UNIX™ TIME-SHARING SYSTEM:

UNIX PROGRAMMER'S MANUAL

Seventh Edition, Volume 1

January, 1979

First PLEXUS Edition

Sections 2-8

September, 1981

Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

NAME

intro, errno — introduction to system calls and error numbers

SYNOPSIS

```
#include <errno.h>
```

DESCRIPTION

Section 2 of this manual lists all the entries into the system. Most of these calls have an error return. An error condition is indicated by an otherwise impossible returned value. Almost always this is -1; the individual sections 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 occurred.

There is a table of messages associated with each error, and a routine for printing the message; See **perror(3)**. The possible error numbers are not recited with each writeup in section 2, since many errors are possible for most of the calls. Here is a list of the error numbers, their names as defined in <errno.h>, and the messages available using `perror`.

0 Error 0
Unused.

1 EPERM Not owner
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

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

3 ESRCH No such process
The process whose number was given to `signal` and `ptrace` does not exist, or is already dead.

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 occurred during a `read` or `write`. 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 that does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialled in or no disk pack is loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 5120 bytes is presented to `exec`.

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(5)**.

9 EBADF Bad file number
Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file that is open only for writing (resp. reading).

10 ECHILD No children
`Wait` and the process has no living or unwaited-for children.

11 EAGAIN No more processes
In a `fork`, the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough core
During an `exec` or `break`, a program asks for more core than the system is able to supply. This is not a temporary condition; the maximum core size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers.

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 access the arguments of a system call.

15 ENOTBLK Block device required
A plain file was mentioned where a block device was required, e.g. in `mount`.

16 EBUSY Mount device busy
An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment).

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

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

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

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path name or as an argument to `chdir`.

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

22 EINVAL Invalid argument
Some invalid argument: dismounting a non-mounted device, mentioning an unknown signal in `signal`, reading or writing a file for which `seek` has generated a negative pointer. Also set by math functions, see **intro(3)**.

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

24 EMFILE Too many open files
Customary configuration limit is 20 per process.

25 ENOTTY Not a typewriter
The file mentioned in `stty` or `gtty` is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program that is currently open for writing (or reading!). Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum (about 10^9 bytes).

28 ENOSPC No space left on device
During a `write` to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An `lseek` was issued to a pipe. This error should also be issued for other non-seekable devices.

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

31 EMLINK Too many links

An attempt to make more than 32767 links to a file.

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

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

34 ERANGE Result too large
The value of a function in the math package (3M) is unrepresentable within machine precision.

SEE ALSO

intro(3)

ASSEMBLER

as `/usr/include/sys.s` *file* ...

The PDP11 assembly language interface is given for each system call. The assembler symbols are defined in `/usr/include/sys.s`.

Return values appear in registers `r0` and `r1`. It is unwise to count on these registers being preserved when no value is expected. An erroneous call is always indicated by turning on the c-bit of the condition codes. The error number is returned in `r0`. The presence of an error is most easily tested by the instructions `bes` and `bec` ('branch on error set (or clear)'). These are synonyms for the `bcs` and `bcc` instructions.

The calling conventions for the PLEXUS Z8000 implementation are very similar to those for the PDP-11. Although registers 4-7 are used for returning values from C routines, registers 0 and 1 are used in passing data to and from the system, in identical fashion to the way in which they are used by the PDP-11.

NAME

access — determine accessibility of file

SYNOPSIS

```
access(name, mode)
char *name;
```

DESCRIPTION

Access checks the given file *name* for accessibility according to *mode*, which is 4 (read), 2 (write) or 1 (execute) or a combination thereof. Specifying mode 0 tests whether the directories leading to the file can be searched and the file exists.

An appropriate error indication is returned if *name* cannot be found or if any of the desired access modes would not be granted. On disallowed accesses -1 is returned and the error code is in **errno**. 0 is returned from successful tests.

The user and group IDs with respect to which permission is checked are the real UID and GID of the process, so this call is useful to set-UID programs.

Notice that it is only access bits that are checked. A directory may be announced as writable by access, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but exec will fail unless it is in proper format.

SEE ALSO

stat(2)

ASSEMBLER

```
(access = 33.)
sys access; name; mode
```

NAME

acct — turn accounting on or off

SYNOPSIS

```
acct(file)
char *file;
```

DESCRIPTION

The system is prepared to write a record in an accounting *file* for each process as it terminates. This call, with a null-terminated string naming an existing file as argument, turns on accounting; records for each terminating process are appended to *file*. An argument of 0 causes accounting to be turned off.

The accounting file format is given in **acct(5)**.

SEE ALSO

acct(5), sa(1)

DIAGNOSTICS

On error -1 is returned. The file must exist and the call may be exercised only by the super-user. It is erroneous to try to turn on accounting when it is already on.

BUGS

No accounting is produced for programs running when a crash occurs. In particular nonterminating programs are never accounted for.

ASSEMBLER

```
(acct = 51.)
sys acct; file
```

NAME

alarm — schedule signal after specified time

SYNOPSIS

```
alarm(seconds)
unsigned seconds;
```

DESCRIPTION

Alarm causes signal SIGALRM, see **signal(2)**, to be sent to the invoking process in a number of seconds given by the argument. Unless caught or ignored, the signal terminates the process.

Alarm requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any alarm request is cancelled. Because the clock has a 1-second resolution, the signal may occur up to one second early; because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 65535 seconds.

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

pause(2), signal(2), sleep(3)

ASSEMBLER

```
(alarm = 27.)
(seconds in r0)
sys alarm
(previous amount in r0)
```

NAME

brk, sbrk, break — change core allocation

SYNOPSIS

```
char *brk(addr)
char *sbrk(incr)
```

DESCRIPTION

`brk` sets the system's idea of the lowest location not used by the program (called the break) to `addr` (rounded up to the next multiple of 64 bytes on the PDP11, 256 bytes on the Interdata 8/32, 512 bytes on the VAX-11/780). Locations not less than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

In the alternate function `sbrk`, *incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution via `exec` the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use `break`.

