NULL**. [EPERM]

The effective user ID is not superuser and not the owner of the file, _times_ is **NULL**, and write access is denied. [EACCES]

The file system containing the file is mounted read-only. [EROFS]

Times is not **NULL** and points outside the process's allocated address space. [EFAULT]

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

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).

NAME

intro - introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke system primitives, which are described in Section 2 of this volume. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library, _libc_, which is automatically loaded by the C compiler, _cc_(1). The link editor _ld_(1) searches this library under the **-lc** option. Some functions require declarations that can be included in the program being compiled by adding the line

#include <_header filename_>

The appropriate #include file is indicated in the SYNOPSIS part of a function description.

(3F) These functions constitute the FORTRAN intrinsic function library, _libF77_. These functions are automatically available to the FORTRAN programmer and require no special invocation of the compiler.

(3M) These functions constitute the Math Library, _libm_. They are automatically loaded as needed by the FORTRAN compiler _f77_(1). They are not automatically loaded by the C compiler, _cc_(1); however, the link editor searches this library under the **-lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**.

(3S) These functions constitute the ``standard I/O package''; an introduction to this package is provided in _stdio_(3S). The functions are in the library _libc_, already mentioned. Declarations should be obtained from the **#include** file **<stdio.h>**.

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

For descriptions and examples of #include files, refer to the "Libraries" section of the Programming Guide.

DEFINITIONS

A _character_ is any bit pattern able to fit into a byte on the machine. The _null character_ is a character with value 0, represented in the C language as '\0'. A _character array_ is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A **NULL** pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. **NULL** is defined as **0** in **<stdio.h>**; the user can include his own definition if he is not using **<stdio.h>**.

Many groups of FORTRAN intrinsic functions have *generic* function names that do not require explicit or implicit type declaration. The type of the function is determined by the type of its argument(s). For example, the generic function *max* returns an integer value if given integer arguments (*max0*), a real value if given real arguments (*amax1*), or a double-precision value if given double-precision arguments (*dmax1*).

FILES

/lib/libc.a
/usr/lib/libF77.a
/lib/libm.a

SEE ALSO

ar(1), cc(1), f77(1), ld(1), nm(1), intro(2), stdio(3S).
Programming Guide.

DIAGNOSTICS

Functions in the Math Library (3M) may return the conventional values 0 or **HUGE** (the largest single-precision floating-point number) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* (see *intro*(2)) is set to the value EDOM or ERANGE. Because many of the FORTRAN intrinsic functions use the routines found in the Math Library, the same conventions apply.

NAME

a64l, l64a - convert between long integer and base-64 ASCII string

SYNOPSIS

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

DESCRIPTION

These functions are used to maintain numbers stored in base-64 ASCII characters. This is a notation by which long integers can be represented by up to 6 characters; each character represents a ``digit'' in a radix-64 notation.

The characters used to represent ``digits'' are **.** for 0, **/** for 1, **0** through **9** for 2-11, **A** through **Z** for 12-37, and **a** through **z** for 38-63.

A64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than 6 characters, *a64l* uses the first 6.

L64a takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

BUGS

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME
abort - generate an IOT fault

SYNOPSIS
int abort ( )

DESCRIPTION
Abort causes an IOT signal to be sent to the process. This usually results in termination with a core dump.

It is possible for *abort* to return control if **SIGIOT** is caught or ignored, in which case the value returned is that of the *kill*(2) system call.

SEE ALSO
adb(1), exit(2), kill(2), signal(2).

DIAGNOSTICS
If SIGIOT is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message **abort - core dumped** is written by the shell.

NAME

abs - return integer absolute value

SYNOPSIS

```
int abs (i)
int i;
```

DESCRIPTION

Abs returns the absolute value of its integer operand.

BUGS

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

SEE ALSO

floor(3M).

NAME

assert - verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

This macro is useful for putting diagnostics into programs. If _expression_ is false (zero) when _assert_ is executed, _assert_ prints

Assertion failed: _expression_**,** _file xyz_**,** _line nnn_

on the standard error output and aborts. In the error message, _xyz_ is the name of the source file and _nnn_ is the source _line_ number of the _assert_ statement.

Compiling with the preprocessor option **-DNDEBUG** (see _cpp_(1)), or with the preprocessor control statement **#define NDEBUG** ahead of the **#include <assert.h>** statement, stops assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

NAME

atof - convert ASCII string to floating-point number

SYNOPSIS

```
double atof (nptr)
char *nptr;
```

DESCRIPTION

Atof converts a character string pointed to by *nptr* to a double-precision floating-point number. The first unrecognized character ends the conversion. *Atof* recognizes an optional string of white-space characters (blanks or tabs), then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or E followed by an optionally signed integer. If the string begins with an unrecognized character, *atof* returns the value zero.

DIAGNOSTICS

When the correct value would overflow, *atof* returns **HUGE**, and sets *errno* to **ERANGE**. Zero is returned on underflow.

SEE ALSO

scanf(3S).

NAME

j0, j1, jn, y0, y1, yn - Bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

DESCRIPTION

J0 and *j1* return Bessel functions of *x* of the first kind of orders 0 and 1 respectively. *Jn* returns the Bessel function of *x* of the first kind of order *n*.

Y0 and *y1* return the Bessel functions of *x* of the second kind of orders 0 and 1 respectively. *Yn* returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1*, and *yn* to return the value **HUGE** and to set *errno* to **EDOM**. They also cause a message indicating DOMAIN error to be printed on the standard error output; the process will continue.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

bsearch - binary search

SYNOPSIS

```
char *bsearch ((char *) key, (char *) base, nel, sizeof
(*key), compar)
unsigned nel;
int (*compar)( );
```

DESCRIPTION

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *Key* points to the datum to be sought in the table. *Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero, depending on whether the first argument is to be considered less than, equal to, or greater than the second.

DIAGNOSTICS

A **NULL** pointer is returned if the key cannot be found in the table.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

lsearch(3C), hsearch(3C), qsort(3C), tsearch(3C).

NAME

clock - report CPU time used

SYNOPSIS

long clock ( )

DESCRIPTION

Clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2) or *system*(3S).

The resolution of the clock is 16.667 milliseconds on M68000 or DEC processors.

SEE ALSO

times(2), wait(2), system(3S).

BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned wraps around after accumulating only 2,147 seconds of CPU time (about 36 minutes).

NAME

toupper, tolower, _toupper, _tolower, toascii - translate characters

SYNOPSIS

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;
```

DESCRIPTION

Toupper and *tolower* have as domain the range of *getc*(3S): the integers from -1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

_toupper and *_tolower* are macros that accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

Toascii yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype(3C), getc(3S).

NAME

crypt, setkey, encrypt - generate DES encryption

SYNOPSIS

```
char *crypt (key, salt)
char *key, *salt;

void setkey (key)
char *key;

void encrypt (block, edflag)
char *block;
int edflag;
```

DESCRIPTION

Crypt is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended to frustrate use of hardware implementations of the DES for key search.

Key is a user's typed password. *Salt* is a 2-character string chosen from the set [a-zA-Z0-9./]; this string is used to perturb the DES algorithm in one of 4,096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first 2 characters are the salt itself.

The *setkey* and *encrypt* entries provide (rather primitive) access to the actual DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. The 56-bit key is used with the above-mentioned algorithm to encrypt or decrypt the string *block* with the function *encrypt*.

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

login(1), passwd(1), getpass(3C), passwd(4).

BUGS

The return value points to static data that is overwritten by each call.

NAME

ctermid - generate filename for terminal

SYNOPSIS

```
#include <stdio.h>

char *ctermid(s)
char *s;
```

DESCRIPTION

Ctermid generates the pathname of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least L_ctermid elements; the pathname is placed in this array and the value of *s* is returned. The constant L_ctermid is defined in the <stdio.h> header file.

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (/dev/tty) that refers to the terminal if used as a filename. For this reason, *ttyname* is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

NAME

ctime, localtime, gmtime, asctime, tzset - convert date and time to string

SYNOPSIS

```
#include <time.h>

char *ctime (clock)
long *clock;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ( )
```

DESCRIPTION

Ctime converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 01:03:52 1973\n\0
```

Localtime and *gmtime* return pointers to *tm* structures, described below. *Localtime* corrects for the time zone and possible Daylight Savings Time; *gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the system uses.

Asctime converts a *tm* structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

Declarations of all the functions and externals, and the *tm* structure, are in the *<time.h>* header file. The structure declaration is:

```
struct tm {
        int tm_sec; /* seconds (0 - 59) */
        int tm_min; /* minutes (0 - 59) */
        int tm_hour; /* hours (0 - 23) */
        int tm_mday; /* day of month (1 - 31) */
```

```
        int tm_mon; /* month of year (0 - 11) */
        int tm_year; /* year - 1900 */
        int tm_wday; /* day of week (Sunday = 0) */
        int tm_yday; /* day of year (0 - 365) */
        int tm_isdst;
};
```

Tm_isdst is non-zero if Daylight Savings Time is in effect.

The external **long** variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, *timezone* is 5*60*60); the external variable *daylight* is non-zero if, and only if, the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

If an environment variable named **TZ** is present, *asctime* uses the contents of the variable to override the default time zone. The value of **TZ** must be a 3-letter time zone name, followed by a number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional 3-letter name for a daylight time zone. For example, the setting for New Jersey would be **EST5EDT**. The effects of setting **TZ** are thus to change the values of the external variables *timezone* and *daylight*; in addition, the time zone names contained in the external variable

```
char *tzname[2] = { "EST", "EDT" };
```

are set from the environment variable **TZ**. The function *tzset* sets these external variables from **TZ**; *tzset* is called by *asctime* and may also be called explicitly by the user.

Note that in most installations, **TZ** is set by default when the user logs on, to a value in the local /etc/profile file (see *profile*(4)).

SEE ALSO

time(2), getenv(3C), profile(4), environ(5).

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii - classify characters

SYNOPSIS

```
#include <ctype.h>

int isalpha (c)
int c;

. . .
```

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. _Isascii_ is defined on all integer values; the rest are defined only where _isascii_ is true and on the single non-ASCII value **EOF** (-1); see _stdio_(3S)).

isalpha	_c_ is a letter.
isupper	_c_ is an upper-case letter.
islower	_c_ is a lower-case letter.
isdigit	_c_ is a digit [0-9].
isxdigit	_c_ is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	_c_ is an alphanumeric (letter or digit).
isspace	_c_ is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	_c_ is a punctuation character (neither control nor alphanumeric).
isprint	_c_ is a printing character, code 040 (space) through 0176 (tilde).
isgraph	_c_ is a printing character, similar to _isprint_ except false for space.
iscntrl	_c_ is a delete character (0177) or an ordinary control character (less than 040).
isascii	_c_ is an ASCII character, code less than 0200.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

NAME

cuserid - get character login name of the user

SYNOPSIS

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

DESCRIPTION

Cuserid generates a character-string representation of the login name of the owner of the current process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character (\0) is placed at *s*[*0*].

SEE ALSO

getlogin(3C), getpwent(3C).

NAME

dial - establish an out-going terminal line connection

SYNOPSIS

```
#include <dial.h>

int dial (call)
CALL *call;

void undial (fd)
int fd;
```

DESCRIPTION

Dial returns a file descriptor for a terminal line open for read/write. The argument to *dial* is a CALL structure (defined in the <dial.h> header file.

When finished with the terminal line, the calling program must invoke *undial* to release the semaphore that has been set during the allocation of the terminal device.

The CALL typedef in the <dial.h> header file is:

```
typedef struct {
     struct termio    *attr;/* pointer to termio attribute struct */
     int               baud;    /* transmission data rate */
     int              speed;    /* 212A modem: low=300, high=1200 */
     char             *line;    /* device name for out-going line */
     char            *telno;    /* pointer to tel-no digits string */
     int              modem;    /* specify modem control for direct line
} CALL;
```

The CALL element *speed* is intended only for use with an out-going dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high-speed or low-speed setting on the 212A modem. The CALL element *baud* is for the desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200).

If the desired terminal line is a direct line, a string pointer to its device name should be placed in the *line* element in the CALL structure. Legal values for such terminal device names are kept in the *L-devices* file. In this case, the value of the *baud* element need not be specified as it will be determined from the *L-devices* file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of symbols described on the *acu*(7). The termination symbol will be supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure.

The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element *attr* is a pointer to a *termio* structure, as defined in the **<termio.h>** header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is important for attributes such as parity and baud rate.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..*tty-device*

SEE ALSO

uucp(1C), alarm(2), read(2), write(2).
termio(7) in the *Administrator's Manual*.

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure is returned. Mnemonics for these negative indices as listed here are defined in the **<dial.h>** header file.

```
INTRPT    -1    /* interrupt occured */
D_HUNG    -2    /* dialer hung (no return from write) */
NO_ANS    -3    /* no answer within 10 seconds */
ILL_BD    -4    /* illegal baud-rate */
A_PROB    -5    /* acu problem (open() failure) */
L_PROB    -6    /* line problem (open() failure) */
NO_Ldv    -7    /* can't open LDEVS file */
DV_NT_A   -8    /* requested device not available */
DV_NT_K   -9    /* requested device not known */
NO_BD_A   -10   /* no device available at requested baud */
NO_BD_K   -11   /* no device known at requested baud */
```

WARNINGS

Including the <dial.h> header file automatically includes the **<termio.h>** header file.

Because the above routine uses **<stdio.h>**, the size of programs not otherwise using standard I/O is increased more than might be expected.

BUGS

An *alarm*(2) system call for 3,600 seconds is made (and caught) within the *dial* module for the purpose of ``touching'' the *LCK*.. file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1C) may simply delete the *LCK*.. entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a *read*(2) or *write*(2) system call, causing an apparent error return. If the user program is to run for an hour or more, error returns from *reads* should be checked for (errno==EINTR), and the *read* possibly reissued.

delim $$

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 - generate uniformly distributed pseudo-random numbers

SYNOPSIS

```
double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];
```

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions drand48 and erand48 return non-negative double-precision floating-point values uniformly distributed over the interval $[0.0,~1.0).$

Functions lrand48 and nrand48 return non-negative long integers uniformly distributed over the interval $[0,~2 sup 31).$

Functions mrand48 and jrand48 return signed long integers uniformly distributed over the interval $[-2 sup 31 ,~2 sup 31).$

Functions srand48, seed48, and lcong48 are initialization entry points, one of which should be invoked before drand48, lrand48, or mrand48 is called. (Although it is not recommended practice, constant default initializer values are supplied automatically if drand48, lrand48, or mrand48 is called without a prior call to an initialization entry point.) Functions *erand48*, *nrand48*, and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, $X sub i ,$ according to the linear congruential formula

X sub{n+1}~=~(aX sub n^+^c) sub{roman mod~m}~~~~~~~~n>=0.

The parameter $m^=^2 sup 48$; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value a and the addend value c are given by

a~mark =~roman 5DEECE66D^sub 16~=~roman 273673163155^sub 8
c~lineup =~roman B^sub 16~=~roman 13^sub 8 .

The value returned by any of the functions *drand48*, *erand48*, *lrand48*, *nrand48*, *mrand48*, or *jrand48* is computed by first generating the next 48-bit $X sub i$ in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of $X sub i$ and transformed into the returned value.

The functions *drand48*, *lrand48*, and *mrand48* store the last 48-bit $X sub i$ generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions *erand48*, *nrand48*, and *jrand48* require the calling program to provide storage for the successive $X sub i$ values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of $X sub i$ into the array and pass it as an argument. By using different arguments, functions *erand48*, *nrand48*, and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream does *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of $X sub i$ to the 32 bits contained in its argument. The low-order 16 bits of $X sub i$ are set to the arbitrary value $roman 330E sub 16 .$

The initializer function *seed48* sets the value of $X sub i$ to the 48-bit value specified in the argument array. The previous value of $X sub i$ is copied into a 48-bit internal buffer, used only by *seed48*. A pointer to this buffer is the value returned by *seed48*. The returned pointer, which can be ignored if not needed, is useful if a program is to be restarted from a given point at some future time. Use the pointer to get and store the last $X sub i$ value; then use this value to reinitialize via seed48 when the program is restarted.

The initialization function lcong48 allows the user to specify the initial $X sub i ,$ the multiplier value $a,$ and the addend value $c.$ Argument array elements param[0-2] specify $X sub i ,$ elements param[3-5] specify the multiplier $a,$ and param[6] specifies the 16-bit addend $c.$ After lcong48 has been called, a subsequent call to either srand48 or seed48 will restore the ``standard'' multiplier and addend values, a and $c,$ specified on the previous page.

NOTES

The versions of these routines for the VAX-11 and PDP-11 are coded in assembly language for maximum speed. It requires approximately 80 Msec on a VAX-11/780 and 130 Msec on a PDP-11/70 to generate one pseudo-random number. On other computers, currently including the M68000 processors, the routines are coded in portable C. The source code for the portable version can even be used on computers which do not have floating-point arithmetic. In such a situation, functions drand48 and erand48 do not exist; instead, they are replaced by the following two functions:

```
long irand48 (m)
unsigned short m;

long krand48 (xsubi, m)
unsigned short xsubi[3], m;
```

Functions irand48 and krand48 return non-negative long integers uniformly distributed over the interval $[0,~m-1].$

SEE ALSO

rand(3C).

NAME

ecvt, fcvt, gcvt - convert floating-point number to string

SYNOPSIS

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION

Ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to this string. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero; otherwise it is zero.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by *ndigit*.

Gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in Fortran F-format, ready for printing; E-format is produced when F-format is not possible. A minus sign, if there is one, or a decimal point is included as part of the returned string. Trailing zeros are suppressed.

SEE ALSO

printf(3S).

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

end, etext, edata - last locations in program

SYNOPSIS

```
extern end;
extern etext;
extern edata;
```

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of _etext_ is the first address above the program text, _edata_ above the initialized data region, and _end_ above the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with _end_, but the program break may be reset by the routines of _brk_(2), _malloc_(3C), standard input/output (_stdio_(3S)), the profile (**-p**) option of _cc_(1), and others. Thus, the current value of the program break should be determined by **sbrk(0)** (see _brk_(2)).

SEE ALSO

brk(2), malloc(3C).

NAME

erf, erfc - error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

DESCRIPTION

Erf returns the error function of *x*, defined as {2 over sqrt pi} int from 0 to x e sup {- t sup 2} dt .

Erfc, which returns 1.0 - *erf*(*x*), is provided because of the extreme loss of relative accuracy if *erf*(*x*) is called for large *x* and the result subtracted from 1.0 (e.g. for *x* = 5, 12 places are lost).

SEE ALSO

exp(3M).

NAME

exp, log, log10, pow, sqrt - exponential, logarithm, power, square root functions

SYNOPSIS

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x ;

double pow (x, y)
double x, y;

double sqrt (x)
double x;
```

DESCRIPTION

Exp returns *e*8*x*9.

Log returns the natural logarithm of *x*. The value of *x* must be positive.

Log10 returns the logarithm base ten of *x*. The value of *x* must be positive.

Pow returns *x*8*y*9. The values of *x* and *y* may not both be zero. If *x* is non-positive, *y* must be an integer.

Sqrt returns the square root of *x*. The value of *x* may not be negative.

DIAGNOSTICS

Exp returns **HUGE** when the correct value would overflow, and sets *errno* to **ERANGE**.

Log and *log10* return 0 and set *errno* to **EDOM** when *x* is non-positive. An error message is printed on the standard error output.

Pow returns 0 and sets *errno* to **EDOM** when *x* is non-positive and *y* is not an integer, or when *x* and *y* are both zero. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow, *pow* returns **HUGE** and sets *errno* to **ERANGE**.

Sqrt returns 0 and sets *errno* to **EDOM** when *x* is negative. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

hypot(3M), matherr(3M), sinh(3M).

NAME

fclose, fflush - close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

DESCRIPTION

Fclose causes any buffered data for the named _stream_ to be written out and the _stream_ to be closed.

Fclose is performed automatically for all open files upon calling _exit_(2).

Fflush causes any buffered data for the named _stream_ to be written to that file. The _stream_ remains open.

DIAGNOSTICS

These functions return 0 for success, and **EOF** if any error (such as trying to write to a file that has not been opened for writing) was detected.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S).

NAME

ferror, feof, clearerr, fileno - stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int feof (stream)
FILE *stream;

int ferror (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

int fileno (stream)
FILE *stream;
```

DESCRIPTION

Feof returns non-zero when **EOF** has previously been detected reading the named input _stream_; otherwise, it returns zero.

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named _stream_; otherwise, it returns zero.

Clearerr resets the error indicator and **EOF** indicator to zero on the named _stream_.

Fileno returns the integer file descriptor associated with the named _stream_; see _open_(2).

NOTE

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open(2), fopen(3S).

NAME

floor, ceil, fmod, fabs - floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

DESCRIPTION

Floor returns the largest integer (as a double-precision number) not greater than *x*.

Ceil returns the smallest integer not less than *x*.

Fmod returns *x* if *y* is zero; otherwise, it returns the number *f* with the same sign as *x*, such that $x = iy + f$ for some integer *i*, and $|f| < |y|$.

Fabs returns $|x|$.

SEE ALSO

abs(3C).

NAME

fopen, freopen, fdopen - open a stream

SYNOPSIS

```
#include &lt;stdio.h&gt;

FILE *fopen (filename, type )
char *filename, *type;

FILE *freopen (filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

DESCRIPTION

Fopen opens the file named by *filename* and associates a *stream* with it. *Fopen* returns a pointer to the FILE structure associated with the *stream*.

Filename points to a character string that contains the name of the file to be opened.

Type is a character string having one of the following values:

r	open for reading
w	truncate or create for writing
a	append; open for writing at end of file, or create for writing
r+	open for update (reading and writing)
w+	truncate or create for update
a+	append; open or create for update at end-of-file

Freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *Freopen* returns a pointer to the FILE structure associated with *stream*.

Freopen is typically used to attach the preopened *streams* associated with **stdin**, **stdout**, and **stderr** to other files.

Fdopen associates a *stream* with a file descriptor by formatting a file structure from the file descriptor. Thus, *fdopen* can be used to access the file descriptors returned by *open*(2), *dup*(2), *creat*(2), or *pipe*(2). (These calls open files but do not return pointers to a FILE structure.) The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *Fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

open(2), fclose(3S).

DIAGNOSTICS

Fopen and *freopen* return a NULL pointer on failure.

NAME

fread, fwrite - binary input/output

SYNOPSIS

```
#include <stdio.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

DESCRIPTION

Fread copies *nitems* items of data from the named input *stream* into an array beginning at *ptr*. An item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *Fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream* or if *nitems* items have been read. *Fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *Fread* does not change the contents of *stream*.

Fwrite appends at most *nitems* items of data from the the array pointed to by *ptr* to the named output *stream*. *Fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *Fwrite* does not change the contents of the array pointed to by *ptr*.

The variable *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

SEE ALSO

read(2), write(2), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S).

DIAGNOSTICS

Fread and *fwrite* return the number of items read or written. If *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

NAME

frexp, ldexp, modf - manipulate parts of floating-point numbers

SYNOPSIS

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp ;

double modf (value, iptr)
double value, *iptr;
```

DESCRIPTION

Every non-zero number can be written uniquely as *x** 28*n*9, where the ``mantissa'' (fraction) *x* is in the range 0.5 ≤ |*x*| < 1.0, and the ``exponent'' *n* is an integer. *Frexp* returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*.

Ldexp returns the quantity *value** 28*exp*9.

Modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If *ldexp* would cause overflow, **HUGE** is returned and *errno* is set to ERANGE.

NAME

fseek, rewind, ftell - reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the _stream_. The new position is at the signed distance _offset_ bytes from the beginning, the current position, or the end of the file, when the value of _ptrname_ is 0, 1, or 2, respectively.

Rewind(_stream_) is equivalent to _fseek_(_stream_, 0L, 0), except that no value is returned.

Fseek and _rewind_ undo any effects of _ungetc_(3S).

After _fseek_ or _rewind_, the next operation on a file opened for update may be either input or output.

Ftell returns the offset of the current byte relative to the beginning of the file associated with the named _stream_.

SEE ALSO

lseek(2), fopen(3S).

DIAGNOSTICS

Fseek returns non-zero for improper seeks; otherwise it returns zero. An improper seek can be, for example, an _fseek_ done on a file that has not been opened via _fopen_; in particular, _fseek_ may not be used on a terminal or on a file opened via _popen_(3S).

WARNING

On an offset returned by _ftell_ is measured in bytes, and it is permissible to seek to positions relative to that offset; however, portability to systems other than requires that an offset be used by _fseek_ directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

ftw - walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;
```

DESCRIPTION

Ftw recursively descends the directory hierarchy rooted in _path_. For each object in the hierarchy, _ftw_ calls _fn_, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure (see _stat_(2)) containing information about the object, and an integer. Possible values of the integer, defined in the **<ftw.h>** header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which _stat_ could not be executed successfully. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to _fn_ is a file in a directory with read permission but not execute (search) permission.

Ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of _fn_ returns a nonzero value, or an error is detected within _ftw_ (such as an I/O error). If the tree is exhausted, _ftw_ returns zero. If _fn_ returns a nonzero value, _ftw_ stops its tree traversal and returns whatever value was returned by _fn_. If _ftw_ detects an error, it returns -1, and sets the error type in _errno_.

Ftw uses one file descriptor for each level in the tree. The _depth_ argument limits the number of file descriptors so used. If _depth_ is zero or negative, the effect is the same as if it were 1. _Depth_ must not be greater than the number of file descriptors currently available for use. _Ftw_ runs more quickly if _depth_ is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

BUGS

Because _ftw_ is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

Ftw could be made to run faster and use less storage on deep structures at the cost of considerable complexity.
Ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* does not have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

NAME

gamma - log gamma function

SYNOPSIS

```
#include <math.h>

extern int signgam;

double gamma (x)
double x;
```

DESCRIPTION

Gamma returns the natural log of gamma as a function of the absolute value of a given value. delim $$ _Gamma_ returns $ln (| GAMMA (^ x) |)$, where $GAMMA (^ x)$ is defined as

$int from 0 to inf e sup { - t } t sup { x - 1 } dt$.

The sign of GAMMA (^ x) is returned in the external integer _signgam_. The argument _x_ may not be a non-positive integer.

The following C program fragment might be used to calculate G:

```
if ((y = gamma(x)) > LOGHUGE)
        error();
y = signgam * exp(y);
```

where LOGHUGE is the least value that causes _exp_(3M) to return a range error.

DIAGNOSTICS

For non-negative integer arguments **HUGE** is returned, and _errno_ is set to **EDOM.** A message indicating DOMAIN error is printed on the standard error output.

If the correct value would overflow, _gamma_ returns **HUGE** and sets _errno_ to **ERANGE.**

These error-handling procedures may be changed with the function _matherr_(3M).

SEE ALSO

exp(3M), matherr(3M).

NAME

getc, getchar, fgetc, getw - get character or word from stream

SYNOPSIS

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next character (i.e., byte) from the named input *stream*. It also moves the file pointer, if defined, ahead one character in *stream*. *Getc* is a macro and therefore cannot be used if a function is necessary; for example, one cannot have a function pointer point to it.

Getchar returns the next character from the standard input stream, *stdin*. As in the case of *getc*, *getchar* is a macro.

Fgetc performs the same function as *getc*, but is a genuine function. *Fgetc* runs more slowly than *getc*, but takes less space per invocation.

Getw returns the next word (i.e., integer) from the named input *stream*. The size of a word varies from machine to machine. It returns the constant **EOF** upon end-of-file or error, but as that is a valid integer value, *feof* and *ferror*(3S) should be used to check the success of *getw*. *Getw* increments the associated file pointer, if defined, to point to the next word. *Getw* assumes no special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S).

DIAGNOSTICS

These functions return the integer constant **EOF** at end-of-file or upon an error.

BUGS

Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. In particular, **getc(*f++)** doesn't work sensibly. *Fgetc* should be used instead.

Because of possible differences in word length and byte ordering, files written using putw are machine-dependent, and may not be read using getw on a different processor.

NAME

getcwd - get pathname of current working directory

SYNOPSIS

```
char *getcwd (buf, size)
char *buf;
int size;
```

DESCRIPTION

Getcwd returns a pointer to the current directory pathname. The value of size must be at least two greater than the length of the pathname to be returned.

If buf is a NULL pointer, getcwd obtains size bytes of space using malloc(3C). In this case, the pointer returned by getcwd may be used as the argument in a subsequent call to free.

The function is implemented by using popen(3S) to pipe the output of the pwd(1) command into the specified string space.

EXAMPLE

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
     perror(``pwd'');
     exit(1);
}
printf(``%s\n'', cwd);
```

SEE ALSO

pwd(1), malloc(3C), popen(3S).

DIAGNOSTICS

Returns **NULL** with errno set if size is not large enough, or if an error occurs in a lower-level function.

NAME

getenv - return value for environment name

SYNOPSIS

```
char *getenv (name)
char *name;
```

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *name*=*value*, and returns a pointer to the *value* in the current environment if such a string is present; otherwise a NULL pointer is returned.

SEE ALSO

environ(5).

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent - obtain group file entry from a group file

SYNOPSIS

```
#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )
```

DESCRIPTION

Getgrent, _getgrgid_, and _getgrnam_ each return pointers to an object with the following structure containing the broken-out fields of a line in the **/etc/group** file. Each line contains a group structure, defined in the **<grp.h>** header file.

```
struct group {
        char    *gr_name;   /* the name of the group */
        char    *gr_passwd; /* the encrypted group password */
        int     gr_gid;     /* the numerical group ID */
        char    **gr_mem;   /* vector of pointers to member names */
};
```

When first called, _getgrent_ returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; therefore, successive calls may be used to search the entire file. _Getgrgid_ searches from the beginning of the file until a numerical group id matching _gid_ is found; it returns a pointer to the particular structure in which the match was found. _Getgrnam_ searches from the beginning of the file until a group name matching _name_ is found; it returns a pointer to the particular structure in which the match was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to _setgrent_ has the effect of rewinding the group file to allow repeated searches. _Endgrent_ may be called to close the group file when processing is complete.

FILES

/etc/group

SEE ALSO

getlogin(3C), getpwent(3C), group(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**. This causes them to increase the size of programs not otherwise using standard I/O more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getlogin - get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

Getlogin returns a pointer to the login name as found in /etc/utmp. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns a **NULL** pointer. The correct procedure for determining the login name is to call *cuserid* or *getlogin*. If *getlogin* fails, call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

DIAGNOSTICS

Getlogin returns the **NULL** pointer if *name* is not found.

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

getopt - get option letter from argument vector

SYNOPSIS

```
int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;
```

DESCRIPTION

Getopt returns the next option letter in *argv* that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *Optarg* is set to point to the start of the option argument on return from *getopt*.

Getopt places in *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to zero automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **EOF**. The special option -- may be used to delimit the end of the options; **EOF** will be returned, and -- will be skipped.

DIAGNOSTICS

Getopt prints an error message on *stderr* and returns a question mark (?) when it encounters an option letter not included in *optstring*.

WARNING

The above routine uses **<stdio.h>**. This causes the size of programs not otherwise using standard I/O to increase more than might be expected.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
     int c;
     extern int optind;
     extern char *optarg;
      .
```

```
     .
      .
     while ((c = getopt (argc, argv, "abf:o:")) != EOF)
          switch (c) {
          case 'a':
               if (bflg)
                    errflg++;
               else
                    aflg++;
               break;
          case 'b':
               if (aflg)
                    errflg++;
               else
                    bproc( );
               break;
          case 'f':
               ifile = optarg;
               break;
          case 'o':
               ofile = optarg;
               bufsiza = 512;
               break;
          case '?':
               errflg++;
          }
     if (errflg) {
          fprintf (stderr, "usage: . . . ");
          exit (2);
     }
     for ( ; optind < argc; optind++) {
          if (access (argv[optind], 4)) {
      .
      .
      .
}
```

SEE ALSO

getopt(1).

NAME

getpass - read a password

SYNOPSIS

```
char *getpass (prompt)
char *prompt;
```

DESCRIPTION

Getpass reads up to a newline or **EOF** from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echo. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a **NULL** pointer is returned. An interrupt terminates input and sends an interrupt signal to the calling program before returning.

FILES

/dev/tty

SEE ALSO

crypt(3C).

WARNING

The above routine uses **<stdio.h>**. This causes the size of programs not otherwise using standard I/O to increase more than might be expected.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpw - get name from UID

SYNOPSIS

```
int getpw (uid, buf)
int uid;
char *buf;
```

DESCRIPTION

Getpw searches the password file for a user id number that equals _uid_, copies the line of the password file in which _uid_ was found into the array pointed to by _buf_, and returns 0. _Getpw_ returns non-zero if _uid_ cannot be found.

This routine is included only for compatibility with prior systems and should not be used; see _getpwent_(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

Getpw returns non-zero on error.

WARNING

The above routine uses <stdio.h>. Therefore, the size of programs not otherwise using standard I/O is increased more than might be expected.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent - get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )
```

DESCRIPTION

Getpwent, getpwuid, and getpwnam each return a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/passwd** file. Each line in the file contains a passwd structure, declared in the <pwd.h> header file:

```
struct passwd {
     char *pw_name;
     char *pw_passwd;
     int  pw_uid;
     int  pw_gid;
     char *pw_age;
     char *pw_comment;
     char *pw_gecos;
     char *pw_dir;
     char *pw_shell;
};

struct comment {
     char *c_dept;
     char *c_name;
     char *c_acct;
     char *c_bin;
};
```

Because this structure is declared in **<pwd.h>**, it is not necessary to redeclare it.

The pw_comment field is unused; the others have meanings described in passwd(4).

When first called, getpwent returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; therefore, successive calls can be used to search the entire file. *Getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found; it returns a pointer to the particular structure in which the match was found. *Getpwnam* searches from the beginning of the file until a login name matching *name* is found; it returns a pointer to the particular structure in which the match was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *Endpwent* may be called to close the password file when processing is complete.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(4).

DIAGNOSTICS

A NULL pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**. Therefore the size of programs not otherwise using standard I/O is increased more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, _stdin_, into the array pointed to by _s_, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

Fgets reads characters from the _stream_ into the array pointed to by _s_ until _n_-1 characters are read, or a new-line character is read and transferred to _s_, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to _s_ and a NULL pointer is returned. If a read error (e.g., trying to use these functions on a file that has not been opened for reading) occurs, a NULL pointer is returned. Otherwise _s_ is returned.

NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname - access utmp file entry

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file )
char *file;
```

DESCRIPTION

Getutent, *getutid*, and *getutline* each return a pointer to a structure of the following type:

```
struct utmp {
        char       ut_user[8];          /* User login name */
        char       ut_id[4];            /* /etc/inittab id (usually lin
        char       ut_line[12];         /* device name (console, lnxx)
        short      ut_pid;              /* process id */
        short      ut_type;             /* type of entry */
        struct     exit_status {
            short      e_termination;   /* Process termination status *
            short      e_exit;          /* Process exit status */
        } ut_exit;                      /* The exit status of a process
                                        /* marked as DEAD_PROCESS. */
        time_t     ut_time;             /* time entry was made */
};
```

Getutent reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

Getutid searches forward from the current point in the *utmp file* until it finds an entry with a *ut type* matching *id->ut type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME, or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut id* field matches *id->ut_id*. *Getutid* fails if the end of file is reached without a match.

Getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the *line->ut_line* string. If the end of file is reached without a match, it fails.

Pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is assumed that the user of *pututline* has searched for the proper entry using one of the *getut* routines. If this has been done, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

Setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

Endutent closes the currently open file.

Utmpname allows the user to change the name of the file examined from **/etc/utmp** to any other filename. It is expected that most often this other file will be **/etc/wtmp**. If the file doesn't exist, this will not be apparent until the first attempt to reference the file is made. *Utmpname* does not open the file. It just closes the old file, if it is currently open, and saves the new filename.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A **NULL** pointer is returned upon failure to read or write. Failure to read may be due to permissions or because end-of-file has been reached.

COMMENTS

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the search of the static structure results in a match, no further search is performed. To use *getutline* to search for multiple occurences, zero out the static structure after each success; otherwise *getutline* will just return the same pointer over and over again.

There is one exception to the rule about removing the structure before further reads are done. If the implicit read done by _pututline_ finds that it isn't already at the correct place in the file, the contents of the static structure returned by the _getutent_, _getutid_, or _getutline_ routines are not harmed, if the user has just modified those contents and passed the pointer back to _pututline_.

These routines use buffered standard I/O for input, but _pututline_ uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the _utmp_ and _wtmp_ files.