SEE ALSO

exec(2), malloc(3), end(3)

DIAGNOSTICS

Zero is returned if the break could be set; -1 if the program requests more memory than the system limit or if too many segmentation registers would be required to implement the break.

BUGS

Setting the break in the range 0177701 to 0177777 (on the PDP11) is the same as setting it to zero.

ASSEMBLER

```
(break = 17.)
sys break; addr
```

break performs the function of brk. The name of the routine differs from that in C for historical reasons.

NAME

brkpt — emulation of PDP11 break point instruction

SYNOPSIS

sys 255
(not available from C)

DESCRIPTION

brkpt causes a SIGTRAP to occur for the process. This is mostly useful to ADB for setting break points.

SEE ALSO

signal(2)

ASSEMBLER

sys 255

NAME

chdir, chroot — change default directory

SYNOPSIS

```
chdir(dirname)
char *dirname;

chroot(dirname)
char *dirname;
```

DESCRIPTION

Dirname is the address of the pathname of a directory, terminated by a null byte. `chdir` causes this directory to become the current working directory, the starting point for path names not beginning with "/".

`chroot` sets the root directory, the starting point for path names beginning with "/". The call is restricted to the super-user.

SEE ALSO

cd(1)

DIAGNOSTICS

Zero is returned if the directory is changed; -1 is returned if the given name is not that of a directory or is not searchable.

ASSEMBLER

```
(chdir = 12.)
sys chdir; dirname

(chroot = 61.)
sys chroot; dirname
```

NAME

chmod — change mode of file

SYNOPSIS

chmod(*name, mode*)
char **name*;

DESCRIPTION

The file whose name is given as the null-terminated string pointed to by *name* has its mode changed to *mode*. Modes are constructed by ORing together some combination of the following:

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 (search on directory) by owner
00070	read, write, execute (search) by group
00007	read, write, execute (search) by others

If an executable file is set up for sharing (`-n` or `-i` option of ld(1)), then mode 1000 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. Ability to set this bit is restricted to the super-user since swap space is consumed by the images; it is only worth while for heavily used commands.

Only the owner of a file (or the super-user) may change the mode. Only the super-user can set the 1000 mode.

SEE ALSO

chmod(1)

DIAGNOSTICS

Zero is returned if the mode is changed; -1 is returned if *name* cannot be found or if current user is neither the owner of the file nor the super-user.

ASSEMBLER

```
(chmod = 15.)
sys chmod; name; mode
```

NAME

chown — change owner and group of a file

SYNOPSIS

```
chown(name, owner, group)
char *name;
```

DESCRIPTION

The file whose name is given by the null-terminated string pointed to by *name* has its *owner* and *group* changed as specified. Only the super-user may execute this call, because if users were able to give files away, they could defeat the (nonexistent) file-space accounting procedures.

SEE ALSO

chown(1), passwd(5)

DIAGNOSTICS

Zero is returned if the owner is changed.
-1 is returned on illegal owner changes.

ASSEMBLER

```
(chown = 16.)
sys chown; name; owner; group
```

NAME

close — close a file

SYNOPSIS

close(*fildes*)

DESCRIPTION

Given a file descriptor such as returned from an **open**, **creat**, **dup** or **pipe(2)** call, close closes the associated file. A close of all files is automatic on *exit*, but since there is a limit on the number of open files per process, close is necessary for programs which deal with many files.

Files are closed upon termination of a process, and certain file descriptors may be closed by **exec(2)** (see **ioctl(2)**).

SEE ALSO

creat(2), open(2), pipe(2), exec(2), ioctl(2)

DIAGNOSTICS

Zero is returned if a file is closed; -1 is returned for an unknown file descriptor.

ASSEMBLER

```
(close = 6.)
(file descriptor in r0)
sys close
```

NAME

creat — create a new file

SYNOPSIS

```
creat(name, mode)
char *name;
```

DESCRIPTION

creat creates a new file or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode*, as modified by the process's mode mask (see **umask(2)**). Also see **chmod(2)** for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing, and its file descriptor is returned.

The *mode* given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a creat, an error is returned and the program knows that the name is unusable for the moment.

SEE ALSO

write(2), close(2), chmod(2), umask (2)

DIAGNOSTICS

The value -1 is returned if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is a directory; there are already too many files open.

ASSEMBLER

```
(creat = 8.)
sys creat; name; mode
(file descriptor in r0)
```

NAME

dup, dup2 — duplicate an open file descriptor

SYNOPSIS

```
dup(fildes)
int fildes;

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

DESCRIPTION

Given a file descriptor returned from an **open**, **pipe**, or **creat** call, dup allocates another file descriptor synonymous with the original. The new file descriptor is returned.

In the second form of the call, *fildes* is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than the maximum value allowed for file descriptors (approximately 19). dup2 causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

SEE ALSO

creat(2), open(2), close(2), pipe(2)

DIAGNOSTICS

The value -1 is returned if: the given file descriptor is invalid; there are already too many open files.

ASSEMBLER

```
(dup = 41.)
(file descriptor in r0)
(new file descriptor in r1)
sys dup
(file descriptor in r0)
```

The dup2 entry is implemented by adding 0100 to *fildes*.

NAME

execl, execv, execle, execve, execlp, execvp, exec, exece, environ — execute a file

SYNOPSIS

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

execv(name, argv)
char *name, *argv[ ];

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

execve(name, argv, envp);
char *name, *argv[ ], *envp[ ];

extern char **environ;
```

DESCRIPTION

exec in all its forms overlays the calling process with the named file, then transfers to the entry point of the core image of the file. There can be no return from a successful exec; the calling core image is lost.