NAME

hsearch, hcreate, hdestroy - manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )
```

DESCRIPTION

Hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. Item is a structure of type ENTRY (defined in the <search.h> header file) containing two pointers. Item.key points to the comparison key and item.data points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) Action is a member of an enumeration type ACTION, indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

Hcreate allocates sufficient space for the table and must be called before hsearch is used. Nel is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

Hdestroy destroys the search table and may be followed by another call to hcreate.

NOTES

Hsearch uses open addressing with a multiplicative hash function. However, many other options are available in the source code. The user may select an option by compiling the hsearch source with the following symbols defined to the preprocessor:

DIV Use the remainder modulo table size as the hash function instead of the multiplicative algorithm.

USCR Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named hcompar and should behave in a mannner similar to *strcmp* (see *string*(3C)).

CHAINED Use a linked list to resolve collisions. If this option is selected, the following other options become available.

- **START** Place new entries at the beginning of the linked list (default is at the end).
- **SORTUP** Keep the linked list sorted by key in ascending order.
- **SORTDOWN** Keep the linked list sorted by key in descending order.

Additionally, there are preprocessor flags for obtaining a debugging printout (-DDEBUG) and for including a test driver in the calling routine (-DDRIVER). The source code should be consulted for further details.

SEE ALSO

bsearch(3C), lsearch(3C), string(3C), tsearch(3C).

DIAGNOSTICS

Hsearch returns a NULL pointer if either the action is **FIND** and the item could not be found or the action is **ENTER** and the table is full.

Hcreate returns zero if it cannot allocate sufficient space for the table.

BUGS

Only one hash search table may be active at any given time.

NAME

hypot - Euclidean distance function

SYNOPSIS

```
#include <math.h>

double hypot (x, y)
double x, y;
```

DESCRIPTION

Hypot returns the following, taking precautions against unwarranted overflows:

```
sqrt(x * x + y * y)
```

DIAGNOSTICS

When the correct value would overflow, *hypot* returns **HUGE** and sets *errno* to **ERANGE**.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M), sqrt(3F).

NAME

l3tol, ltol3 - convert between 3-byte integers and long integers

SYNOPSIS

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION

L3tol converts a list of n 3-byte integers (packed into a character string pointed to by cp) into a list of long integers pointed to by lp.

Ltol3 performs the reverse conversion from long integers (lp) to 3-byte integers (cp).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

SEE ALSO

fs(4).

BUGS

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent:

NAME

ldahread - read the archive header of a member of an archive file

SYNOPSIS

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <ldfcn.h>


int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

DESCRIPTION

If **TYPE**(_ldptr_) is the archive file magic number, _ldahread_ reads the archive header of the common object file currently associated with _ldptr_ into the area of memory beginning at _arhead_.

Ldahread returns **SUCCESS** or **FAILURE**. _Ldahread_ fails if **TYPE**(_ldptr_) does not represent an archive file or if it cannot read the archive header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldclose, ldaclose - close a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>


int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldopen(3X) and _ldclose_ are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If **TYPE(**_ldptr_**)** does not represent an archive file, _ldclose_ closes the file and frees the memory allocated to the LDFILE structure associated with _ldptr_. If **TYPE(**_ldptr_**)** is the magic number of an archive file, and if there are any more files in the archive, _ldclose_ reinitializes **OFFSET(**_ldptr_**)** to the file address of the next archive member and returns **FAILURE**. The LDFILE structure is prepared for a subsequent _ldopen_(3X). In all other cases, _ldclose_ returns **SUCCESS**.

Ldaclose closes the file and frees the memory allocated to the **LDFILE** structure associated with _ldptr_ regardless of the value of **TYPE(**_ldptr_**)**. _Ldaclose_ always returns **SUCCESS**. The function is often used in conjunction with _ldaopen_.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4).

NAME

ldfhread - read the file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

DESCRIPTION

Ldfhread reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

Ldfhread returns **SUCCESS** or **FAILURE**. *Ldfhread* fails if it cannot read the file header.

In most cases the use of *ldfhread* can be avoided by using the macro HEADER(*ldptr*) defined in <ldfcn.h> (see *ldfcn*(4)). The information in any field, *fieldname*, of the file header may be accessed using HEADER(*ldptr*).*fieldname*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldgetname - retrieve symbol name for object file symbol table entry

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char ldgetname (ldptr, symbol)
LDFILE ldptr;
SYMENT symbol;
```

DESCRIPTION

Ldgetname returns a pointer to the name associated with _symbol_ as a string. The string is contained in a static buffer local to _ldgetname_. Because the buffer is overwritten by each call to _ldgetname_, it must be copied by the caller if the name is to be saved.

The common object file format has been extended to handle arbitrary length symbol names with the addition of a "string table". _Ldgetname_ returns the symbol name associated with a symbol table entry for either an object file or a pre-object file. Thus, _ldgetname_ can be used to retrieve names from object files without any backward compatibility problems. _Ldgetname_ returns **NULL** (defined in **<stdio.h>**) for an object file if the name cannot be retrieved. This occurs when:

-- the string table cannot be found.

-- not enough memory can be allocated for the string table.

-- the string table appears not to be a string table (e.g., if an auxiliary entry is handed to _ldgetname_ that looks like a reference to a name in a non-existent string table).

-- the name's offset into the string table is beyond the end of the string table.

Typically, _ldgetname_ is called immediately after a successful call to _ldtbread_ to retrieve the name associated with the symbol table entry filled by _ldtbread_.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldtbread(3X),

NAME

ldlread, ldlinit, ldlitem - manipulate line number entries of a common object file function

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>


int ldlread (ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO linent;

int ldlinit (ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem (ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO linent;
```

DESCRIPTION

Ldlread searches the line number entries of the common object file currently associated with *ldptr*. *Ldlread* begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. *Ldlread* reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlinit and *ldlitem* together perform exactly the same function as *ldlread*. After an initial call to *ldlread* or *ldlinit*, *ldlitem* may be used to retrieve a series of line number entries associated with a single function. *Ldlinit* simply locates the line number entries for the function identified by *fcnindx*. *Ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlread, *ldlinit*, and *ldlitem* each return either **SUCCESS** or **FAILURE**. *Ldlread* fails if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. *Ldlinit* fails if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. *Ldlitem* fails if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbindex(3X), ldfcn(4).

NAME

ldlseek, ldnlseek - seek to line number entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldlseek seeks to the line number entries of the section specified by _sectindx_ of the common object file currently associated with _ldptr_.

Ldnlseek seeks to the line number entries of the section specified by _sectname_.

Ldlseek and _ldnlseek_ return **SUCCESS** or **FAILURE**. _Ldlseek_ fails if _sectindx_ is greater than the number of sections in the object file; _ldnlseek_ fails if there is no section name corresponding to *_sectname_. Either function fails if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of _one_.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldohseek - seek to the optional file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldohseek seeks to the optional file header of the common object file currently associated with _ldptr_.

Ldohseek returns **SUCCESS** or **FAILURE**. _Ldohseek_ fails if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfhread(3X), ldfcn(4).

NAME

ldopen, ldaopen - open a common object file for reading

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

DESCRIPTION

Ldopen and _ldclose_(3X) are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.

If _ldptr_ has the value **NUll**, _ldopen_ opens _filename_, allocates and initializes the **LDFILE** structure, and returns a pointer to the structure to the calling program.

If _ldptr_ is valid and **TYPE(**_ldptr_**)** is the archive magic number, _ldopen_ reinitializes the **LDFILE** structure for the next archive member of _filename_.

Ldopen and _ldclose_ are designed to work in concert. _Ldclose_ returns **FAILURE** only when **TYPE(**_ldptr_**)** is the archive magic number and there is another file in the archive to be processed. Only then should _ldopen_ be called with the current value of _ldptr_. In all other cases, in particular whenever a new _filename_ is opened, _ldopen_ should be called with a **NULL** _ldptr_ argument.

The following is a prototype for the use of _ldopen_ and _ldclose_.

```
/* for each filename to be processed */

ldptr = NULL;
do
     if ( (ldptr = ldopen(filename, ldptr)) != NULL )

     {
          /* check magic number */
          /* process the file */
     }
} while (ldclose(ldptr) == FAILURE );
```

If the value of _oldptr_ is not **NULL**, _ldaopen_ opens _filename_ anew and allocates and initializes a new **LDFILE** structure, copying the **TYPE**, **OFFSET**, and **HEADER** fields from _oldptr_. _Ldaopen_ returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, _oldptr_. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both _ldopen_ and _ldaopen_ open _filename_ for reading. Both functions return **NULL** if _filename_ cannot be opened or if memory for the **LDFILE** structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fopen(3S), ldclose(3X), ldfcn(4).

NAME

ldrseek, ldnrseek - seek to relocation entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldrseek seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnrseek seeks to the relocation entries of the section specified by *sectname*.

Ldrseek and *ldnrseek* return **SUCCESS** or **FAILURE**. *Ldrseek* fails if *sectindx* is greater than the number of sections in the object file; *ldnrseek* fails if there is no section name corresponding with *sectname*. Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldshread, ldnshread - read an indexed/named section header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char sectname;
SCNHDR *secthead;
```

DESCRIPTION

Ldshread reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

Ldnshread reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

Ldshread and *ldnshread* return **SUCCESS** or **FAILURE**. *Ldshread* fails if *sectindx* is greater than the number of sections in the object file; *ldnshread* fails if there is no section name corresponding with *sectname*. Either function fails if it cannot read the specified section header.

Note that the first section header has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldsseek, ldnsseek - seek to an indexed/named section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldsseek seeks to the section specified by _sectindx_ of the common object file currently associated with _ldptr_.

Ldnsseek seeks to the section specified by _sectname_.

Ldsseek and _ldnsseek_ return **SUCCESS** or **FAILURE**. _Ldsseek_ fails if _sectindx_ is greater than the number of sections in the object file; _ldnsseek_ fails if there is no section name corresponding with _sectname_. Either function fails if there is no section data for the specified section or if it cannot seek to the specified section.

Note that the first section has an index of _one_.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldtbindex - compute the index of a symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbindex returns the (**long**) index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by *ldtbindex* may be used in subsequent calls to *ldtbread*(3X). However, since *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file, if *ldtbindex* is called immediately after a particular symbol table entry has been read, it returns the the index of the next entry.

Ldtbindex fails if there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbread - read an indexed symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

DESCRIPTION

Ldtbread reads the symbol table entry specified by _symindex_ of the common object file currently associated with _ldptr_ into the area of memory beginning at _symbol_.

Ldtbread returns **SUCCESS** or **FAILURE**. _Ldtbread_ fails if _symindex_ is greater than the number of symbols in the object file or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of _zero_.

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldgetname(3X), ldopen(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbseek - seek to the symbol table of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbseek seeks to the symbol table of the object file currently associated with *ldptr*.

Ldtbseek returns **SUCCESS** or **FAILURE**. *Ldtbseek* fails if the symbol table has been stripped from the object file or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4).

NAME

logname - return login name of user

SYNOPSIS

char *logname( )

DESCRIPTION

Logname returns a pointer to the null-terminated login name; it extracts the **$LOGNAME** variable from the user's environment.

This routine is kept in **/lib/libPW.a.**

FILES

/etc/profile

SEE ALSO

env(1), login(1), profile(4), environ(5).

BUGS

The return values point to static data whose content is overwritten by each call.

This method of determining a login name is subject to forgery.

NAME

lsearch - linear search and update

SYNOPSIS

```
char *lsearch ((char *)key,(char *)base, nelp, sizeof(*key),
compar
unsigned *nelp;
int (*compar)( );
```

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where data may be found. If the data does not occur, it is added at the end of the table. Key points to the data to be sought in the table. Base points to the first element in the table. Nelp points to an integer containing the current number of elements in the table. The integer is incremented if the data is added to the table. Compar is the name of the comparison function which the user must supply (strcmp, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

bsearch(3C), hsearch(3C), tsearch(3C).

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

malloc, free, realloc, calloc - main memory allocator

SYNOPSIS

```
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Malloc allocates the first contiguous reach of free space of sufficient size found in a circular search from the last block allocated or freed; it coalesces adjacent free blocks as it searches. It calls *sbrk* (see *brk*(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, *realloc* asks *malloc* to enlarge the arena by *size* bytes and then moves the data to the new space.

Realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc*, and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS

Malloc, *realloc*, and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

NOTE

Search time increases when many objects have been allocated; i.e., if a program allocates space but never frees it, each successive allocation takes longer.

NAME

matherr - error-handling function

SYNOPSIS

```
#include <math.h>

int matherr (x)
struct exception *x;
```

DESCRIPTION

Matherr is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors by including a function named _matherr_ in their programs. _Matherr_ must be of the form described above. A pointer to the exception structure _x_ will be passed to the user-supplied _matherr_ function when an error occurs. This structure, which is defined in the **<math.h>** header file, is as follows:

```
struct exception {
     int type;
     char *name;
     double arg1, arg2, retval;
};
```

The element _type_ is an integer describing the type of error that has occurred; one of the following constants (defined in the header file) is used:

DOMAIN	domain error
SING	singularity
OVERFLOW	overflow
UNDERFLOW	underflow
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element _name_ points to a string containing the name of the function that had the error. The variables _arg1_ and _arg2_ are the arguments to the function that had the error. _Retval_ is a double that is returned by the function having the error. If it supplies a return value, the user's _matherr_ must return non-zero. If the default error value is to be returned, the user's _matherr_ must return 0.

If _matherr_ is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are summarized in the table following the example below. In every case, _errno_ is set to non-zero and the program continues.

EXAMPLE

```
matherr(x)
register struct exception *x;
{
     switch (x->type) {
     case DOMAIN:
     case SING: /* print message and abort */
          fprintf(stderr, "domain error in %s\n", x->name);
          abort( );
     case OVERFLOW:
          if (!strcmp("exp", x->name)) {
               /* if exp, print message, return the argument */
               fprintf(stderr, "exp of %f\n", x->arg1);
               x->retval = x->arg1;
          } else if (!strcmp("sinh", x->name)) {
               /* if sinh, set errno, return 0 */
               errno = ERANGE;
               x->retval = 0;
          } else
               /* otherwise, return HUGE */
               x->retval = HUGE;
          break;
     case UNDERFLOW:
          return (0); /* execute default procedure */
     case TLOSS:
     case PLOSS:
          /* print message and return 0 */
          fprintf(stderr, "loss of significance in %s\n", x->nam ;
          x->retval = 0;
          break;
     }
     return (1);
}
```

DEFAULT ERROR HANDLING PROCEDURES						
	Types of Errors					
	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
BESSEL:	-	-	H	0	-	*
y0, y1, yn (neg. no.)	M, -H	-	-	-	-	-
EXP:	-	-	H	0	-	
POW:	-	-	H	0	-	-
(neg.)**(non-int.), 0**0	M, 0	-	-	-	-	-
LOG:						
log(0):	-	M, -H	-	-	-	-
log(neg.):	M, -H	-	-	-	-	-
SQRT:	M, 0	-	-	-	-	-
GAMMA:	-	M, H	-	-	-	-
HYPOT:	-	-	H	-	-	-
SINH, COSH:	-	-	H	-	-	-
SIN, COS:	-	-	-	-	M, 0	M, *
TAN:	-	-	H	-	0	*
ACOS, ASIN:	M, 0	-	-	-	-	-

ABBREVIATIONS	
*	As much as possible of the value is returned.
M	Message is printed.
H	HUGE is returned.
-H	-HUGE is returned.
0	0 is returned.

NAME

memccpy, memchr, memcmp, memcpy, memset - memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memccpy copies characters from memory area s2 into s1, stopping after the first occurrence of character c has been copied or after n characters have been copied, whichever comes first. It returns either a pointer to the character after the copy of c in s1 or a NULL pointer if c was not found in the first n characters of s2.

Memchr returns either a pointer to the first occurrence of character c in the first n characters of memory area s or a NULL pointer if c does not occur.

Memcmp compares its arguments, looking at the first n characters only. It returns an integer less than, equal to, or greater than 0, depending on whether s1 is lexicographically less than, equal to, or greater than s2.

Memcpy copies n characters from memory area s2 to s1. It returns s1.

Memset sets the first n characters in memory area s to the value of character c. It returns s .

NOTE

For user convenience, all these functions are declared in the optional <memory.h> header file.

BUGS

Memcmp uses native character comparison, which is signed on PDP-11s, unsigned on other machines.

Because character movement is performed differently in different implementations, overlapping moves may yield unexpected results.

NAME

mktemp - make a unique filename

SYNOPSIS

char *mktemp (template)
char *template;

DESCRIPTION

Mktemp replaces the contents of the string pointed to by *template* with a unique filename; it returns the address of *template*. The string in *template* should look like a filename with six trailing **X**s; *mktemp* replaces the **X**s with a letter and the current process ID. The letter is chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

BUGS

It is possible to run out of letters.

NAME

monitor - prepare execution profile

SYNOPSIS

```
void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
short *buffer;
int bufsize, nfunc;
```

DESCRIPTION

An executable program created by cc -p automatically includes calls for *monitor* with default parameters; *monitor* needn't be called explicitly except to gain fine control over profiling.

Monitor is an interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. *Monitor* arranges to record a histogram in the buffer. This histogram shows periodically sampled values of the program counter and counts of calls of certain functions. The lowest address sampled is that of *lowpc*; the highest address is just below *highpc*. *Lowpc* may not equal 0 for this use of *monitor*. *Nfunc* is the maximum number of call counts that can be kept; only calls of functions compiled with the profiling option -p of *cc*(1) are recorded. (The C Library and Math Library supplied when cc -p is used also have call counts recorded.) For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
...
monitor ((int (*)())2, etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text; see *end*(3C).

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor ((int (*)())NULL, 0, 0, 0, 0);
```

Prof(1) can then be used to examine the results.

FILES

mon.out

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

NAME

nlist - get entries from name list

SYNOPSIS

```
#include <a.out.h>

int nlist (filename, nl)
char *filename;
struct nlist nl[ ];
```

DESCRIPTION

Nlist examines the name list in the executable file whose name is pointed to by *filename*; it selectively extracts a list of values and puts them in the array of *nlist* structures pointed to by *nl*. The name list *nl* consists of an array of structures containing names of variables, types, and values. The list is terminated with a null name; i.e., a null string is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See *a.out*(4) for a discussion of the symbol table structure.

This subroutine is useful for examining the system name list kept in the file **/unix**. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out(4).

DIAGNOSTICS

All type entries are set to 0 if the file cannot be read or if it doesn't contain a valid name list.

Nlist returns -1 upon error; otherwise it returns 0.

NAME

perror, errno, sys_errlist, sys_nerr - system error messages

SYNOPSIS

```
void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;
```

DESCRIPTION

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *Sys nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2).