Files remain open across **exec** unless explicit arrangement has been made; see **ioctl(2)**. Ignored signals remain ignored across these calls, but signals that are caught (see **signal(2)**) are reset to their default values.

Each user has a *real* user ID and group ID and an *effective* user ID and group ID. The real ID identifies the person using the system; the effective ID determines his access privileges. **exec** changes the effective user and group ID to the owner of the executed file if the file has the "set-user-ID" or "set-group-ID" modes. The real user ID is not affected.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg0, arg1* ... address null-terminated strings. Conventionally *arg0* is the name of the file.

From C, two interfaces are available. **execl** is useful when a known file with known arguments is being called; the arguments to **execl** are the character strings constituting the file and the arguments; the first argument is conventionally the

same as the file name (or its last component). A 0 argument must end the argument list.

The `execv` version is useful when the number of arguments is unknown in advance; the arguments to `execv` are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

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.

argv is directly usable in another `execv` because *argv*[*argc*] is 0.

envp is a pointer to an array of strings that constitute the environment of the process. Each string consists of a name, an "=", and a null-terminated value. The array of pointers is terminated by a null pointer. The shell **sh(1)** passes an environment entry for each global shell variable defined when the program is called. See **environ(5)** for some conventionally used names. The C run-time start-off routine places a copy of `envp` in the global cell `environ`, which is used by `execv` and `execl` to pass the environment to any subprograms executed by the current program. The `exec` routines use lower-level routines as follows to pass an environment explicitly:

```
execle( file, arg0, arg1, . . . , argn, 0, environ);
execve( file, argv, environ);
```

`execlp` and `execvp` are called with the same arguments as `execl` and `execv`, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

FILES

`/bin/sh` shell, invoked if command file found by `execlp` or `execvp`

SEE ALSO

fork(2), environ(5)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not start with a valid magic number (see **a.out(5)**), if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is *-1*. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

BUGS

If `execvp` is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of `argv[0]` and `argv[-1]` will be modified before return.

ASSEMBLER

```
(exec = 11.)
sys exec; name; argv

(exece = 59.)
sys exece; name; argv; envp
```

Plain `exec` is obsoleted by `exece`, but remains for historical reasons.

When the called file starts execution on the PDP11, the stack pointer points to a word containing the number of arguments. Just above this number is a list of pointers to the argument strings, followed by a null pointer, followed by the pointers to the environment strings and then another null pointer. The strings themselves follow; a 0 word is left at the very top of memory.

```
sp-> nargs
       arg0
       ...
       argn
       0
       env0
       ...
       envm
       0
arg0: <arg0\0>
       ...
env0: <env0\0>
       0
```

On the Interdata 8/32, the stack begins at a conventional place (currently 0xD0000) and grows upwards. After exec, the layout of data on the stack is as follows.

```
        int 0
arg0:   byte ...
        ...
argp0:  int arg0
        ...
        int 0
envp0:  int env0
        ...
        int 0
%2->    space 40
        int nargs
        int argp0
        int envp0
%3->
```

This arrangement happens to conform well to C calling conventions.

NAME

exit — terminate process

SYNOPSIS

```
exit(status)
int status;

_exit(status)
int status;
```

DESCRIPTION

exit is the normal means of terminating a process. `exit` closes all the process's files and notifies the parent process if it is executing a `wait`. The low-order 8 bits of *status* are available to the parent process.

This call can never return.

The C function `exit` may cause cleanup actions before the final "`sys exit`". The function `_exit` circumvents all cleanup.

SEE ALSO

wait(2)

ASSEMBLER

```
(exit = 1.)
(status in r0)
sys exit
```

NAME

fork — spawn new process

SYNOPSIS

```
fork( )
```

DESCRIPTION

`fork` is the only way new processes are created. The new process's core image is a copy of that of the caller of `fork`. The only distinction is the fact that the value returned in the old (parent) process contains the process ID of the new (child) process, while the value returned in the child is 0. Process ID's range from 1 to 30,000. This process ID is used by **wait(2)**.

Files open before the fork are shared, and have a common read-write pointer. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

SEE ALSO

wait(2), exec(2)

DIAGNOSTICS

Returns -1 and fails to create a process if: there is inadequate swap space, the user is not super-user and has too many processes, or the system's process table is full. Only the super-user can take the last process-table slot.

ASSEMBLER

```
(fork = 2.)
sys fork
(new process return)
(old process return, new process ID in r0)
```

The return locations in the old and new process differ by one word. The C-bit is set in the old process if a new process could not be created.

NAME

getpid — get process identification

SYNOPSIS

```
getpid( )
```

DESCRIPTION

`getpid` returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

SEE ALSO

mktemp(3)

ASSEMBLER

```
(getpid = 20.)
sys getpid
(pid in r0)
```

NAME

getuid, getgid, geteuid, getegid — get user and group identity

SYNOPSIS

```
getuid( )
geteuid( )
getgid( )
getegid( )
```

DESCRIPTION

`getuid` returns the real user ID of the current process, `geteuid` the effective user ID. The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the "set user ID" mode, to find out who invoked them.

`getgid` returns the real group ID, `getegid` the effective group ID.

SEE ALSO

setuid(2)

ASSEMBLER

```
(getuid = 24.)
sys getuid
(real user ID in r0, effective user ID in r1)

(getgid = 47.)
sys getgid
(real group ID in r0, effective group ID in r1)
```

NAME

indir — indirect system call

ASSEMBLER

```
(indir = 0.)
sys indir; call
```

The system call at the location `call` is executed. Execution resumes after the `indir` call.