NAME

popen, pclose - initiate pipe to/from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;
```

DESCRIPTION

The arguments to _popen_ are pointers to null-terminated strings; one string contains a shell command line and the other contains an I/O mode. The mode may be either **r** for reading or **w** for writing. _Popen_ creates a pipe between the calling program and the command to be executed. The value returned is a stream pointer. If the I/O mode is **w**, one can write to the standard input of the command by writing to the file _stream_; if the I/O mode is **r**, one can read from the standard output of the command, by reading from the file _stream_.

A stream opened by _popen_ should be closed by _pclose_, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter and a type **w** as an output filter.

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), system(3S).

DIAGNOSTICS

Popen returns a NULL pointer if files or processes cannot be created or if the shell cannot be accessed.

Pclose returns -1 if _stream_ is not associated with a command opened by _popen_.

BUGS

If the original processes and processes opened by _popen_ concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Problems with an output filter may be forestalled by careful buffer flushing, e.g., by using _fflush_; see _fclose_(3S).

NAME

printf, fprintf, sprintf - print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf (format [ , arg ] ... )
char *format;

int fprintf (stream, format [ , arg ] ... )
FILE *stream;
char *format;

int sprintf (s, format [ , arg ] ... )
char *s, format;
```

DESCRIPTION

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places ``output'', followed by the null character (\0) in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the \0 in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

> Zero or more *flags*, which modify the meaning of the conversion specification.
>
> An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded to the field width on the left (default) or right (if the left-adjustment flag has been given); see below for flag specification.
>
> A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e** and **f** conversions, the maximum number of significant

digits for the **g** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The format of the precision is a period (**.**) followed by a decimal digit string; a null digit string is treated as zero.

An optional **l** specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer _arg_.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*****) instead of a digit string. In this case, an integer _arg_ supplies the field width or precision. The _arg_ that is actually converted is not fetched until the conversion letter is seen; therefore, the _args_ specifying field width or precision must appear _before_ the _arg_ (if any) to be converted.

The flag characters and their meanings are:

- The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (**+** or **-**).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and **+** flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an ``alternate form.'' For c, d, s, and u conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** (**X**) conversion, a non-zero result will have **0x** (**0X**) prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will _not_ be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X The integer _arg_ is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f The float or double arg is converted to decimal notation in the style ``[-]ddd.ddd'', where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are output; if the precision is explicitly 0, no decimal point appears.

e,E The float or double arg is converted in the style ``[-]d.ddd**e±**dd'', where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The **E** format code produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The float or double arg is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** is used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

c The character arg is printed.

s The arg is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. If the string pointer arg has the value zero, the result is undefined. A null arg yields undefined results.

% Print a **%**; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form ``Sunday, July 3, 10:02'', where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %.2d:%.2d", weekday, month, day, hour, min);
```

To print *pi* to 5 decimal places:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S).

NAME

putc, putchar, fputc, putw - put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc (c, stream)
char c;
FILE *stream;

int putchar (c)
char c;

int fputc (c, stream)
char c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Putc writes the character *c* onto the output *stream* at the position where the file pointer, if defined, is pointing. *Putchar*(*c*) is defined as *putc*(*c*, *stdout*). *Putc* and *putchar* are macros.

Fputc behaves like *putc*, but is a function rather than a macro. *Fputc* runs more slowly than *putc*, but takes less space per invocation.

Putw writes the word (i.e., integer) *w* to the output *stream* at the position at which the file pointer, if defined, is pointing. The size of a word is the size of an integer and varies from machine to machine. *Putw* neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen*(see *fopen*(3S)) causes it to become buffered or line-buffered. When an output stream is unbuffered information, it is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block; when it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (i.e., as soon as a new-line character is written or terminal input is requested). *Setbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant EOF. This occurs if the file *stream* is not open for writing or if the output file cannot be grown. Because EOF is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

BUGS

Because it is implemented as a macro, *putc* treats incorrectly a *stream* argument with side effects. In particular, putc(c, *f++); doesn't work sensibly. *Fputc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent and may not be read using *getw* on a different processor. For this reason the use of *putw* should be avoided.

NAME

putpwent - write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

Putpwent is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwuid* writes a line on the stream *f* which matches the format of /etc/passwd.

The <pwd.h> header file is described in *getpwent*(3C).

SEE ALSO

getpwent(3C).

DIAGNOSTICS

Putpwent returns non-zero if an error was detected during its operation; otherwise it returns zero.

WARNING

The above routine uses <stdio.h>. Therefore, the size of programs not otherwise using standard I/O is increased more than might be expected.

NAME

puts, fputs - put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Puts writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream *stdout*.

Fputs writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

DIAGNOSTICS

Both routines return **EOF** on error. This occurs if the routines try to write on a file that has not been opened for writing.

NOTES

Puts appends a new-line character while *fputs* does not.

NAME

qsort - quicker sort

SYNOPSIS

```
void qsort ((char *) base, nel, sizeof (*base), compar
unsigned int nel;
int (*compar)( );
```

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Base points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. Depending on whether the first argument is to be considered less than, equal to, or greater than the second argument, the *compar* function must return an integer less than, equal to, or greater than zero.

NOTES

The pointer to the base of the table should be of type pointer-to-element and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), string(3C).

NAME

rand, srand - simple random-number generator

SYNOPSIS

```
int rand ( )

void srand (seed)
unsigned seed;
```

DESCRIPTION

Rand uses a multiplicative congruential random-number generator with period 28329 that returns successive pseudo-random numbers in the range from 0 to 28159-1.

Srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTE

The spectral properties of rand leave a great deal to be desired. Drand48(3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C).

NAME

regcmp, regex - compile and execute a regular expression

SYNOPSIS

```
char *regcmp(string1 [, string2, ...], 0)
char *string1, *string2, ...;

char *regex(re, subject[, ret0, ...])
char *re, *subject, *ret0, ...;

extern char *loc1;
```

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. Malloc(3C) is used to create space for the vector. It is the user's responsibility to free unneeded space that has been allocated by malloc. A NULL return from regcmp indicates an incorrect argument. Regcmp(1) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. Regex returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer loc1 points to where the match began. Regcmp and regex were mostly borrowed from the editor, ed(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[]*.^ These symbols retain their current meaning.

$ This symbol matches the end of the string; \n matches the new-line.

\- Within brackets the minus means "through". For example, [a-z] is equivalent to [abcd...xyz]. The - can appear as itself only if used as the last or first character. For example, the character class expression []-] matches the characters] and -.

\+ A regular expression followed by + means "one or more times". For example, [0-9]+ is equivalent to [0-9][0-9]*.

{m} {m,} {m,u} Integer values enclosed in {} indicate the number of times the preceding regular expression is to be applied. The minimum number is m and the maximum number is u, which must be less than 256. If only m is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,}, respectively.

(...)$*n* The value of the enclosed regular expression is to be returned. The value will be stored in the (*n*+*1*)th argument following the subject argument. At present, at most 10 enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

(...) Parentheses are used for grouping. An operator (e.g., *, +, {}) can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
     ...
newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
char *newcursor, *name;
     ...
name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0", 0);
newcursor = regex(name, "123Testing321", ret0);
```

This example will match through the string ``Testing3'' and will return the address of the character after the last matched character (cursor+11). The string ``Testing3'' will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
     ...
newcursor = regex(name, string);
```

This example applies a precompiled regular expression in **file.i** (see *regcmp*(1)) against *string*.

This routine is kept in **/lib/libPW.a**.

SEE ALSO

ed(1), regcmp(1), malloc(3C).

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required. The following user-supplied replacement for *malloc*(3C) reuses the same vector, saving time and space:

```
/* user's program */
      ...
malloc(n) {
      static int rebuf[256];
      return rebuf;
}
```

NAME

scanf, fscanf, sscanf - convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf (format [ , pointer ] ... )
char *format;

int fscanf (stream, format [ , pointer ] ... )
FILE *stream;
char *format;

int sscanf (s, format [ , pointer ] ... )
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to *format*, and stores the results in its arguments. Each function expects two arguments: a control string *format* (described below) and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks and tabs) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional numerical maximum field width, an optional **l** or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression has been indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-white-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument should be given. The following conversion codes are legal:

% A single **%** is expected in the input at this point; no assignment is done.

d A decimal integer is expected; the corresponding argument should be an integer pointer.

u An unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o An octal integer is expected; the corresponding argument should be an integer pointer.

x A hexadecimal integer is expected; the corresponding argument should be an integer pointer.

e,f,g
A floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a _float_. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optionally signed integer.

s A character string is expected; the corresponding argument should be a character pointer to an array of characters large enough to accept the string and a terminating \0, which will be added automatically. The input field is terminated by a white-space character.

c A character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[String data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters (the _scanset_) and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the _scanset_. The circumflex, (^), when it appears as the first character in the _scanset_, serves as a complement operator and redefines the _scanset_ as the set of all characters _not_ contained in the remainder of the _scanset_ string. There are some conventions used in the construction of the _scanset_. A range of characters may be represented by the construct _first-last_; thus, **[0123456789]** may be expressed **[0-9]**. Using this convention, _first_ must be lexically less than or equal to _last_, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the *scanset*. To include the right square bracket as an element of the *scanset*, it must appear as the first character (possibly preceded by a circumflex) of the *scanset*; otherwise it will be interpreted syntactically as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \0, which will be added automatically.

The conversion characters **d**, **u**, **o**, and **x** may be preceded by **l** or **h** to indicate that a pointer to **long** or **short**, rather than **int**, is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double**, rather than **float**, is in the argument list.

Scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero when an early conflict between an input character and the control string occurs. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call

```
int i; float x; char name[50];
scanf ("%d%f%s", &i, &x, name);
```

with the input line

```
25 54.32E-1 thompson
```

will assign the value 25 to *i*, and the value 5.432 to *x*; *name* will contain **thompson\0**.

The call

```
int i; float x; char name[50];
scanf ("%2d%f%*d %[0-9]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

atof(3C), getc(3S), printf(3S), strtol(3C).

NOTE

Trailing white space is left unread unless matched in the control string.

DIAGNOSTICS

These functions return **EOF** on end of input and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf - assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;
```

DESCRIPTION

Setbuf is used after a stream has been opened but before it is read or written. It causes the character array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is a NULL character pointer, input/output will be completely unbuffered.

A constant **BUFSIZ**, defined in the **<stdio.h>** header file, tells how big an array is needed:

charbuf[BUFSIZ];

A buffer is normally obtained from *malloc*(3C) at the time of the first *getc*(3S) or *putc*(3S) on the file, except that the standard error stream *stderr* is normally not buffered.

Output streams directed to terminals are always line-buffered unless they are unbuffered.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S).

NOTE

A common source of error is allocating buffer space as an ``automatic'' variable in a code block and then failing to close the stream in the same block.

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* for later use by *longjmp*. The environment type *jmp_buf* is defined in the <setjmp.h> header file. *Setjmp* returns the value 0.

Longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. All accessible data have values as of the time *longjmp* was called.

SEE ALSO

signal(2).

WARNING

Longjmp fails if it is called when *env* was never primed by a call to *setjmp* or when the last such call is in a function which has since returned.

NAME

sinh, cosh, tanh - hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

DESCRIPTION

Sinh, _cosh_, and _tanh_ return, respectively, the hyberbolic sine, cosine, and tangent of their argument.

DIAGNOSTICS

Sinh and _cosh_ return **HUGE** when the correct value would overflow and set _errno_ to **ERANGE.**

These error-handling procedures may be changed with the function _matherr_(3M).

SEE ALSO

matherr(3M).

NAME

sleep - suspend execution for interval

SYNOPSIS

unsigned sleep (seconds)
unsigned seconds;

DESCRIPTION

Sleep suspends the current process from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) any caught signal will terminate *sleep* following execution of the signal catching routine. The suspension time may be longer than requested by an arbitrary amount, due to the scheduling of other activity in the system. The value returned by *sleep* is the ``unslept'' amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time or in case there is premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time before the alarm signal, the process sleeps only until the alarm signal would have occurred and the caller's alarm catch routine is executed just before the *sleep* routine returns. If the *sleep* time is less than the time before the calling program's alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2), pause(2), signal(2).

NAME

sputl, sgetl - access long integer data in a machine independent fashion.

SYNOPSIS

```
void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;
```

DESCRIPTION

Sputl takes the 4 bytes of the long integer value and places them in memory, starting at the address pointed to by buffer. The ordering of the bytes is the same across all machines.

Sgetl retrieves the 4 bytes in memory, starting at the address pointed to by buffer, and returns the long integer value in the byte ordering of the host machine.

Use of sputl and sgetl in combination provides a machine independent way of storing long numeric data in a file in binary form without conversion to characters.

A program that uses these functions must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ar(4).

NAME

ssignal, gsignal - software signals

SYNOPSIS

```
#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;
```

DESCRIPTION

Ssignal and gsignal implement a software facility similar to signal(2). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions; it is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. A call to ssignal associates a procedure, action, with the software signal, sig; the software signal, sig, is raised by a call to gsignal. Raising a software signal causes the action established for that signal to be taken.

The first argument to ssignal is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a user-defined action function or one of the manifest constants SIG_DFL (default) or SIG_IGN (ignore). Ssignal returns the action previously established for that signal type; if no action has been established or the signal number (sig) is illegal, ssignal returns SIG_DFL.

Gsignal raises the signal identified by its argument, sig:

> If an action function has been established for sig, then that action is reset to SIG_DFL and the action function is entered with argument sig. Gsignal returns the value returned to it by the action function.
>
> If the action for sig is SIG_IGN, gsignal returns the value 1 and takes no other action.
>
> If the action for sig is SIG_DFL, gsignal returns the value 0 and takes no other action.
>
> If sig has an illegal value or no action was ever specified for sig, gsignal returns the value 0 and takes no other action.

NOTES

There are some additional signals with numbers outside the range 1 through 15 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

NAME

stdio - standard buffered input/output package

SYNOPSIS

```
#include <stdio.h>

FILE *stdin, *stdout, *stderr;
```

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The input/output function may be grouped into the following categories: file access, file status, input, output, miscellaneous. For lists of the functions in each category, refer to the "Libraries" section of the *Programming Guide*. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **<stdio.h>** header file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file.

A constant **NULL** (0) designates a nonexistent pointer.

An integer constant **EOF** (-1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that header file and need no further declaration. The constants and the following functions are implemented as macros: *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, *clearerr*, and *fileno*. Redeclaration of these names is perilous.

The **<stdio.h>** file is illustrated in the "Libraries" section of the *Programming Guide*.

SEE ALSO

open(2), close(2), lseek(2), pipe(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S).

DIAGNOSTICS

Invalid *stream* pointers cause serious errors, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

stdipc - standard interprocess communication package

SYNOPSIS

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

key_t ftok(path, id)
char *path;
char id;
```

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If a standard is not adhered to, unrelated processes may interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

Ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *Path* must be the pathname of an existing file that is accessible to the process. *Id* is a character that uniquely identifies a project. *Ftok* returns the same key for linked files when called with the same *id*; it returns different keys when called with the same filename but different *ids*.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

DIAGNOSTICS

Ftok returns (key_t) -1 if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to *ftok* is removed when keys still refer to the file, future calls to *ftok* with the same *path* and *id* will return an error. If the same file is recreated, *ftok* is likely to return a different key than it did the original time it was called.

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

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

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok - string operations

SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

DESCRIPTION

The arguments _s1_, _s2_, and _s_ point to strings (arrays of characters terminated by a null character). The functions _strcat_, _strncat_, _strcpy_, and _strncpy_ all alter _s1_. These functions do not check for overflow of the array pointed to by _s1_.

Strcat appends a copy of string _s2_ to the end of string _s1_. _Strncat_ appends at most _n_ characters. Each function returns a pointer to the null-terminated result.

Strcmp performs a lexicographical comparison of its arguments and returns an integer less than, equal to, or greater than 0, when _s1_ is less than, equal to, or greater than _s2_, respectively. _Strncmp_ makes the same comparison but looks at a maximum of _n_ characters.

Strcpy copies string _s2_ to string _s1_, stopping after the null character has been copied. _Strncpy_ copies exactly _n_ characters, truncating _s2_ or adding null characters to _s1_ if necessary. The result is not null-terminated if the length of _s2_ is _n_ or more. Each function returns _s1_.

Strlen returns the number of characters in _s_, not including the terminating null character.

Strchr (_strrchr_) returns a pointer to the first (last) occurrence of character _c_ in string _s_, or a NULL pointer if _c_ does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string _s1_ of any character from string _s2_, or a NULL pointer if no character from _s2_ exists in _s1_.

Strspn (_strcspn_) returns the length of the initial segment of string _s1_ which consists entirely of characters from (not from) string _s2_.

Strtok considers the string _s1_ to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string _s2_. The first call (with pointer _s1_ specified) returns a pointer to the first character of the first token, and writes a null character into _s1_ immediately following the returned token. The function keeps track of its position in the string between separate calls, so that on subsequent calls (which must be made with a NULL pointer as the first argument) it works through the string _s1_ immediately following that token. This can be continued until no tokens remain. The separator string _s2_ may be different from call to call. When no token remains in _s1_, a NULL pointer is returned.

NOTE

For user convenience, all these functions are declared in the optional **<string.h>** header file.

BUGS

Strcmp and *strncmp* use native character comparison, which is signed on PDP-11s, unsigned on other machines.

Character movement is performed differently in different implementations; therefore, overlapping moves may yield unexpected results.

NAME

strtol, atol, atoi - convert string to integer

SYNOPSIS

```
long strtol (str, ptr, base)
char *str;
char **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;
```

DESCRIPTION

Strtol returns as a long integer the value represented by the character string *str*. The string is scanned up to the first character inconsistent with the base. Leading white-space characters (blanks and tabs) are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in **ptr*. If no integer can be formed, zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored; a leading **0x** or **0X** is ignored if *base* is 16.

If *base* is zero, the string itself determines the base. After an optional leading sign, a leading zero indicates octal conversion and a leading **0x** or **0X** indicates hexadecimal conversion; otherwise, decimal conversion is used.

Truncation from long to int can take place upon assignment or by an explicit cast.

Atol(*str*) is equivalent to **strtol**(*str*, **(char **)NULL, 10)**.

Atoi(*str*) is equivalent to **(***int***) strtol**(*str*, **(char **)NULL, 10)**.

SEE ALSO

atof(3C), scanf(3S).

BUGS

Overflow conditions are ignored.