The main purpose of `indir` is to allow a program to store arguments in system calls and execute them out of line in the data segment. This preserves the purity of the text segment.

If `indir` is executed indirectly, it is a no-op. If the instruction at the indirect location is not a system call, `indir` returns error code EINVAL; see **intro(2)**.

NAME

kill — send signal to a process

SYNOPSIS

`kill(`*pid, sig*`);`

DESCRIPTION

`kill` sends the signal *sig* to the process specified by the process number in tt r0. See **signal(2)** for a list of signals.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user.

If the process number is 0, the signal is sent to all other processes in the sender's process group; see **tty(4)**. there is nothing at tty(4) about process group or 0 in sorage

If the process number is -1, and the user is the super-user, the signal is broadcast universally except to processes 0 and 1, the scheduler and initialization processes, see **init(8)**.

Processes may send signals to themselves.

SEE ALSO

signal(2), kill(1)

DIAGNOSTICS

Zero is returned if the process is killed; -1 is returned if the process does not have the same effective user ID and the user is not super-user, or if the process does not exist.

ASSEMBLER

```
(kill = 37.)
(process number in tt r0)
sys kill; sig
```

NAME

link — link to a file

SYNOPSIS

```
link(name1, name2)
char *name1, *name2;
```

DESCRIPTION

A link to *name1* is created; the link has the name *name2*. Either name may be an arbitrary path name.

SEE ALSO

ln(1), unlink(2)

DIAGNOSTICS

Zero is returned when a link is made; -1 is returned when *name1* cannot be found; when *name2* already exists; when the directory of *name2* cannot be written; when an attempt is made to link to a directory by a user other than the superuser; when an attempt is made to link to a file on another file system; when a file has too many links.

ASSEMBLER

```
(link = 9.)
sys link; name1; name2
```

NAME

lock — lock a process in primary memory

SYNOPSIS

`lock(`*flag*`)`

DESCRIPTION

If the *flag* argument is non-zero, the process executing this call will not be swapped except if it is required to grow. If the argument is zero, the process is unlocked. This call may only be executed by the super-user.

BUGS

locked processes interfere with the compaction of primary memory and can cause deadlock. This system call is not considered a permanent part of the system.

ASSEMBLER

```
(lock = 53.)
sys lock; flag
```

NAME

lseek, tell — move read/write pointer

SYNOPSIS

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

long tell(fildes)
```

DESCRIPTION

The file descriptor refers to a file open for reading or writing. The read (resp. write) pointer for the file is set 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*.

The returned value is the resulting pointer location.

The obsolete function `tell(`*fildes*`)` is identical to

`lseek(`*fildes*`, 0L, 1)`.

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

SEE ALSO

open(2), creat(2), fseek(3)

DIAGNOSTICS

-1 is returned for an undefined file descriptor, seek on a pipe, or seek to a position before the beginning of file.

BUGS

`lseek` is a no-op on character special files.

ASSEMBLER

```
(lseek = 19.)
(file descriptor in tt r0)
sys lseek; offset1; offset2; whence
```

Offset1 and *offset2* are the high and low words of *offset*; tt r0 and tt r1 contain the pointer upon return.

NAME

mknod — make a directory or a special file

SYNOPSIS

```
mknod(name, mode, addr)
char *name;
```

DESCRIPTION

mknod creates a new file whose name is the null-terminated string pointed to by *name*. The mode of the new file (including directory and special file bits) is initialized from *mode*. (The protection part of the mode is modified by the process's mode mask; see **umask(2)**). The first block pointer of the i-node is initialized from *addr*. For ordinary files and directories *addr* is normally zero. In the case of a special file, *addr* specifies which special file.

mknod may be invoked only by the super-user.

SEE ALSO

mkdir(1), mknod(1), filsys(5)

DIAGNOSTICS

Zero is returned if the file has been made;
-1 if the file already exists or if the user is not the super-user.

ASSEMBLER

```
(mknod = 14.)
sys mknod; name; mode; addr
```

NAME

mount, umount — mount or remove file system

SYNOPSIS

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

umount(special)
char *special;
```

DESCRIPTION

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

Name must exist already. *Name* must be a directory (unless the root of the mounted file system is not a directory). Its old contents are inaccessible while the file system is mounted.

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

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

SEE ALSO

mount(1)

DIAGNOSTICS

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

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

ASSEMBLER

```
(mount = 21.)
sys mount; special; name; rwflag
```

NAME

nice — set program priority

SYNOPSIS

`nice(`*incr*`)`

DESCRIPTION

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without flak from the administration.

Negative increments are ignored except on behalf of the super-user. The priority is limited to the range -20 (most urgent) to 20 (least).

The priority of a process is passed to a child process by **fork(2)**. For a privileged process to return to normal priority from an unknown state, `nice` should be called successively with arguments -40 (goes to priority -20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

SEE ALSO

nice(1)

ASSEMBLER

```
(nice = 34.)
(priority in r0)
sys nice
```

NAME

open — open for reading or writing

SYNOPSIS

```
open(name, mode)
char *name;
```

DESCRIPTION

open opens the file *name* for reading (if *mode* is 0), writing (if *mode* is 1) or for both reading and writing (if *mode* is 2). *Name* is the address of a string of ASCII characters representing a path name, terminated by a null character.

The file is positioned at the beginning (byte 0). The returned file descriptor must be used for subsequent calls for other input-output functions on the file.

SEE ALSO

creat(2), read(2), write(2), dup(2), close(2)

DIAGNOSTICS

The value -1 is returned if the file does not exist, if one of the necessary directories does not exist or is unreadable, if the file is not readable (resp. writable), or if too many files are open.

ASSEMBLER

```
(open = 5.)
sys open; name; mode
(file descriptor in tt r0)
```

NAME

pause — stop until signal

SYNOPSIS

```
pause( )
```

DESCRIPTION

pause never returns normally. It is used to give up control while waiting for a signal from kill(2) or alarm(2).

SEE ALSO

kill(1), kill(2), alarm(2), signal(2), setjmp(3)

ASSEMBLER

```
(pause = 29.)
sys pause
```

NAME

pipe — create an interprocess channel

SYNOPSIS

```
pipe(fildes)
int fildes[2];
```

DESCRIPTION

The `pipe` system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor *fildes[1]* up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor *fildes[0]* will pick up the data. Writes with a count of 4096 bytes or less are atomic; no other process can intersperse data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent `fork` calls) will pass data through the pipe with `read` and `write` calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

SEE ALSO

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

DIAGNOSTICS

The function value zero is returned if the pipe was created; -1 if too many files are already open. A signal is generated if a write on a pipe with only one end is attempted.

BUGS

Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.

ASSEMBLER

```
(pipe = 42.)
sys pipe
(read file descriptor in r0)
(write file descriptor in r1)
```

NAME

profil — execution time profile

SYNOPSIS

```
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 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: $.1777777_8$ gives a 1-1 mapping of pc's to words in *buff*; $.777777_8$ maps each pair of instruction words together. 2_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 may be turned off if an update in *buff* would cause a memory fault.

SEE ALSO

monitor(3), prof(1)

ASSEMBLER

```
(profil = 44.)
sys profil; buff; bufsiz; offset; scale
```

NAME

ptrace — process trace

SYNOPSIS

```
#include <signal.h>

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

DESCRIPTION

ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like "illegal instruction" or externally generated like "interrupt." See **signal(2)** for the list. Then the traced process enters a stopped state and its parent is notified via **wait(2)**. When the child is in the stopped state, its core image can be examined and modified using ptrace. If desired, another ptrace request can then cause the child either to terminate or to continue, possibly ignoring the signal.

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

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

1,2 The word in the child process's address space at *addr* is returned. If I and D space are separated, request 1 indicates I space, 2 D space. *Addr* must be even. The child must be stopped. The input *data* is ignored.

3 The word of the system's per-process data area corresponding to *addr* is returned. *Addr* must be even and less than 512. This space contains the registers and other information about the process; its layout corresponds to the user structure in the system.

4,5 The given *data* is written at the word in the process's address space corresponding to *addr*, which must be even. No useful value is returned. If I

and D space are separated, request 4 indicates I space, 5 D space. Attempts to write in pure procedure fail if another process is executing the same file.

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

7 The *data* argument is taken as a signal number and the child's execution continues at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If *addr* is `(int *)1` then execution continues from where it stopped.

8 The traced process terminates.

9 Execution continues as in request 7. However, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The *wait* call is used to determine when a process stops; in such a case the "termination" status returned by *wait* has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, `ptrace` inhibits the set-user-id facility on subsequent `exec`(2) calls. If a traced process calls `exec`, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

SEE ALSO

wait(2), signal(2), adb(1)

DIAGNOSTICS

The value -1 is returned if *request* is invalid, *pid* is not a traceable process, *addr* is out of bounds, or *data* specifies an illegal signal number.

BUGS

On the Interdata 8/32, "as soon as possible" (request 7) means "as soon as a store instruction has been executed."

The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating

point (which use "illegal instruction" signals at a very high rate) could be efficiently debugged. The error indication, -1, is a legitimate function value; errno, see intro(2), can be used to disambiguate.

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

ASSEMBLER

```
(ptrace = 26.)
(data in r0)
sys ptrace; pid; addr; request
(value in r0)
```

NAME

read — read from file

SYNOPSIS

```
read(fildes, buffer, nbytes)
char *buffer;
```

DESCRIPTION

A file descriptor is a word returned from a successful `open`, `creat`, `dup`, or `pipe` call. *Buffer* is the location of *nbytes* contiguous bytes into which the input will be placed. It is not guaranteed that all *nbytes* bytes will be read; for example if the file refers to a typewriter at most one line will be returned. In any event the number of characters read is returned.

If the returned value is 0, then end-of-file has been reached.

SEE ALSO

open(2), creat(2), dup(2), pipe(2)

DIAGNOSTICS

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the return value is -1. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous *nbytes*, file descriptor not that of an input file.

ASSEMBLER

```
(read = 3.)
(file descriptor in r0)
sys read; buffer; nbytes
(byte count in r0)
```

NAME

setuid, setgid — set user and group ID

SYNOPSIS

`setuid(`*uid*`)`

`setgid(`*gid*`)`

DESCRIPTION

The user ID (group ID) of the current process is set to the argument. Both the effective and the real ID are set. These calls are only permitted to the super-user or if the argument is the real ID.

SEE ALSO

getuid(2)

DIAGNOSTICS

Zero is returned if the user (group) ID is set; -1 is returned otherwise.

ASSEMBLER

```
(setuid = 23.)
(user ID in r0)
sys setuid

(setgid = 46.)
(group ID in r0)
sys setgid
```