NAME

swab - swap bytes

SYNOPSIS

```
void swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

DESCRIPTION

Swab copies _nbytes_ bytes pointed to by _from_ to the array pointed to by _to_, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP-11s and other machines. _Nbytes_ should be even and non-negative. If _nbytes_ is odd and positive, _swab_ uses _nbytes_-1 instead. If _nbytes_ is negative, _swab_ does nothing.

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

Administrator's Manual.

NAME

system - issue a shell command

SYNOPSIS

```
#include &lt;stdio.h&gt;

int system (string)
char *string;
```

DESCRIPTION

System causes *string* to be given to *sh*(1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

FILES

/bin/sh

SEE ALSO

sh(1), exec(2).

DIAGNOSTICS

System forks to create a child process that in turn performs *exec*(2) on /*bin*/*sh* in order to execute *string*. If the fork or exec fails, *system* returns -1 and sets *errno*.

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs - terminal independent operation routines

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, *area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int *outc)();
```

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base _termcap_(5). Note that these are low-level routines.

Tgetent extracts the entry for terminal _name_ into the buffer at _bp_. _Bp_ should be a character buffer of size 1024 and must be retained through all subsequent calls to _tgetnum_, _tgetflag_, and _tgetstr_. _Tgetent_ returns -1 if it cannot open the **termcap** file, 0 if the terminal name given does not have an entry, and 1 if successful. It looks in the environment for a TERMCAP variable. If a variable is found whose value does not begin with a slash and the terminal type _name_ is the same as the environment string TERM, the TERMCAP string is used instead of reading the **termcap** file. If the value does begin with a slash, the string is used as a pathname rather than **/etc/termcap**. This can speed up entry into programs that call _tgetent_. It can also help debug new terminal descriptions or be used to make one for your terminal if you can't write the file **/etc/termcap**.

Tgetnum gets the numeric value of capability *id*, returning -1 if is not given for the terminal. *Tgetflag* returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. *Tgetstr* gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in *termcap*(5), except for cursor addressing and padding information.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing \n, ^D or ^@ in the returned string. (Programs that call *tgoto* should be sure to turn off the **XTABS** bit(s), since *tgoto* may now output a tab. Note that programs using *termcap* should in general turn off **XTABS** anyway since some terminals use control-I for other functions, such as nondestructive space.) If a **%** sequence is given which is not understood, then *tgoto* returns OOPS.

Tputs decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable; *outc* is a routine that is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by *stty* (*2*). The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libtermcap.a	-ltermcap library
/etc/termcap	data base

SEE ALSO

ex(1), termcap(5)

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

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

tmpfile - create a temporary file

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile ()
```

DESCRIPTION

Tmpfile creates a temporary file and returns a corresponding FILE pointer. The file is automatically deleted when the process using it terminates. The file is opened for update.

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), tmpnam(3S).

NAME

tmpnam, tempnam - create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

DESCRIPTION

These functions generate filenames that can safely be used for a temporary file.

Tmpnam always generates a filename using the pathname defined as *P_tmpdir* in the <stdio.h> header file. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least *L_tmpnam* bytes, where *L_tmpnam* is a constant defined in <stdio.h>; *tmpnam* places its result in that array and returns *s*.

Tempnam allows the user to control the choice of a directory. The argument *dir* points to the pathname of the directory in which the file is to be created. If *dir* is NULL or points to a string which is not a pathname for an appropriate directory, the pathname defined as *P_tmpdir* in the <stdio.h> header file is used. If that pathname is not accessible, /tmp will be used as a last resort. This entire sequence can be upstaged by providing an environment variable TMPDIR in the user's environment, whose value is a pathname for the desired temporary-file directory.

Many applications prefer that names of temporary files contain favorite initial letter sequences. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to 5 characters to be used as the first few characters of the name of the temporary file.

Tempnam uses *malloc*(3C) to get space for the constructed filename and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* (see *malloc*(3C)). If *tempnam* cannot return the expected result for any reason (i.e., *malloc* failed or attempts to find an appropriate directory were unsuccessful), a NULL pointer will be returned.

NOTES

These functions generate a different filename each time they are called.

Files created using these functions and either *fopen*(2) or *creat*(2) are temporary only in the sense that they reside in a directory intended for temporary use and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S).

BUGS

If called more than 17,576 times in a single process, *tmpnam* and *tempnam* will start recycling previously used names. Between the time a filename is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using *tmpnam*, *tempnam*, or *mktemp*(3C) and the filenames are chosen carefully to avoid duplication by other means.

NAME

sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double tan (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double x, y;
```

DESCRIPTION

Sin, *cos*, and *tan* return, respectively, the sine, cosine, and tangent of their argument, which is in radians.

Asin returns the arcsine of *x*, in the range -J/2 to J/2.

Acos returns the arccosine of *x*, in the range 0 to J.

Atan returns the arctangent of *x*, in the range -J/2 to J/2.

Atan2 returns the arctangent of *y*/*x*, in the range -J to J, using the signs of both arguments to determine the quadrant of the return value.

DIAGNOSTICS

Sin, *cos*, and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0 when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to **ERANGE**.

Tan returns **HUGE** for an argument which is near an odd multiple of J/2 when the correct value would overflow; it sets *errno* to **ERANGE**.

Arguments of magnitude greater than 1.0 cause *asin* and *acos* to return 0 and to set *errno* to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

tsearch, tdelete, twalk - manage binary search trees

SYNOPSIS

```
#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );
```

DESCRIPTION

Tsearch is a binary tree search routine generalized from Knuth (6.2.2) Algorithm T. It returns a pointer into a tree indicating where data may be found. If the data does not occur, it is added at an appropriate point in the tree. _Key_ points to the data to be sought in the tree. _Rootp_ points to a variable that points to the root of the tree. A NULL pointer value for the variable denotes an empty tree; in this case, the variable is set to point to the data at the root of the new tree. _Compar_ is the name of the comparison function. It is called with two arguments that point to the elements being compared. If the first argument is to be considered less than, equal to, or greater than the second argument, the function must return an integer less than, equal to, or greater than zero, respectively.

Tdelete deletes a node from a binary search tree. It is generalized from Knuth (6.2.2) algorithm D. The arguments are the same as for _tsearch_. The variable pointed to by _rootp_ will be changed if the deleted node was the root of the tree. _Tdelete_ returns a pointer to the parent of the deleted node or a NULL pointer if the node is not found.

Twalk traverses a binary search tree. _Root_ is the root of the tree to be traversed. Any node in a tree may be used as the root for a walk below that node. _Action_ is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type _typedef enum_ { _preorder_, _postorder_, _endorder_, _leaf_ } _VISIT_; As defined in the **<search.h>** header file, the value of this data type depends on whether this is the first, second, or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree; the root is level zero.

NOTES

The pointers to the key and the root of the tree should be of type pointer-to-element and cast to type pointer-to-character.

The comparison function need not compare every byte; therefore, arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element. on entry.

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

BUGS

Tsearch fails if the calling function alters the pointer to the root.

WARNING

The *root* argument to *twalk* is one level of indirection less than the *rootp* arguments to *tsearch* and *tdelete*.

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.

A NULL pointer is returned by *tsearch* and *tdelete* if *rootp* is NULL

NAME

ttyname, isatty - find name of a terminal

SYNOPSIS

```
char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;
```

DESCRIPTION

Ttyname returns a pointer to a string containing the null-terminated pathname of the terminal device associated with file descriptor *fildes*.

Isatty returns 1 if *fildes* is associated with a terminal device; otherwise, it returns 0.

FILES

/dev/*

DIAGNOSTICS

Ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory /dev.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ttyslot - find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

DESCRIPTION

Ttyslot returns the index of the current user's entry in the /etc/utmp file. This is accomplished by scanning the file /etc/inittab for the name of the terminal device associated with the standard input, the standard output, or the error output (0, 1, or 2).

FILES

/etc/inittab
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of 0 is returned if an error is encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

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 re

NAME

ungetc - push character back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc (c, stream)
char c;
FILE *stream;
```

DESCRIPTION

Ungetc inserts the character _c_ into the buffer associated with an input _stream_. That character, _c_, will be returned by the next _getc_ call on that _stream_. _Ungetc_ returns _c_ and leaves the file _stream_ unchanged.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered.

If _c_ equals EOF, _ungetc_ does nothing to the buffer and returns EOF.

Fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S).

DIAGNOSTICS

For _ungetc_ to perform correctly, a read statement must have been performed prior to the call of the _ungetc_ function. _Ungetc_ returns EOF if it can't insert the character. If _stream_ is _stdin_, _ungetc_ allows exactly one character to be pushed back onto the buffer without a previous read statement.

NAME

intro - introduction to file formats

DESCRIPTION

This section outlines the header files and file formats used by C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys**.

NAME

a.out - common assembler and link editor output

DESCRIPTION

A.out is the output file from the assembler _as_(1) and the link editor _ld_(1). _A.out_ can be executed on the target machine if there were no errors in assembling or linking and no unresolved external references.

The object file format supports user-defined sections and contains extensive information for symbolic software testing. A common object file consists of a file header, an optional aout header, a table of section headers, relocation information, (optional) line numbers, and a symbol table. The order is given below.

```
File header.
Optional aout 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 four sections (relocation, line numbers, symbol table, and string table) may be missing if the program was linked with the -s option of _ld_(1) or if the symbol table and relocation bits were removed by _strip_(1). Also note that if the program was linked without the -r option, the relocation information will be absent. The string table exists only if necessary.

When an **a.out** file is loaded into memory for execution, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized data, the latter actually being initialized to all 0's), and a stack. The text segment begins at location 0 in the core image; the header is not loaded. If the magic number (the first field in the optional aout header) is 407 (octal), it indicates that the text segment is not to be write-protected or shared, so the data segment will be contiguous with the text segment. If the magic number is 410 (octal), the data segment begins at the next segment boundary following the text segment, and the text segment is not writable by the program. If other processes are executing the same **a.out** file, they will share a single text segment.

On the 3B20S, the stack begins at the end of the text and data sections and grows towards higher addresses. On the M68000 family of processors and the VAX, the stack begins at the end of memory and grows toward lower addresses. The stack is automatically extended as required. The data segment is extended only as requested by the _brk_(2) and _sbrk_(2) system calls.

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, the storage class of the symbol-table entry for that word will be marked as an ``external symbol'', and the section number will be set to 0. 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.

See _aouthdr_(4), _filehdr_(4), _linenum_(4), _scnhdr_(4), _reloc_(4), and _syms_(4) for descriptions of the individuals parts. Every section created by _as_(1) contains a multiple-of-four number of bytes; directives to _ld_(1) can create a section with an odd number of bytes.

SEE ALSO

as(1), cc(1), ld(1), aouthdr(4), 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   ushort comp_t; /* "floating point" */
          /* 13-bit fraction, 3-bit exponent  */

struct  acct
{
        char   ac_flag;     /* Accounting flag */
        char   ac_stat;     /* Exit status */
        ushort ac_uid;
        ushort ac_gid;
        dev_t  ac_tty;
        time_t ac_btime;    /* Beginning time */
        comp_t ac_utime;    /* acctng user time in clock ticks */
        comp_t ac_stime;    /* acctng system time in clock ticks */
        comp_t ac_etime;    /* acctng elapsed time in clock ticks */
        comp_t ac_mem;      /* memory usage in clicks */
        comp_t ac_io;       /* chars trnsfrd by read/write */
        comp_t ac_rw;       /* number of block reads/writes */
        char   ac_comm[8];  /* command name */
};

extern  struct acct         acctbuf;
extern  struct inode        *acctp;  /* inode of accounting file */

#define 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 using 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 **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;          /* initialized data (.data) */
        long    bsize;          /* uninitialized 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

ar - common archive file format

DESCRIPTION

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor *ld*(1).

Each archive begins with the archive magic string.

```
#define  ARMAG  "!<arch>\n"     /* magic string */
#define  SARMAG  8             /* length of magic string */
```

Each archive which contains common object files (see *a.out*(4)) includes an archive symbol table. This symbol table is used by the link editor *ld*(1) to determine which archive members must be loaded during the link edit process. The archive symbol table (if it exists) is always the first file in the archive (but is never listed) and is automatically created and/or updated by *ar*.

Following the archive magic string are the archive file members. Each file member is preceded by a file member header which is of the following format:

```
#define  ARFMAG  "`\n"    /* header trailer string */

struct    ar_hdr          /* file member header */
{
     char ar_name[16];    /* '/' terminated file member name */
     char ar_date[12];    /* file member date */
     char ar_uid[6];      /* file member user identification */
     char ar_gid[6];      /* file member group identification */
     char ar_mode[8];     /* file member mode */
     char ar_size[10];    /* file member size */
     char ar_fmag[2];     /* header trailer string */
};
```

All information in the file member headers is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for *ar_mode* which is in octal). Thus, if the archive contains printable files, the archive itself is printable.

The *ar_name* filed is blank-padded and slash (/) terminated. The *ar_date* field is the modification date of the file at the time of its insertion into the archive. Common format archives can be moved from system to system as long as the portable archive command *ar*(1) is used.

Each archive file member begins on an even byte boundary; a newline is inserted between files if necessary. Nevertheless, the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

If the archive symbol table exists, the first file in the archive has a zero length name (i.e., **ar_name[0] = '/'**). The contents of this file are as follows:

- The number of symbols. Length: 4 bytes.
- The array of offsets into the archive file. Length: 4 bytes * ``the number of symbols''.
- The name string table. Length: *ar_size* - (4 bytes * (``the number of symbols'' +1)). The number of symbols and the array of offsets are managed with *sgetl* and *sputl*. The string table contains exactly as many null terminated strings as there are elements in the offsets array. Each offset from the array is associated with the corresponding name from the string table (in order). The names in the string table are all the defined global symbols found in the common object files in the archive. Each offset is the location of the archive header for the associated symbol.

SEE ALSO

ar(1), ld(1), strip(1), sputl(3X), a.out(4).

WARNINGS

Strip(1) will remove all archive symbol entries from the header. The archive symbol entries must be restored via the **s** option of the *ar*(1) command before the archive can be used with the link editor *ld*(1).

NAME

checklist - list of file systems processed by fsck

DESCRIPTION

Checklist resides in directory **/etc** and contains a list of at most 15 _special_ _file_names. Each _special_ _file_name is contained on a separate line and corresponds to a file system. If no _file-system_ argument is provided to _fsck_(1M), each file listed in **/etc/checklist** is automatically read and checked for inconsistencies.

SEE ALSO

fsck(1M).

NAME

core - format of core image file

DESCRIPTION

The system writes out a core image of a terminated process when any of various errors occur. _Signal_(2) describes reasons for errors. The most common errors are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called **core** and is written in the working directory of the process (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter _usize_, which is defined in **/usr/include/sys/param.h.** The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the _user_ structure of the system, defined in **/usr/include/sys/user.h.** The locations of the registers are outlined in **/usr/include/sys/reg.h.**

SEE ALSO

crash(1M), sdb(1), setuid(2), signal(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%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*.

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/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:

```
#ifndef DIRSIZ
#define DIRSIZ      14
#endif
struct  direct
{
        ino_t d_ino;
        char  d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for . and ... The first is an entry for the directory itself. The second is for the parent directory. The meaning of .. is modified for the root directory of the master file system; because there is no parent, .. has the same meaning as ..

SEE ALSO

fs(4).

NAME

errfile - error-log file format

DESCRIPTION

When hardware errors are detected by the system, an error record is generated and passed to the error-logging daemon for recording in the error log for later analysis. The default error log is **/usr/adm/errfile**.

The format of an error record depends on the type of error that was encountered. Every record, however, has a header with the following format:

```
struct errhdr {
     short      e_type;    /* record type */
     short      e_len;     /* bytes in record (inc hdr) */
     time_t     e_time;    /* time of day */
};
```

The permissible record types are as follows:

```
#define E_GOTS        010  /* start for the UNIX/TS*/
#define E_GORT        011  /* start for the UNIX/RT*/
#define E_STOP        012  /* stop */
#define E_TCHG        013  /* time change */
#define E_CCHG        014  /* configuration change */
#define E_BLK         020  /* block device error */
#define E_STRAY       030  /* stray interrupt */
#define E_PRTY        031  /* memory parity */
```

Some records in the error file are of an administrative nature. These include the startup record that is entered into the file when logging is activated, the stop record that is written if the daemon is terminated ``gracefully'', and the time-change record that is used to account for changes in the system's time-of-day. These records have the following formats:

```
struct estart {
     short                e_cpu; /* CPU type */
     struct utsname       e_name;/* system names */
};

#define eend errhdr              /* record header */

struct etimchg {
     time_t               e_ntime;/* new time */
};
```

Stray interrupts cause a record with the following format to be logged:

```
struct estray {
     uint                e_saddr;  /* stray loc or device addr */
};
```

Generation of memory subsystem errors is not supported in this release.

Error records for block devices have the following format:

```
struct eblock {
       dev_t             e_dev;     /* "true" major + minor dev no */
       physadr           e_regloc;  /* controller address */
       short             e_bacty;   /* other block I/O activity */
       struct iostat {
          long           io_ops;    /* number read/writes */
          long           io_misc;   /* number "other" operations */
          ushort         io_unlog;  /* number unlogged errors */
       }                 e_stats;
       short             e_bflags;  /* read/write, error, etc */
       short             e_cyloff;  /* logical dev start cyl */
       daddr_t           e_bnum;    /* logical block number */
       ushort            e_bytes;   /* number bytes to transfer */
       paddr_t           e_memadd;  /* buffer memory address */
       ushort            e_rtry;    /* number retries */
       short             e_nreg;    /* number device registers */
};
```

The following values are used in the e_bflags word:

```
#define E_WRITE      0     /* write operation */
#define E_READ       1     /* read operation */
#define E_NOIO       02    /* no I/O pending */
#define E_PHYS       04    /* physical I/O */
#define E_FORMAT     010   /* Formatting Disk*/
#define E_ERROR      020   /* I/O failed */
```

SEE ALSO

errdemon(1M).

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    /*  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

file system - format of system volume

SYNOPSIS

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

DESCRIPTION

Every file system storage volume has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte long sectors. Sector 0 is unused and is available to contain a bootstrap program or other information.

Sector 1 is the *superblock*. The format of a superblock is:

```
/*
 * Structure of the superblock
 */
struct  filsys
{
        ushort   s_isize;           /* size in blocks of i-list */
        daddr_t  s_fsize;           /* size in blocks of entire volume
        short    s_nfree;           /* number of addresses in s_free *
        daddr_t  s_free[NICFREE];   /* free block list */
        short    s_ninode;          /* number of inodes in s_inode */
        ino_t    s_inode[NICINOD];  /* free inode list */
        char     s_flock;           /* lock during free list manipulat
        char     s_ilock;           /* lock during i-list manipulatio
        char     s_fmod;            /* superblock modified flag */
        char     s_ronly;           /* mounted read-only flag */
        time_t   s_time;            /* last superblock update */
        short    s_dinfo[4];        /* device information */
        daddr_t  s_tfree;           /* total free blocks*/
        ino_t    s_tinode;          /* total free inodes */
        char     s_fname[6];        /* file system name */
        char     s_fpack[6];        /* file system pack name */
        long     s_fill[13];        /* ADJUST size of filsys to 512 */
        long     s_magic;           /* magic number to indicate new fi
        long     s_type;            /* type of new file system */
};