NAME

signal — catch or ignore signals

SYNOPSIS

```
#include <signal.h>

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

DESCRIPTION

A signal is generated by some abnormal event, initiated either by user at a typewriter (quit, interrupt), by a program error (bus error, etc.), or by request of another program (kill). Normally all signals cause termination of the receiving process, but a `signal` call allows them either to be ignored or to cause an interrupt to a specified location. Here is the list of signals with names as in the include file.

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction (not reset when caught)
SIGTRAP	5*	trace trap (not reset when caught)
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe or link with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
	16	unassigned

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

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination, sometimes with a core image. If *func* is SIG_IGN the signal is ignored. Otherwise when the signal occurs *func* will be called with the signal number as argument. A return from the function will continue the process at the point it was

interrupted. Except as indicated, a signal is reset to SIG_DFL after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another `signal` call.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during a **read** or **write(2)** on a slow device (like a typewriter; but not a file); and during **pause** or **wait(2)**. When such a signal occurs, the saved user status is arranged in such a way that when return from the signal-catching takes place, it will appear that the system call returned an error status. The user's program may then, if it wishes, re-execute the call.

The value of `signal` is the previous (or initial) value of *func* for the particular signal.

After a **fork(2)** the child inherits all signals. **Exec(2)** resets all caught signals to default action.

SEE ALSO

kill(1), kill(2), ptrace(2), setjmp(3)

DIAGNOSTICS

The value (int)-1 is returned if the given signal is out of range.

BUGS

If a repeated signal arrives before the last one can be reset, there is no chance to catch it.

The type specification of the routine and its *func* argument are problematical.

ASSEMBLER

```
(signal = 48.)
sys signal; sig; label
(old label in r0)
```

If *label* is 0, default action is reinstated. If *label* is odd, the signal is ignored. Any other even *label* specifies an address in the process where an interrupt is simulated. An RTI or RTT instruction will return from the interrupt.

NAME

stat, fstat — get file status

SYNOPSIS

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

stat(name, buf)
char *name;
struct stat *buf;

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

DESCRIPTION

stat obtains detailed information about a named file. fstat obtains the same information about an open file known by the file descriptor from a successful **open, creat, dup or pipe(2)** call.

Name points to a null-terminated string naming a file; *buf* is the address of a buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be searchable. The layout of the structure pointed to by buf as defined in <stat.h> is given below. St_mode is encoded according to the #define statements.

```
struct stat {
        dev_t           st_dev;
        ino_t           st_ino;
        unsigned short  st_mode;
        short           st_nlink;
        short           st_uid;
        short           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_IFMPC    0030000   /* multiplexed char special */
#define   S_IFMPB    0070000   /* multiplexed block special */
#define   S_ISUID    0004000   /* set user id on execution */
#define   S_ISGID    0002000   /* set group id on execution */
#define   S_ISVTX    0001000   /* save swapped text even after use */
#define   S_IREAD    0000400   /* read permission, owner */
#define   S_IWRITE   0000200   /* write permission, owner */
#define   S_IEXEC    0000100   /* execute/search permission, owner */
```

The mode bits 0000070 and 0000007 encode group and others permissions (see **chmod(2)**). The defined types, `ino_t`, `off_t`, `time_t`, name various width integer values; `dev_t` encodes major and minor device numbers; their exact definitions are in the include file **<sys/types.h>** (see **types(5)**.

When *fildes* is associated with a pipe, `fstat` reports an ordinary file with restricted permissions. The size is the number of bytes queued in the pipe.

`st_atime` is the file was last read. For reasons of efficiency, it is not set when a directory is searched, although this would be more logical. `st_mtime` is the time the file was last written or created. It is not set by changes of owner, group, link count, or mode. `st_ctime` is set both both by writing and changing the i-node.

SEE ALSO

ls(1), filsys(5)

DIAGNOSTICS

Zero is returned if a status is available; -1 if the file cannot be found.

ASSEMBLER

```
(stat = 18.)
sys stat; name; buf

(fstat = 28.)
(file descriptor in r0)
sys fstat; buf
```

NAME

stime — set time

SYNOPSIS

```
stime(tp)
long *tp;
```

DESCRIPTION

stime sets the system's idea of the time and date. Time, pointed to by *tp*, is measured in seconds from 0000 GMT Jan 1, 1970. Only the super-user may use this call.

SEE ALSO

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

DIAGNOSTICS

Zero is returned if the time was set;
-1 if user is not the super-user.

ASSEMBLER

```
(stime = 25.)
(time in r0-r1)
sys stime
```

NAME

sync — update super-block

SYNOPSIS

```
sync( )
```

DESCRIPTION

sync causes all information in core memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example **icheck**, **df**, etc. It is mandatory before a boot.

SEE ALSO

sync(1), update(8)

BUGS

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

ASSEMBLER

```
(sync = 36.)
sys sync
```

NAME

time, ftime — get date and time

SYNOPSIS

```
long time(0)

long time(it tloc)
long *it tloc;

#include <sys/types.h>
#include <sys/timeb.h>
ftime(it tp)
struct timeb *it tp;
```

DESCRIPTION

time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If *tloc* is nonnull, the return value is also stored in the place to which *tloc* points.

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

```
/*
* Structure returned by ftime system call
*/

struct timeb {
      time_t    time;
      unsigned  short millitm;
      short     timezone;
      short     dstflag;

};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

SEE ALSO

date(1), stime(2), ctime(3)

ASSEMBLER

```
(ftime = 35.)
sys ftime; bufptr

(time = 13.; obsolete call)
sys time
(time since 1970 in r0-r1)
```

NAME

umask — set file creation mode mask

SYNOPSIS

umask(*complmode*)

DESCRIPTION

umask sets a mask used whenever a file is created by **creat(2)** or **mknod(2)**: the actual mode (see **chmod(2)**) of the newly-created file is the logical and of the given mode and the complement of the argument. Only the low-order 9 bits of the mask (the protection bits) participate. In other words, the mask shows the bits to be turned off when files are created.

The previous value of the mask is returned by the call. The value is initially 0 (no restrictions). The mask is inherited by child processes.

SEE ALSO

creat(2), mknod(2), chmod(2)

ASSEMBLER

```
(umask = 60.)
sys umask; complmode
```

NAME

unlink — remove directory entry

SYNOPSIS

```
unlink(name)
char *name;
```

DESCRIPTION

Name points to a null-terminated string. `unlink` removes the entry for the file pointed to by *name* from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

SEE ALSO

rm(1), link(2)

DIAGNOSTICS

Zero is normally returned; -1 indicates that the file does not exist, that its directory cannot be written, or that the file contains pure procedure text that is currently in use. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user).

ASSEMBLER

```
(unlink = 10.)
sys unlink; name
```

NAME

utime — set file times

SYNOPSIS

```
#include <sys/types.h>
utime(file, timep)
char *file;
time_t timep[2];
```

DESCRIPTION

The utime call uses the "accessed" and "updated" times in that order from the *timep* vector to set the corresponding recorded times for *file*.

The caller must be the owner of the file or the super-user. The "inode-changed" time of the file is set to the current time.

SEE ALSO

stat (2)

ASSEMBLER

```
(utime = 30.)
sys utime; file; timep
```

NAME

wait — wait for process to terminate

SYNOPSIS

```
wait(status)
int *status;

wait(0)
```

DESCRIPTION

`wait` causes its caller to delay until a signal is received or one of its child processes terminates. If any child has died since the last `wait`, return is immediate; if there are no children, return is immediate with the error bit set (resp. with a value of -1 returned). The normal return yields the process ID of the terminated child. In the case of several children several `wait` calls are needed to learn of all the deaths.

If (int)*status* is nonzero, the high byte of the word pointed to receives the low byte of the argument of `exit` when the child terminated. The low byte receives the termination status of the process. See **signal(2)** for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted. See **ptrace(2)**. If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

SEE ALSO

exit(2), fork(2), signal(2)

DIAGNOSTICS

Returns -1 if there are no children not previously waited for.

ASSEMBLER

```
(wait = 7.)
sys wait
(process ID in r0)
(status in r1)
```

The high byte of the status is the low byte of `r0` in the child at termination.

NAME

write — write on a file

SYNOPSIS

```
write(fildes, buffer, nbytes)
char *buffer;
```

DESCRIPTION

A file descriptor is a word returned from a successful open, creat, dup, or pipe(2) call.

Buffer is the address of *nbytes* contiguous bytes which are written on the output file. The number of characters actually written is returned. It should be regarded as an error if this is not the same as requested.

Writes which are multiples of 512 characters long and begin on a 512-byte boundary in the file are more efficient than any others.

SEE ALSO

creat(2), open(2), pipe(2)

DIAGNOSTICS

Returns -1 on error: bad descriptor, buffer address, or count; physical I/O errors.

ASSEMBLER

```
(write = 4.)
(file descriptor in r0)
sys write; buffer; nbytes
(byte count in r0)
```

NAME

intro — introduction to library functions

SYNOPSIS

```
#include <stdio.h>
#include <math.h>
```

DESCRIPTION

This section describes functions that may be found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in section 2. Functions are divided into various libraries distinguished by the section number at the top of the page:

(3) These functions, together with those of section 2 and those marked (3S), constitute library **libc**, which is automatically loaded by the C compiler **cc**(1) and the Fortran compiler **f77**(1). The link editor **ld**(1) searches this library under the "`-lc`" option. Declarations for some of these functions may be obtained from include files indicated on the appropriate pages.

(3M) These functions constitute the math library, **libm**. They are automatically loaded as needed by the Fortran compiler **f77**(1). 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", see **stdio**(3). These functions are in the library libc already mentioned. Declarations for these functions may be obtained from the include file <stdio.h>.

(3X) Various specialized libraries have not been given distinctive captions. The files in which these libraries are found are named on the appropriate pages.

FILES

`/lib/libc.a`
`/lib/libm.a, /usr/lib/libm.a` (one or the other)

SEE ALSO

stdio(3), nm(1), ld(1), cc(1), f77(1), intro(2)

DIAGNOSTICS

Functions in the math library (3M) may return conventional values 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. The values of EDOM and ERANGE are defined in the include file `<math.h>`.

ASSEMBLER

In assembly language these functions may be accessed by simulating the C calling sequence. For example, **ecvt(3)** might be called this way:

```
setd
mov     $sign,-(sp)
mov     $decpt,-(sp)
mov     ndigit,-(sp)
movf    value,-(sp)
jsr     pc,_ecvt
add     $14.,sp
```

NAME

abort — generate IOT fault

DESCRIPTION

abort executes the PDP11 IOT instruction. This causes a signal that normally terminates the process with a core dump, which may be used for debugging.

SEE ALSO

adb(1), signal(2), exit(2)

DIAGNOSTICS

Usually "IOT trap - core dumped" from the shell.

NAME

abs — integer absolute value

SYNOPSIS

abs(*i*)

DESCRIPTION

abs returns the absolute value of its integer operand.

SEE ALSO

floor(3) for `fabs`

BUGS

You get what the hardware gives on the largest negative integer.

NAME

assert — program verification

SYNOPSIS

```
#include <assert.h>
assert (expression)
```

DESCRIPTION

`assert` is a macro that indicates *expression* is expected to be true at this point in the program. It causes an **exit(2)** with a diagnostic comment on the standard output when *expression* is false (0). Compiling with the cc(1) option -DNDEBUG effectively deletes `assert` from the program.

DIAGNOSTICS

"`Assertion failed: file` *f* `line` *n*." *F* is the source file and *n* the source line number of the `assert` statement.

NAME

atof, atoi, atol — convert ASCII to numbers

SYNOPSIS

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

atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

DESCRIPTION

These functions convert a string pointed to by *nptr* to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

`atof` recognizes an optional string of tabs and spaces, 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.

`atoi` and `atol` recognize an optional string of tabs and spaces, then an optional sign, then a string of digits.

SEE ALSO

scanf(3)

BUGS

There are no provisions for overflow.