#define FsMAGIC 0xfd187e20          /* s_magic number */
#define Fs1b    1                   /* 512-byte block */
#define Fs2b    2                   /* 1024-byte block */
```

S_type indicates the file system type. Currently, two types of file systems are supported: the original 512-byte oriented and the new improved 1024-byte oriented. *S_magic* is used to distinguish the original 512-byte oriented file systems from the newer file systems. If this field is not equal to the magic number, *FsMAGIC*, the type is assumed to be *Fs1b*, otherwise the *s_type* field is used. In the following description, a block is then determined by the type. For the original 512-byte oriented file system, a block is 512 bytes. For the 1024-byte oriented file system, a block is 1024 bytes or two sectors. The operating system takes care of all conversions from logical block numbers to physical sector numbers.

S_isize is the address of the first data block after the i-list; the i-list starts just after the super-block, namely in block 2; thus the i-list is s_isize-2 blocks long. S_fsize is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an ``impossible'' block number is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The s_free array contains, in s_free[1], ..., s_free[s_nfree-1], up to 49 numbers of free blocks. S_free[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement s_nfree, and the new block is s_free[s_nfree]. If the new block number is 0, there are no blocks left, so give an error. If s_nfree became 0, read in the block named by the new block number, replace s_nfree by its first word, and copy the block numbers in the next 50 longs into the s_free array. To free a block, check if s_nfree is 50; if so, copy s_nfree and the s_free array into it, write it out, and set s_nfree to 0. In any event set s_free[s_nfree] to the freed block's number and increment s_nfree.

S_tfree is the total free blocks available in the file system.

S_ninode is the number of free i-numbers in the s_inode array. To allocate an inode: if s_ninode is greater than 0, decrement it and return s_inode[s_ninode]. If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the s_inode array, then try again. To free an inode, provided s_ninode is less than 100, place its number into s_inode[s_ninode] and increment s_ninode. If s_ninode is already 100, do not bother to enter the freed inode into any table. This list of inodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

S_tinode is the total free inodes available in the file system.

S_flock and s_ilock are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of s_fmod on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

S_ronly is a read-only flag to indicate write-protection.

S_time is the last time the super-block of the file system was changed, and is the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT). During a reboot, the s_time of the super-block for the root file system is used to set the system's idea of the time.

S_fname is the name of the file system and s_fpack is the name of the pack.

I-numbers begin at 1, and the storage for inodes begins in block 2. Also, inodes are 64 bytes long. Inode 1 is reserved for future use. Inode 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each inode represents one file. For the format of an inode and its flags, see inode(4).

FILES

/usr/include/sys/filsys.h
/usr/include/sys/stat.h

SEE ALSO

fsck(1M), fsdb(1M), mkfs(1M), inode(4).

NAME

fspec - format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the UNIX System with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX System commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets <: and :>. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t_tabs_ The **t** parameter specifies the tab settings for the file. The value of _tabs_ must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;

2. a - followed immediately by an integer _n_, indicating tabs at intervals of _n_ columns;

3. a - followed by the name of a ``canned'' tab specification.

Standard tabs are specified by **t-8**, or equivalently, **t1,9,17,25**,etc. The canned tabs which are recognized are defined by the _tabs_(1) command.

s_size_ The **s** parameter specifies a maximum line size. The value of _size_ must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m_margin_ The **m** parameter specifies a number of spaces to be prepended to each line. The value of _margin_ must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t**-8 and **m**0. 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

gettydefs - speed and terminal settings used by getty

DESCRIPTION

The **/etc/gettydefs** file contains information used by _getty_(1M) to set up the speed and terminal settings for a line. It supplies information on what the _login_ prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a <_break_> character.

Each entry in **/etc/gettydefs** has the following format:

label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. 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

gps - graphical primitive string, format of graphical files

DESCRIPTION

GPS is a format used to store graphical data. Several routines have been developed to edit and display GPS files on various devices. Also, higher level graphics programs such as plot (in stat(1G)) and vtoc (in toc(1G)) produce GPS format output files.

A GPS is composed of five types of graphical data or primitives.

GPS PRIMITIVES

lines The lines primitive has a variable number of points from which zero or more connected line segments are produced. The first point given produces a move to that location. (A move is a relocation of the graphic cursor without drawing.) Successive points produce line segments from the previous point. Parameters are available to set color, weight, and style (see below).

arc The arc primitive has a variable number of points to which a curve is fit. The first point produces a move to that point. If only two points are included, a line connecting the points will result. If three points are included, a circular arc through the points is drawn. If more than three points are included, lines connect the points. (In the future, a spline will be fit to the points if they number greater than three.) Parameters are available to set color, weight, and style.

text The text primitive draws characters. It requires a single point which locates the center of the first character to be drawn. Parameters are color, font, textsize, and textangle.

hardware The hardware primitive draws hardware characters or gives control commands to a hardware device. A single point locates the beginning location of the hardware string.

comment A comment is an integer string that is included in a GPS file but causes nothing to be displayed. All GPS files begin with a comment of zero length.

GPS PARAMETERS

color Color is an integer value set for arc, lines, and text primitives.

weight *Weight* is an integer value set for *arc* and *lines* primitives to indicate line thickness. The value 0 is narrow weight, 1 is bold weight, and 2 is medium weight.

style *Style* is an integer value set for *lines* and *arc* primitives to give one of the five different line styles that can be drawn on Tektronix 4010 series storage tubes. They are:

0	solid
1	dotted
2	dot dashed
3	dashed
4	long dashed

font An integer value set for *text* primitives to designate the text font to be *used* in drawing a character string. (Currently *font* is expressed as a 4-bit *weight* value followed by a 4-bit *style* value.)

textsize *Textsize* is an integer value used in *text* primitives to express the size of the characters to be drawn. *Textsize* represents the height of characters in absolute *universe-units* and is stored at one-fifth this value in the size-orientation (*so*) word (see below).

textangle *Textangle* is a signed integer value used in *text* primitives to express rotation of the character string around the beginning point. *Textangle* is expressed in degrees from the positive x-axis and can be a positive or negative value. It is stored in the size-orientation (*so*) word as a value 256/360 of its absolute value.

ORGANIZATION

GPS primitives are organized internally as follows:

lines	*cw* *points* *sw*
arc	*cw* *points* *sw*
text	*cw* *point* *sw* *so* [*string*]
hardware	*cw* *point* [*string*]
comment	*cw* [*string*]

cw *Cw* is the control word and begins all primitives. It consists of 4 bits that contain a primitive-type code and 12 bits that contain the word-count for that primitive.

point(s) *Point(s)* is one or more pairs of integer coordinates. *Text* and *hardware* primitives only require a single *point*. *Point(s)* are values within a Cartesian plane or *universe* having 64K (-32K to +32K) points on each axis.

sw *Sw* is the style-word and is used in *lines*, *arc*, and *text* primitives. The first 8 bits contain *color* information. In *arc* and *lines* the last 8 bits are divided as 4 bits *weight* and 4 bits *style*. In the *text* primitive the last 8 bits of *sw* contain the *font*.

so *So* is the size-orientation word used in *text* primitives. The first 8 bits contain text size (see *textsize*) and the remaining 8 bits contain text rotation (see *textangle*).

string *String* is a null-terminated character string. If the string does not end on a word boundary, an additional null is added to the GPS file to assure word-boundary alignment.

SEE ALSO

graphics(1G).

NAME
group - group file

DESCRIPTION
Group contains the following information for each group:

group name
encrypted password
numerical group ID
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 directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group IDs to names.

FILES
/etc/group

SEE ALSO
newgrp(1), passwd(1), crypt(3C), passwd(4).

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 initialize devices on which *init* might try to ask the *run-level* question. These entries are executed and waited for before continuing.

process This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh -c 'exec** *command***'**. For this reason, any legal *sh* syntax can appear in the the *process* field. Comments can be inserted with the **; #***comment* syntax.

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 <sys/ino.h>
```

DESCRIPTION

An inode for a plain file or directory in a file system has the following structure defined by **<sys/ino.h>**.

```
/* Inode structure as it appears on a disk block. */
struct dinode
{
        ushort di_mode;       /* mode and type of file */
        short  di_nlink;      /* number of links to file */
        ushort di_uid;        /* owner's user id */
        ushort di_gid;        /* owner's group id */
        off_t  di_size;       /* number of bytes in file */
        char   di_addr[40];   /* disk block addresses */
        time_t di_atime;      /* time last accessed */
        time_t di_mtime;      /* time last modified */
        time_t di_ctime;      /* time created */
};
/*
 * the 40 address bytes:
 *     39 used; 13 addresses
 *     of 3 bytes each.
 */
```

For the meaning of the defined types off_t and time_t, see types(5).

FILES

/usr/include/sys/ino.h

SEE ALSO

stat(2), fs(4), types(5).

NAME

issue - issue identification file

DESCRIPTION

The file **/etc/issue** contains the _issue_ or project identification to be printed as a login prompt. This is an ASCII file which is read by _getty_(1M) and then written to any terminal spawned or respawned from the _lines_ file.

FILES

/etc/issue

SEE ALSO

getty(1M), login(1).

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).

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 initializes 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 object file. See the manual entries for the corresponding standard input/output library functions for details on the use of these macros. (The functions are identified as 3S in Section 3 of this manual.)

The program must be loaded with the object file access routine library **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

master - master device information table

DESCRIPTION

This file is used by the *config*(1M) program to obtain device information that enables it to generate the configuration files. The file consists of 3 or 4 parts, each separated by a line with a dollar sign ($) in column 1. Part 1 contains device information; part 2 contains names of devices that have aliases; part 3 contains tunable parameter information. Part 4 is optional and contains information related to configuring the M68000 family systems only. Any line with an asterisk (*) in column 1 is treated as a comment.

Part 1 contains lines consisting of at least 10 fields and at most 13 fields. The fields are delimited by tabs and/or blanks.

Field 1: device name (8 chars. maximum).

Field 2: interrupt vector size (decimal, in bytes).

Field 3: device mask (octal)-each ``on'' bit indicates that the handler exists: 000100 initialization handler 000040 power-failure handler 000020 open handler 000010 close handler 000004 read handler 000002 write handler 000001 ioctl handler

Field 4: device type indicator (octal): 000200 allow only one of these devices 000100 suppress count field in the **conf.c** file 000040 suppress interrupt vector 000020 required device 000010 block device 000004 character device 000002 interrupt driven device other than block or char. device

Field 5: handler prefix (4 chars. maximum).

Field 6: device address size (decimal).

Field 7: major device number for block-type device.

Field 8: major device number for character-type device.

Field 9: maximum number of devices per controller (decimal).

Field 10: maximum bus request level (1 through 7).

Fields 11-13: optional configuration table structure declarations (8 chars. maximum)

Part 2 contains lines with 2 fields each:

Field 1: alias name of device (8 chars. maximum).

Field 2: reference name of device (8 chars. maximum; specified in part 1).

Part 3 contains lines with 2 or 3 fields each:

Field 1: parameter name (as it appears in description file; 30 chars. maximum)

Field 2: parameter name (as it appears in the **conf.c** file; 30 chars. maximum)

Field 3: default parameter value (30 chars. maximum; parameter specification is required if this field is omitted)

Part 4 contains M68000-specific lines exactly like those for the M68000-specific portion of the **dfile.** See config (1M) for a description of these lines.

Devices that are not interrupt-driven have an interrupt vector size of zero. The 040 bit in Field 4 causes config(1M) to record the interrupt vectors although the **m68kvec.s** file will show no interrupt vector assignment at those locations (interrupts here will be treated as strays).

SEE ALSO
config(1M).

NAME

mnttab - mounted file system table

SYNOPSIS

#include <mnttab.h>

DESCRIPTION

Mnttab resides in directory **/etc** and contains a table of devices, mounted by the mount(1M) command, in the following structure as defined by **<mnttab.h>**:

```
struct  mnttab {
        char        mt_dev[10];
        char        mt_filsys[10];
        short       mt_ro_flg;
        time_t      mt_time;
};
```

Each entry is 26 bytes in length; the first 10 bytes are the null-padded name of the place where the special file is mounted; the next 10 bytes represent the null-padded root name of the mounted special file; the remaining 6 bytes contain the mounted special file's read/write permissions and the date on which it was mounted.

The maximum number of entries in mnttab is based on the system parameter **NMOUNT** located in **/usr/src/uts/cf/conf.c**, which defines the number of allowable mounted special files.

SEE ALSO

mount(1M), setmnt(1M).

NAME

passwd - password file

DESCRIPTION

Passwd contains the following information for each user:

login name
encrypted password
numerical user ID
numerical group ID
GCOS job number, box number, optional GCOS user ID
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user entry 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, the Shell itself is used.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user IDs to names.

The encrypted password consists of 13 characters chosen from a 64-character alphabet (**.**, **/**, **0-9**, **A-Z**, **a-z**). If the password is null, the encrypted password is also null. Password aging is effected for a particular user if the encrypted password in the password file is followed by a comma and a non-null string of characters from the above alphabet. Such a string must be introduced in the first instance by the superuser.

The first character of the password age, e.g., *M*, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after the password has expired will be forced to supply a new one. The next character, e.g., *m*, denotes the minimum period (in weeks) which must expire before the password may be changed. The remaining characters define the week (counted from the beginning of 1970) when the password was last changed. A null string is equivalent to zero. *M* and *m* have numerical values in the range 0-63 that correspond to the 64-character alphabet shown above (i.e., **/** = 1 week; **z** = 63 weeks). If *m* = *M* = 0 (derived from the string **.** or **..**) the password must be changed the next time the user logs in (and the \`\`age'' will disappear from the user's entry in the password file). If *m* > *M* (signified, e.g., by the string **./**), only the superuser will be able to change the password.

FILES

/etc/passwd

SEE ALSO

login(1), passwd(1), a641(3C), crypt(3C), getpwent(3C), group(4).

NAME

plot - graphics interface

DESCRIPTION

Files of this format are produced by routines described in plot(3X) and are interpreted for various devices by commands described in tplot(1G). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by 4 bytes representing the **x** and **y** values; each value is a signed integer. The last designated point in an **l**, **m**, **n**, or **p** instruction becomes the ``current point'' for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in plot(3X).

m move: The next 4 bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next 4 bytes. See tplot(1G).

p point: Plot the point given by the next 4 bytes.

l line: Draw a line from the point given by the next 4 bytes to the point given by the following 4 bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a new-line.

e erase: Start another frame of output.

f linemod: Take the following string, up to a new-line, as the style for drawing further lines. The styles are ``dotted'', ``solid'', ``longdashed'', ``shortdashed'', and ``dotdashed''. This instruction is effective only for the -T4014 and -Tver options of tplot(1G) (Tektronix 4014 terminal and Versatec plotter).

s space: The next 4 bytes give the lower left corner of the plotting area; the following 4 give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of tplot(1G). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points outside may be displayable on devices whose face is not square.

```
DASI 300          space(0, 0, 4096, 4096);
DASI 300s         space(0, 0, 4096, 4096);
DASI 450          space(0, 0, 4096, 4096);
Tektronix 4014    space(0, 0, 3120, 3120);
Versatec plotter  space(0, 0, 2048, 2048);
```

SEE ALSO

graph(1G), tplot(1G), plot(3X), gps(4), term(5).

NAME

pnch - file format for card images

DESCRIPTION

The PNCH format is a convenient representation for files consisting of card images in an arbitrary code.

A PNCH file is a simple concatenation of card records. A card record consists of a single control byte followed by a variable number of data bytes. The control byte specifies the number (which must lie in the range 0-80) of data bytes that follow. The data bytes are 8-bit codes that constitute the card image. If there are fewer than 80 data bytes, it is understood that the remainder of the card image consists of trailing blanks.

SEE ALSO

send(1C).

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; **.profiles** are 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 (except for the comments):

```
#  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 */
         short     r_type ;     /* relocation type */
} ;


/*
 * All generics
 *       reloc. already performed to symbol in the same section
 */
#define  R_ABS                  0

/*
 * DEC Processors  VAX 11/780 and VAX 11/750
 *
 */
#define R_RELBYTE  017
#define R_RELWORD  020
#define R_RELLONG  021
#define R_PCRBYTE  022
#define R_PCRWORD  023
#define R_PCRLONG  024

/*
 * Motorola 68000 uses R_RELBYTE, R_RELWORD, R_RELLONG,
 * R_PCRBYTE, and R_PCRWORD as for DEC machines above.
 */
```

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.

On the VAX processors, relocation of a symbol index of -1 indicates that the relative difference between the current segment's start address and the program's load address is added to the relocatable 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[SYMNMLEN]; /* 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 initialized, the file contains the actual bytes. An uninitialized section is somewhat different. It has a size, symbols defined in it, and symbols that refer to it, but it can have no relocation entries, line numbers, or data. Consequently, an uninitialized 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
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
        {
                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 machines in which longs are equivalent to ints (M68000 and VAX), the longs are converted to ints in the compiler to minimize the complexity of the compiler code generator. Thus, the information about which symbols are declared as longs and which as ints cannot be determined from the symbol table.

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"
#define   WTMP_FILE    "/etc/wtmp"
#define   ut_name      ut_user

struct utmp {
       char       ut_user[8];          /* User login name */
       char       ut_id[4];            /* /etc/inittab id (usually lin
       char       ut_line[12];         /* device name (console, lnxx)
       short      ut_pid;              /* process id */
       short      ut_type;             /* type of entry */
       struct     exit_status {
           short      e_termination;   /* Process termination status *
           short      e_exit;          /* Process exit status */
       } ut_exit;                      /* The exit status of a process
                                        * marked as DEAD_PROCESS. */
       time_t     ut_time;             /* time entry was made */
};

/*  Definitions for ut_type  */
#define EMPTY          0
#define RUN_LVL        1
#define BOOT_TIME      2
#define OLD_TIME       3
#define NEW_TIME       4
#define INIT_PROCESS   5           /* Process spawned by "init" */
#define LOGIN_PROCESS  6           /* A "getty" process waiting for lo
#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 facilities such as formatting documentation and setting the terminal environment. It also contains descriptions of various character set tables, flag values, and user-accessible data types.

NAME
ascii - map of ASCII character set

SYNOPSIS
cat /usr/pub/ascii

DESCRIPTION
<u>Ascii</u> is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, to be printed as needed. It contains:

000 nul	001 soh	002 stx	003 etx	004 eot	005 enq	006 ack	007 bel
010 bs	011 ht	012 nl	013 vt	014 np	015 cr	016 so	017 si
020 dle	021 dc1	022 dc2	023 dc3	024 dc4	025 nak	026 syn	027 etb
030 can	031 em	032 sub	033 esc	034 fs	035 gs	036 rs	037 us
040 sp	041 !	042 "	043 #	044 $	045 %	046 &	047 '
050 (	051)	052 *	053 +	054 ,	055 -	056 .	057 /
060 0	061 1	062 2	063 3	064 4	065 5	066 6	067 7
070 8	071 9	072 :	073 ;	074 <	075 =	076 >	077 ?
100 @	101 A	102 B	103 C	104 D	105 E	106 F	107 G
110 H	111 I	112 J	113 K	114 L	115 M	116 N	117 O
120 P	121 Q	122 R	123 S	124 T	125 U	126 V	127 W
130 X	131 Y	132 Z	133 [	134 \	135]	136 ^	137 _
140 `	141 a	142 b	143 c	144 d	145 e	146 f	147 g
150 h	151 i	152 j	153 k	154 l	155 m	156 n	157 o
160 p	161 q	162 r	163 s	164 t	165 u	166 v	167 w
170 x	171 y	172 z	173 {	174 ¦	175 }	176 ~	177 del

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0a nl	0b vt	0c np	0d cr	0e so	0f si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1a sub	1b esc	1c fs	1d gs	1e rs	1f us
20 sp	21 !	22 "	23 #	24 $	25 %	26 &	27 '
28 (	29)	2a *	2b +	2c ,	2d -	2e .	2f /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3a :	3b ;	3c <	3d =	3e >	3f ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4a J	4b K	4c L	4d M	4e N	4f O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5a Z	5b [	5c \	5d]	5e ^	5f _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6a j	6b k	6c l	6d m	6e n	6f o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7a z	7b {	7c ¦	7d }	7e ~	7f del

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), *vi*(1), and *tplot*(1G), which may exploit special capabilities of the terminal.

TZ Time zone information. The format is **xxx***n***zzz** 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), sh(1), exec(2), getenv(3C), profile(4), term(5).

NAME

fcntl - file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The _fcntl_(2) function provides for control over open files. This #include file describes _requests_ and _arguments_ to _fcntl_(2) and _open_(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR   2
#define O_NDELAY 04       /* Non-blocking I/O */
#define O_APPEND 010      /* append (writes guaranteed at the end) */

/* Flag values accessible only to open(2) */
#define O_CREAT  00400    /* open with file create (uses third open a
#define O_TRUNC  01000    /* open with truncation */
#define O_EXCL   02000    /* exclusive open */

/* fcntl(2) requests */
#define F_DUPFD  0        /* Duplicate fildes */
#define F_GETFD  1        /* Get fildes flags */
#define F_SETFD  2        /* Set fildes flags */
#define F_GETFL  3        /* Get file flags */
#define F_SETFL  4        /* Set file flags */
```

SEE ALSO

fcntl(2), open(2).

NAME

greek – graphics for the extended TTY-37 type-box

SYNOPSIS

cat /usr/pub/greek [¦ **greek** -Tterminal]

DESCRIPTION

Greek gives the mapping from ASCII to the ``shift-out'' graphics in effect between SO and SI on TELETYPE(Reg.) Model 37 terminals equipped with a 128-character type-box. These are the default greek characters produced by *nroff*. The filters of *greek*(1) attempt to print them on various other terminals. The file contains:

alpha	A	A	beta	B	B	gamma	\	\
GAMMA	G	G	delta	D	D	DELTA	W	W
epsilon	S	S	zeta	Q	Q	eta	N	N
THETA	O	T	theta	T	O	lambda	L	L
LAMBDA	E	E	mu	M	M	nu	@	@
xi	X	X	pi	J	J	PI	P	P
rho	K	K	sigma	Y	Y	SIGMA	R	R
tau	I	I	phi	U	U	PHI	F	F
psi	V	V	PSI	H	H	omega	C	C
OMEGA	Z	Z	nabla	[	[	not	–	–
partial	]	]	integral	^	^			

FILES

/usr/pub/greek

SEE ALSO

300(1), 4014(1), 450(1), greek(1), hp(1), tc(1), nroff(1).

NAME

man - macros for formatting entries in this manual

SYNOPSIS

nroff -man files

troff -man [**-rs1**] files

DESCRIPTION

These *troff*(1) macros are used to lay out the format of the entries of this manual. A skeleton entry may be found in the file **/usr/man/u_man/man0/skeleton.** These macros are used by the *man*(1) command.

The default page size is 8.5"x11", with a 6.5"x10" text area; the **-rs1** option reduces these dimensions to 6"x9" and 4.75"x8.375", respectively; this option, which is not effective in *nroff*(1), also reduces the default type size from 10-point to 9-point and the vertical line spacing from 12-point to 10-point. The **-rV2** option may be used to set certain parameters to values appropriate for certain Versatec printers: it sets the line length to 82 characters and the page length to 84 lines, and it inhibits underlining; this option should not be confused with the **-Tvp** option of the *man*(1) command, which is available at some UNIX System sites.

Any *text* argument below may be one to six ``words''. Double quotes ("") must be used to include blanks in a ``word''. If *text* is empty, the special treatment is applied to the next line that contains text to be printed. For example, **.I** may be used to italicize a whole line, or **.SM** followed by **.B** to make small bold text. By default, hyphenation is turned off for *nroff* but remains on for *troff*.

Type font and size are reset to default values before each paragraph and after processing font-setting and size-setting macros, e.g., .I, .RB, .SM. Tab stops are neither used nor set by any macro except .DT and .TH.

Default units for indents (*in*) are ens. When a macro is given without the *in* argument, the previous indent is used. The "remembered" indent is set to its default value by the .TH,.P,.SH,and .SS macros. This value is 7.2 ens in *troff* and 5 ens in *nroff*; both are equal to 0.5 inches in the default page size. This means that within each subheading section (SYNOPSIS, DESCRIPTION, etc.) the default left margin is 0.5 inches to the right of the page offset (i.e., normal left margin) of the page. If the entire page width is needed (e.g., to format a large table), use **.in** alone on a line to override the default indented margin.

Each macro description below includes the effect on indentation, as applicable.

.TH _t_ _s_ _c_ _n_	Set the title and entry heading; _t_ is the title, _s_ is the section number, _c_ is extra commentary, e.g., ``local'', _n_ is new manual name. Invokes **.DT** (see below).
.SH _text_	Place subhead _text_, e.g., **SYNOPSIS**, here. The text lines that follow the heading are block-style paragraphs; the whole block is indented 0.5 inches.
.SS _text_	Place sub-subhead _text_, e.g., **Options**, here. The text lines that follow the heading are block-style paragraphs; the whole block is indented 0.5 inches.
.B _text_	Make _text_ bold.
.I _text_	Make _text_ italic.
.SM _text_	Make _text_ 1 point smaller than default point size.
.RI _a_ _b_	Concatenate roman _a_ with italic _b_, and alternate these two fonts for up to six arguments. Similar macros alternate between any two of roman, italic, and bold: **.IR .RB .BR .IB .BI**
.P	Skip one vertical space and begin a paragraph with normal font, point size, and indent (0.5 inches). **.PP** has the same effect as **.P**.
.HP _in_	Skip one vertical space and begin a paragraph with a hanging indent. The first line of the paragraph will be indented the default 0.5 inches from the page offset. The other lines will be indented the additional number of ens specified by _in_.
.TP _in_	Skip one vertical space and begin indented paragraph with hanging tag. The next line that contains text to be printed is taken as the tag. The indentation from the beginning of the tag to the beginning of the paragraph is specified by the _in_ argument. If the tag does not fit, it is printed on a separate line. Format within the paragraph can be controlled by using the _nroff_ commands **.br** and **.nf** (refer to the _Document Processing Guide_).
.IP _t_ _in_	Same as **.TP** _in_ with tag _t_; often used to get an indented paragraph without a tag.
.RS _in_	Increase indentation relative to the current margin. If given without an argument, the text following the macro will be indented 0.5 inches. The **.RS** macro does not cause a vertical space to be inserted before the following output. Use **.sp** on the line before the **.RS** line to obtain this space. If an _in_ argument

	is given, the **.RS** macro will indent the following output *in* units from the current left margin.
.RE *k*	Return to the *k*th relative indent level (initially, *k*=1; *k*=0 is equivalent to *k*=1); if *k* is omitted, return to the most recent lower indent level. .RS/.RE pairs can be nested.
.PM *m*	Produces proprietary markings; where *m* may be **P** for **PRIVATE**, **N** for **NOTICE**, **BP** for **BELL LABORATORIES PROPRIETARY**, or **BR** for **BELL LABORATORIES RESTRICTED**.
.DT	Restore default tab settings (every 7.2 ens in *troff*, 5 ens in *nroff*).
.PD *v*	Set the interparagraph distance to *v* vertical spaces. If *v* is omitted, set the interparagraph distance to the default value (0.4v in *troff*, 1v in *nroff*).

The following *strings* are defined:

*R	``(Reg.)'' in *nroff*(1), ``Registered'' symbol in *troff*(1).
*S	Change to default type size.
*(Tm	Trademark indicator.

The following *number registers* are given default values by .TH:

IN	Left margin indent relative to subheads (default is 7.2 ens in *troff*, 5 ens in *nroff*).
LL	Line length including **IN**.
PD	Current interparagraph distance.

WARNINGS

In addition to the macros, strings, and number registers mentioned above, there are defined a number of *internal* macros, strings, and number registers. Except for names predefined by *troff* and number registers **d**, **m**, and **y**, all such internal names are of the form *XA*, where *X* is one of),], and }, and *A* stands for any alphanumeric character.

If a manual entry needs to be preprocessed by *cw*(1), *eqn*(1) (or *neqn*), and/or *tbl*(1), it must begin with a special line (described in *man*(1)), causing the *man* command to invoke the appropriate preprocessor(s).

The programs that prepare the Table of Contents and the Permuted Index for the User's Manual and Administrator's Manual assume the *NAME* section of each entry consists of a single line of input that has the following format:

```
name[, name, name ...] \- explanatory text
```

To eliminate ambiguity, the macro package increases the inter-word spaces in the SYNOPSIS section of each entry.

The macro package itself uses only the roman font (so that one can replace, for example, the bold font by the constant-width font-see *cw*(1)). Of course, if the input text of an entry contains requests for other fonts (e.g., .I, .RB, \fI), the corresponding fonts must be mounted. If a single word or short phrase needs to be italicized or emboldened, the following usage can be placed within a line, rather than creating a separate .B or .I line: \fItext\fR.

Nroff and *troff* formatting commands and macros are described in the Document Processing Guide.

FILES

/usr/lib/tmac/tmac.an
/usr/lib/macros/cmp.[nt].[dt].an
/usr/lib/macros/ucmp.[nt].an
/usr/man/[ua]_man/man0/skeleton

SEE ALSO

man(1), nroff(1), troff(1).

BUGS

When using the macros to alternate fonts (e.g., .RB, .IR), quotation marks must be used to maintain spacing. For example, .IR **filename** produces filename as one word. .IR **"file " name** produces it as two words.

NAME

mm - the MM macro package for formatting documents

SYNOPSIS

mm [options] [files]

nroff -mm [options] [files]

nroff -cm [options] [files]

mmt [options] [files]

troff -mm [options] [files]

troff -cm [options] [files]

DESCRIPTION

This package provides a formatting capability for a wide variety of documents. The manner in which a document is typed and edited is essentially independent of whether the document is to be eventually formatted at a terminal or phototypeset. See the references below for further details.

The **-mm** option causes nroff(1) and troff(1) to use the non-compacted version of the macro package, while the **-cm** option results in the use of the compacted version, thus speeding up the process of loading the macro package.

FILES

/usr/lib/tmac/tmac.m	pointer to the non-compacted version of the package
/usr/lib/macros/mm[nt]	non-compacted version of the package
/usr/lib/macros/cmp.[nt].[dt].m	compacted version of the package
/usr/lib/macros/ucmp.[nt].m	initializers for the compacted version of the package

SEE ALSO

mm(1), mmt(1), nroff(1), troff(1).
Document Processing Guide.
MM-Memorandum Macros by D. W. Smith and J. R. Mashey.
Typing Documents with MM by D. W. Smith and E. M. Piskorik.

NAME

mosd - the OSDD adapter macro package for formatting documents

SYNOPSIS

osdd [options] [files]

mm -mosd [options] [files]

nroff -mm -mosd [options] [files]

nroff -cm -mosd [options] [files]

mmt -mosd [options] [files]

troff -mm -mosd [options] [files]

troff -cm -mosd [options] [files]

DESCRIPTION

The OSDD adapter macro package is a tool used in conjunction with the *mm*(1) macro package to prepare Operations Systems Deliverable Documentation. Many of the OSDD Standards are different than the default format provided by *mm*(1). The OSDD adapter package sets the appropriate *mm*(1) options for automatic production of the OSDD Standards. The OSDD adapter package also generates the correct OSDD page headers and footers, heading styles, Table of Contents format, etc.

OSDD document (input) files are prepared with the *mm*(1) macros. Additional information which must be given at the beginning of the document file is specified by the following string definitions:

```
.ds H1 document-number
.ds H2 section-number
.ds H3 issue-number
.ds H4 date
.ds H5 rating
```

The *document-number* should be of the standard 10-character format. The words ``Section'' and ``Issue'' should not be included in the string definitions; they will be supplied automatically when the document is printed. For example,

```
.ds H1 OPA-1P135-01
.ds H2 4
.ds H3 2
```

automatically produces

```
OPA-1P135-01
Section 4
Issue 2
```

as the document page header. Quotation marks are not used in string definitions.

If certain information is not to be included in a page header, the string is defined as null; e.g.,

```
.ds H2
```

means that there is no _section-number_.

The OSDD Standards require that the _Table of Contents_ be numbered beginning with _Page 1_. By default, the first page of text will be numbered _Page 2_. If the _Table of Contents_ has more than one page, for example _n_, either -rP_n+1_ must be included as a command line option or .nr P _n_ must be included in the document file. For example, if the _Table of Contents_ is four pages, use -rP5 on the command line or .nr P 4 in the document file.

The OSDD Standards require that certain information such as the document _rating_ appear on the _Document Index_ or on the _Table of Contents_ page if there is no index. By default, it is assumed that an index has been prepared separately. If there is no index, the following must be included in the document file:

```
.nr Di 0
```

This will ensure that the necessary information is included on the _Table of Contents_ page.

The OSDD Standards require that all numbered figures be placed at the end of the document. The **.Fg** macro is used to produce full page figures. This macro produces a blank page with the appropriate header, footer, and figure caption. Insertion of the actual figure on the page is a manual operation. The macro usage is

```
.Fg page-count "figure caption"
```

where _page-count_ is the number of pages required for a multi-page figure (default 1 page).

Figure captions are produced by the **.Fg** macro using the **.BS**/**.BE** macros; therefore, the **.BS**/**.BE** macros are not available for users. The **.Fg** macro cannot be used within the document unless the final **.Fg** in a series of figures is followed by a **.SK** macro to force out the last figure page.

The _Table of Contents_ for OSDD documents (see Figure 4 in Section 4.1 of the OSDD Standards) is produced with:

```
.Tc
System Type
System Name
Document Type
.Td
```

The **.Tc**/**.Td** macros are used instead of the .TC macro from _mm_(1).

By default, the adapter package causes the **NOTICE** disclosure statement to be printed. The **.PM** macro may be used to suppress the **NOTICE** or to replace it with the **PRIVATE** disclosure statement as follows:

.PM	none printed
.PM P	PRIVATE printed
.PM N	NOTICE printed (default)

The **.P** macro is used for paragraphs. The **Np** register is set automatically to indicate the paragraph numbering style. It is very important that the **.P** macro be used correctly. All paragraphs (including those immediately following a .H macro) must use a **.P** macro. Unless there is a .P macro, there will not be a number generated for the paragraph. Similarly, the **.P** macro should not be used for text which is not a paragraph. The **.SP** macro may be appropriate for these cases, e.g., for ``paragraphs'' within a list item.

The page header format is produced automatically in accordance with the OSDD Standards. The OSDD Adapter macro package uses the **.TP** macro for this purpose. Therefore the .TP macro normally available in *mm*(1) is not available for users.

FILES

/usr/lib/tmac/tmac.osd

SEE ALSO

mm(1), mmt(1), nroff(1), troff(1), mm(5).
MM-Memorandum Macros by D. W. Smith and J. R. Mashey.
Operations Systems Deliverable Documentation Standards, June 1980.

NAME

mptx - the macro package for formatting a permuted index

SYNOPSIS

nroff -mptx [options] [files]

troff -mptx [options] [files]

DESCRIPTION

This package provides a definition for the **.xx** macro used for formatting a permuted index as produced by _ptx_(1). This package does not provide any other formatting capabilities such as headers and footers. If these or other capabilities are required, the _mptx_ macro package may be used in conjunction with the _mm(1)_ macro package. In this case, the **-mptx** option must be invoked _after_ the **-mm** call. For example:

nroff -cm -mptx _file_

or

mm -mptx _file_

FILES

/usr/lib/tmac/tmac.ptx	pointer to the non-compacted version of the package
/usr/lib/macros/ptx	non-compacted version of the package

SEE ALSO

mm(1), nroff(1), ptx(1), troff(1), mm(5).

NAME

mv - a troff macro package for typesetting viewgraphs and slides

SYNOPSIS

mvt [-a] [options] [files]

troff [-a] [-rX1] -mv [options] [files]

DESCRIPTION

This package makes it easy to typeset viewgraphs and projection slides in a variety of sizes. A few macros (briefly described below) accomplish most of the formatting tasks needed in making transparencies. All the facilities of _troff_(1), _cw_(1), _eqn_(1), and _tbl_(1) are available for more difficult tasks.

The output can be previewed on most terminals (in particular, the Tektronix 4014) and on the Versatec printer. For these two devices, specify the **-rX1** option (this option is automatically specified by the _mvt_ command when that command is invoked with the **-T4014** or **-Tvp** options; see _mmt_(1)). To preview output on other terminals, specify the **-a** option.

The available macros are:

.VS [_n_] [_i_] [_d_] Foil-start macro; foil size is to be 7''x7''; _n_ is the foil number, _i_ is the foil identification, _d_ is the date; the foil-start macro resets all parameters (e.g., indent, point size) to initial default values, except for the values of _i_ and _d_ arguments inherited from a previous foil-start macro; it also invokes the **.A** macro (see below).

The naming convention for this and the following 8 macros is that the first character of the name (**V** or **S**) distinguishes between viewgraphs and slides, while the second character indicates whether the foil is square (**S**), small wide (**w**), small high (**h**), big wide (**W**), or big high (**H**). Slides are narrower than the corresponding viewgraphs: the ratio of the longer dimension to the shorter one is larger for slides than for viewgraphs. As a result, slide foils can be used for viewgraphs, but not vice versa; on the other hand, viewgraphs can accommodate a bit more text.

.Vw [_n_] [_i_] [_d_]	Same as **.VS**, except that foil size is 7'' wide x 5'' high.
.Vh [_n_] [_i_] [_d_]	Same as **.VS**, except that foil size is 5''x7''.
.VW [_n_] [_i_] [_d_]	Same as **.VS**, except that foil size is 7''x5.4''.
.VH [_n_] [_i_] [_d_]	Same as **.VS**, except that foil size is 7''x9''.
.Sw [_n_] [_i_] [_d_]	Same as **.VS**, except that foil size is 7''x5''.
.Sh [_n_] [_i_] [_d_]	Same as **.VS**, except that foil size is 5''x7''.
.SW [_n_] [_i_] [_d_]	Same as **.VS**, except that foil size is 7''x5.4''.
.SH [_n_] [_i_] [_d_]	Same as **.VS**, except that foil size is 7''x9''.
.A [_x_]	Place text that follows at the first indentation level (left margin); the presence of _x_ suppresses the 1/2 line spacing from the preceding text.
.B [_m_ [_s_]]	Place text that follows at the second indentation level; text is preceded by a mark; _m_ is the mark (default is a large bullet); _s_ is the increment or decrement to the point size of the mark with respect to the _prevailing_ point size (default is 0); if _s_ is 100, it causes the point size of the mark to be the same as that of the _default_ mark.
.C [_m_ [_s_]]	Same as **.B**, but for the third indentation level; default mark is a dash.
.D [_m_ [_s_]]	Same as **.B**, but for the fourth indentation level; default mark is a small bullet.
.T _string_	_String_ is printed as an over-size, centered title.
.I [_in_] [_a_ [_x_]]	Change the current text indent (does not affect titles); _in_ is the indent (in inches unless dimensioned, default is 0); if _in_ is signed, it is an increment or decrement; the presence of _a_ invokes the **.A** macro (see below) and passes _x_ (if any) to it.
.S [_p_] [_l_]	Set the point size and line length; _p_ is the point size (default is ``previous''); if _p_ is 100, the point size reverts to the _initial_ default for the current foil-start macro; if _p_ is signed, it is an increment or decrement (default is 18 for **.VS**, **.VH**, and **.SH**, and 14 for the other foil-start macros); _l_ is the line length (in inches

```
                              unless dimensioned; default is 4.2''
                              for .Vh, 3.8'' for .Sh, 5'' for .SH,
                              and 6'' for the other foil-start mac-
                              ros).
.DF n f [n f ...]             Define font positions; may not appear
                              within a foil's input text (i.e., it
                              may only appear after all the input
                              text for a foil, but before the next
                              foil-start macro); n is the position of
                              font f; up to 4 ``n f'' pairs may be
                              specified; the first font named becomes
                              the prevailing font; the initial set-
                              ting is (H is a synonym for G):
                                   .DF 1 H 2 I 3 B 4 S
.DV [a] [b] [c] [d]           Alter the vertical spacing between
                              indentation levels; a is the spacing
                              for .A, b is for .B, c is for .C, and d
                              is for .D; all non-null arguments must
                              be dimensioned; null arguments leave
                              the corresponding spacing unaffected;
                              initial setting is:
                                   .DV .5v .5v .5v 0v
.U str1 [str2]                Underline str1 and concatenate str2 (if
                              any) to it.
```

The last 4 macros in the above list do not cause a break; the .I macro causes a break only if it is invoked with more than one argument; all the other macros cause a break.

The macro package also recognizes the following upper-case synonyms for the corresponding lower-case *troff* requests:
.AD .BR .CE .FI .HY .NA .NF .NH .NX .SO .SP .TA .TI

The **Tm** string produces the trademark symbol.

The input tilde (~) character is translated into a blank on output.

See the references cited below for further details.

FILES

/usr/lib/tmac/tmac.v
/usr/lib/macros/vmca

SEE ALSO

cw(1), eqn(1), mmt(1), tbl(1), troff(1).
Document Processing Guide.
A Macro Package for View Graphs and Slides by T. A. Dolotta and D. W. Smith.

BUGS

The .VW and .SW foils are meant to be 9'' wide by 7'' high, but because the typesetter paper is generally only 8'' wide, they are printed 7'' wide by 5.4'' high and have to be enlarged by a factor of 9/7 before use as viewgraphs; this makes them less useful.

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

This page describes 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 \(.
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 initializations. Most often it is used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for GETC(), PEEKC() and UNGETC(). Otherwise it can be used to declare external variables that might be used by GETC(), PEEKC() and UNGETC(). See the example below of the declarations taken from _grep_(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function _step_. The call to _step_ is as follows:

```
step(string, expbuf)
```

The first parameter to _step_ is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter _expbuf_ is the compiled regular expression which was obtained by a call of the function _compile_.

The function _step_ returns one, 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 of how the regular expression macros and calls look from _grep_(1):

```
#define INIT   register char *sp = instring;
#define GETC() (*sp++)
#define PEEKC()      (*sp)
#define UNGETC(c)    (--sp)
#define RETURN(c)    return;
#define ERROR(c)     regerr()

#include <regexp.h>
...
        compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
        if(step(linebuf, expbuf))
                succeed();
```

FILES

/usr/include/regexp.h

SEE ALSO

ed(1), grep(1), sed(1).

BUGS

The routine _ecmp_ is equivalent to the Standard I/O routine _strncmp_ and should be replaced by that routine.

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**:

1520	Datamedia 1520
155	Motorola EXORterm 155
1620	Diablo 1620 and others using the HyType II printer
1620-12	same, in 12-pitch mode
165	Motorola EXORset 165
2621	Hewlett-Packard HP2621 series
2631	Hewlett-Packard 2631 line printer
2631-c	Hewlett-Packard 2631 line printer - compressed mode
2631-e	Hewlett-Packard 2631 line printer - expanded mode
2640	Hewlett-Packard HP2640 series
2645	Hewlett-Packard HP264n series (other than the 2640 series)
300	DASI/DTC/GSI 300 and others using the HyType I printer
300-12	same, in 12-pitch mode
300s	DASI/DTC/GSI 300s
382	DTC 382
300s-12	same, in 12-pitch mode
3045	Datamedia 3045
33	TELETYPE(Reg.) Terminal Model 33 KSR
37	TELETYPE Terminal Model 37 KSR
40-2	TELETYPE Terminal Model 40/2
40-4	TELETYPE Terminal Model 40/4
4540	TELETYPE Terminal Model 4540
3270	IBM Model 3270
4000a	Trendata 4000a
4014	Tektronix 4014
43	TELETYPE Model 43 KSR
450	DASI 450 (same as Diablo 1620)
450-12	same, in 12-pitch mode
735	Texas Instruments TI735 and TI725
745	Texas Instruments TI745
dumb	generic name for terminals that lack reverse line-feed and other special escape sequences
sync	generic name for synchronous TELETYPE 4540-compatible terminals
hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer
tn1200	General Electric TermiNet 1200
tn300	General Electric TermiNet 300
tvi950	TeleVideo 950

Local changes to this list are common. Refer to **/etc/termcap** for information on terminals supported for your system.

Up to 8 characters, chosen from [-a-z0-9], 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 behavior depends on the type of terminal should accept arguments of the form -T*term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable **$TERM**, which, in turn, should contain *term*.

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

stat - data returned by stat system call

SYNOPSIS

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

DESCRIPTION

The system calls _stat_ 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;
         ushort    st_uid;
         ushort    st_gid;
         dev_t     st_rdev;
         off_t     st_size;
         time_t    st_atime;
         time_t    st_mtime;
         time_t    st_ctime;
};

#define S_IFMT   0170000 /* type of file */
#define S_IFDIR  0040000 /* directory */
#define S_IFCHR  0020000 /* character special */
#define S_IFBLK  0060000 /* block special */
#define S_IFREG  0100000 /* regular */
#define S_IFIFO  0010000 /* fifo */
#define S_ISUID  04000   /* set user id on execution */
#define S_ISGID  02000   /* set group id on execution */
#define S_ISVTX  01000   /* save swapped text even after use */
#define S_IREAD  00400   /* read permission, owner */
#define S_IWRITE 00200   /* write permission, owner */
#define S_IEXEC  00100   /* execute/search permission, owner */
```

FILES

/usr/include/sys/types.h
/usr/include/sys/stat.h

SEE ALSO

stat(2), types(5).

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  struct { int r[1]; } *  physadr;
typedef  long             daddr_t;
typedef  char *           caddr_t;
typedef  unsigned int     uint;
typedef  unsigned short   ushort;
typedef  ushort           ino_t;
typedef  short            cnt_t;
typedef  long             time_t;
typedef  int              label_t[10];
typedef  short            dev_t;
typedef  long             off_t;
typedef  long             paddr_t;
typedef  long             key_t;
```

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

termcap - terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base which describes terminals. Each entry in the file gives a set of capabilities for a terminal and describes how operations are performed. Padding requirements and initialization sequences are included in *termcap*. The data base is used by programs such as *vi*(1).

Entries in *termcap* consist of a number of `:' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by `|' characters. The first name is always 2 characters long and is used by older systems which store the terminal type in a 16-bit word in a systemwide data base. The second name is the most common abbreviation for the terminal and the last name should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

Preparing Descriptions

The most effective way to prepare a terminal description is to imitate the description of a similar terminal in *termcap* and build up a description gradually, using partial descriptions with *ex* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *termcap* file to describe it or bugs in *ex*. To easily test a new terminal description, set the environment variable TERMCAP to a pathname of a file containing the description being worked on; the editor will look there rather than in **/etc/termcap.** TERMCAP can also be set to the *termcap* entry itself to avoid reading the file when starting up the editor.

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 tc can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since *termlib* routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be cancelled with **xx@** where xx is the capability. For example, the entry

```
hn|2621nl:ks@:ke@:tc=2621:
```

defines a 2621nl that does not have the **ks** or **ke** 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.

CAPABILITIES

Capabilities in termcap are of three types: Boolean capabilities, which indicate that the terminal has some particular feature; numeric capabilities, which give 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.

Entries may be continued onto multiple lines by giving a \ as the last character of a line. Empty fields may be included for readability (e.g., between the last field on a line and the first field on the next).

List of Capabilities

(P) indicates padding may be specified
(P*) indicates that padding may be based on no. lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace character, if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal settab
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays sa
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)

dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give :ei=: if **ic**
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no **cm**)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing **is**
im	bool		Insert mode (enter); give :im=: if **ic**
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by other function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of keypad transmit mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of other keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in keypad transmit mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on other function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no **cm**)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
ms	bool		Safe to move while in standout and underline mode
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,H200
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with **is**)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use **cm**

ti	str	String to begin programs that use **cm**
uc	str	Underscore one char and move past it
ue	str	End underscore mode
ug	num	Number of blank chars left by us or ue
ul	bool	Terminal underlines even though it doesn't overst
up	str	Upline (cursor up)
us	str	Start underscore mode
vb	str	Visible bell (may not move cursor)
ve	str	Sequence to end open/visual mode
vs	str	Sequence to start open/visual mode
xb	bool	Beehive (f1=escape, f2=ctrl C)
xn	bool	A newline is ignored after a wrap (Concept)
xr	bool	Return acts like **ce** \r \n (Delta Data)
xs	bool	Standout not erased by writing over it (HP 264?)
xt	bool	Tabs are destructive, magic so char (Teleray 1061

A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the termcap file as of this writing. (This particular concept entry is outdated, and is used as an example only.)

```
c1|c100|concept100:is=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200:\
        :al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :c
        :dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:li#2   i
        :se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Capability Descriptions

All capabilities have 2-letter codes. For instance, the fact that the Concept-100 has automatic margins (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am** in the sample description above. Numeric capabilities are followed by the character `#' and then the value. Thus, **co**, which indicates the number of columns the terminal has, gives the value `80' for the Concept-100.

String-valued capabilities, such as **ce** (clear to end of line sequence), are given by the 2-character code, an `=', and a string ending at the next field separator (:). A delay in milliseconds may appear after the `=' in such a capability and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either an integer, e.g., `20', or an integer followed by an `*', i.e., `3*'. An `*' 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. When an `*' 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.

A number of escape sequences are provided in the string-valued capabilities for easy encoding of characters there. A \E maps to an ESCAPE character, **^x** maps to a control-x for any appropriate x, and the sequences \n \r \t \b \f give a newline, return, tab, backspace and formfeed. Finally, characters may be given as 3 octal digits after a \, and the characters ^ and \ may be given as \^ and \\. If it is necessary to place a colon (:) in a capability, it must be escaped in octal as \072. If it is necessary to place a null character in a string capability, it must be encoded as \200. The routines which deal with termcap use C strings, and strip the high bits of the output very late; therefore, a \200 comes out as a \000 would.

Basic capabilities

The number of columns on each line for the terminal is given by the **co** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **li** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, its description should include the **am** capability. If the terminal can clear its screen, this is given by the **cl** string capability. If the terminal can backspace, it should have the **bs** capability, unless a backspace is accomplished by a character other than ^H, in which case the alternate character should be given as the **bc** string capability. If it overstrikes (rather than clearing a position when a character is struck over), it should have the **os** capability.

A very important point is that the local cursor motions encoded in termcap are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the **am** capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch-selectable automatic margins, the termcap file usually assumes that this is on, i.e., **am**.

These capabilities suffice to describe hardcopy and glass-tty terminals. Thus, the model 33 teletype is described as

```
t3|33|tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as

```
cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80
```

Cursor addressing

Cursor addressing in the terminal is described by the **cm** string capability. It uses escapes like those in printf(3s), i.e., **%x**. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the **cm** string is thought of as being a function, then its arguments are the line and column to which motion is desired. The **%** encodings have the following meanings:

%d	as in printf, 0 origin
%2	like %2d
%3	like %3d
%.	like %c
%+x	adds x to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %
%n	exclusive or row and column with 0140 (DM2500)
%B	BCD (16*(x/10)) + (x%10), no output.
%D	Reverse coding (x-2*(x%16)), no output. (Delta Data).

For example, to get to row 3 and column 12 the HP2645 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 2 digits. Thus, its **cm** capability is cm=6\E&%r%2c%2Y. The Microterm ACT-IV needs the current row and column sent, preceded by a ^T, with the row and column simply encoded in binary, cm=^T%.%.. Terminals which use **%.** need to be able to backspace the cursor (**bs** or bc), and to move the cursor up one line on the screen (**up** is introduced below). This is necessary because it is not always safe to transmit \t, \n ^D and **\r**, because the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character; thus, cm=\E=%+ %+ .

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, this sequence should be given as **nd** (non-destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as **up**. If the terminal has no cursor addressing capability, but can home the cursor (to the very upper left corner of screen), this can be given as **ho**; similarly, a fast way of getting to the lower left hand corner can be given as **ll**; this may involve moving up with **up** from the home position, but the editor will never do this itself (unless **ll** does) because it makes no assumption about the effect of moving up from the home position.

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 **ce**. If the terminal can clear from the current position to the end of the display, then this should be given as **cd**. The editor only uses **cd** from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **al**; 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, this should be given as **dl**; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, this can be given as **sb**, although just **al** suffices. If the terminal can retain display memory above, the **da** capability should be given; if display memory can be retained below, **db** should be given. These capabilities let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with **sb** may bring down non-blank lines.

Insert/delete character

Termcap can be used to describe two basic kinds of intelligent terminals with respect to insert/delete characters. 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; the blank is either eliminated or expanded to 2 untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type abc def using local cursor motions (not spaces) between the abc and the def. Then position the cursor before the abc and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the abc shifts over to the def which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for insert null. If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals with an insert mode that does not fall into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) To specify **im**, give the sequence to get into insert mode or give an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode If you gave **im** with an empty value, give **ei** with an empty value also. Now give as **ic** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ic**; terminals which send a sequence to open a screen position should give it here. 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**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify delete mode by giving **dm** and **ed** to enter and exit delete mode; give **dc** to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as **so** and **se** respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining - half bright is not usually an acceptable standout mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves 1 or even 2 blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **ug** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **us** and **ue**, 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**. If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.

Many 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.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), this can be given as **vb**; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of *ex*, this can be given as **vs** and **ve**, sent at the start and end of these modes, respectively. These can be used to change, e.g., from an underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as **ti** and **te**. This need arises, for example, from terminals like the Concept-100 with more than one page of memory. If the terminal has only memory-relative cursor addressing and not screen relative cursor addressing, a 1-screen sized window must be fixed into the terminal for cursor addressing to work properly.

If the terminal correctly generates underlined characters (with no special codes needed), even though it does not overstrike, you should give the capability **ul**. If overstrikes are erasable with a blank, this should be indicated 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 **ks** and **ke**; 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 **kl**, **kr**, **ku**, **kd**, and **kh**, respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0**, **k1**, **...**, **k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0**, **l1**, **...**, **l9**. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the *termcap* 2-letter codes can be given in the **ko** capability. For example, :ko=cl,ll,sf,sb: says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the **cl**, **ll**, **sf**, and **sb** entries.

The **ma** entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with **kl**, **kr**, **ku**, **kd**, and **kh**. It consists of groups of 2 characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and H for **kh**. For example, the Mime would be **:ma=^Kj^Zk^Xl:**, indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the Mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, this can be given as **pc**.

If tabs on the terminal require padding, or if the terminal uses a character other than ^I to tab, this can be given as **ta**.

Hazeltine terminals, which don't allow \`~' characters to be printed, should indicate **hz**. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed, should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **x**x.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, **is** will be printed before **if**. This is useful where **if** is /usr/lib/tabset/std but **is** clears the tabs first.

NOTE

Termcap is based on software developed by The University of California, Berkeley, California, Computer Science Division, Department of Electrical Engineering and Computer Science.

Termcap will be replaced by terminfo in the next release. Transition tools will be provided.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), termcap(3), vi(1)

WARNINGS AND BUGS

Ex allows only 256 characters for string capabilities, and the routines in *termcap(3)* do not check for overflow of this buffer. The total length of a single entry (excluding only escaped new-lines) may not exceed 1,024.

The **ma**, **vs**, and **ve** entries are specific to the *vi* program.

Not all programs support all entries. There are entries that are not supported by any program.