NAME

crypt, setkey, encrypt — DES encryption

SYNOPSIS

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

setkey(key)
char *key;

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

DESCRIPTION

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

The first argument to crypt is a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. The *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

The other 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, leading to a 56-bit key which is set into the machine.

The argument to the encrypt entry is likewise a character array of length 64 containing 0's and 1's. 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 0, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

passwd(1), passwd(5), login(1), getpass(3)

BUGS

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

NAME

ctime, localtime, gmtime, asctime, timezone — convert date and time to ASCII

SYNOPSIS

```
char *ctime(clock)
long *clock;

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)
```

DESCRIPTION

ctime converts a time pointed to by clock such as returned by time(2) into ASCII 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 structures containing the broken-down time. Localtime corrects for the time zone and possible daylight savings time; gmtime converts directly to GMT, which is the time UNIX uses. Asctime converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file is:

```
struct tm {/* see ctime(3) */
        int tm_sec;
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;
        int tm_yday;
        int tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday = 0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

When local time is called for, the program consults the system to determine the time zone and whether the standard U.S.A. daylight saving time adjustment is appropriate. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

Timezone returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used, otherwise the Daylight Saving version. If the required name does not appear in a table built into the routine, the difference from GMT is produced; e.g. in Afghanistan `timezone(-(60*4+30) , 0)` is appropriate because it is 4:30 ahead of GMT and the string GMT+4:30 is produced.

SEE ALSO

time(2)

BUGS

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

NAME

isalpha, isupper, islower, isdigit, isalnum, isspace,
ispunct, isprint, iscntrl, isascii - character classification

SYNOPSIS

```
#include <ctype.h>
isalpha(c) ...
```

DESCRIPTION

These macros classify ASCII-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 (see **stdio(3)**).

`isalpha`	*c* is a letter
`isupper`	*c* is an upper case letter
`islower`	*c* is a lower case letter
`isdigit`	*c* is a digit
`isalnum`	*c* is an alphanumeric character
`isspace`	*c* is a space, tab, carriage return, newline, or formfeed
`ispunct`	*c* is a punctuation character (neither control nor alphanumeric)
`isprint`	*c* is a printing character, code 040_8 (space) through 0176 (tilde)
`iscntrl`	*c* is a delete character (0177) or ordinary control character (less than 040).
`isascii`	*c* is an ASCII character, code less than 0200

SEE ALSO

ascii(7)

NAME

dbminit, fetch, store, delete, firstkey, nextkey — data base subroutines

SYNOPSIS

```
typedef struct  char *dptr; int dsize;  datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey();

datum nextkey(key);
datum key;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two filesystem accesses. The functions are obtained with the loader option `-ldbm`.

keys and *contents* are described by the `datum` typedef. A `datum` specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has "`.dir`" as its suffix. The second file contains all data and has "`.pag`" as its suffix.

Before a database can be accessed, it must be opened by `dbminit`. At the time of this call, the files *file*.dir and *file*.pag must exist. (An empty database is created by creating zero-length "`.dir`" and "`.pag`" files.)

Once open, the data stored under a key is accessed by `fetch` and data is placed under a key by `store`. A key (and its associated contents) is deleted by `delete`. A

linear pass through all keys in a database may be made, in an (apparently) random order, by use of `firstkey` and `nextkey`. `Firstkey` will return the first key in the database. With any key `nextkey` will return the next key in the database. This code will traverse the data base:

```
for (key=firstkey(); key.dptr!=NULL; key=nextkey(key))
```

DIAGNOSTICS

All functions that return an `int` indicate errors with negative values. A zero return indicates ok. Routines that return a `datum` indicate errors with a null (0) *dptr*.

BUGS

The ".pag" file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 512 bytes). Moreover all key/content pairs that hash together must fit on a single block. `Store` will return an error in the event that a disk block fills with inseparable data.

`delete` does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by `firstkey` and `nextkey` depends on a hashing function, not on anything interesting.

NAME

ecvt, fcvt, gcvt — output conversion

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 the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer thereto. 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). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero. The low-order digit is rounded.

`fcvt` is identical to `ecvt`, except that the correct di has been rounded for Fortran F-format output of the number of digits specified by *ndigits*.

`gcvt` converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in Fortran F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

SEE ALSO

printf(3)

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 coincides with `end`, but many functions reset the program break, among them the routines of **brk(2)**, **malloc(3)**, standard input/output (**stdio(3)**), the profile (**-p**) option of **cc(1)**, etc. The current value of the program break is reliably returned by "`sbrk(0)`", see **brk(2)**.

SEE ALSO

brk(2), malloc(3)

NAME

exp, log, log10, pow, sqrt — exponential, logarithm, power, square root

SYNOPSIS

```
#include <math.h>
double x;

double exp(x)
double log(x)
double log10(x)
double pow( x , y )
double sqrt(x)
```

DESCRIPTION

exp returns e^x

log returns $\log_e$

log10 returns $\log_{10}$

pow returns x^y.

sqrt returns $\sqrt{x}$

SEE ALSO

hypot(3), sinh(3), intro(2)

DIAGNOSTICS

exp and pow return a huge value when the correct value would overflow; errno is set to ERANGE.
pow returns 0 and sets errno to EDOM when the second argument is negative and non-integral and when both arguments are 0.
log returns 0 when x is zero or negative; errno is set to EDOM.
sqrt returns 0 when x is negative; errno is set to EDOM.

NAME

fclose, fflush — close or flush a stream

SYNOPSIS

```
#include <stdio.h>

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

DESCRIPTION

`fclose` causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed.

`fclose` is performed automatically upon calling **exit(2)**.

`fflush` causes any buffered data for the named output *stream* to be written to that file. The stream remains open.

SEE ALSO

close(2), fopen(3), setbuf(3)

DIAGNOSTICS

These routines return EOF if *stream* is not associated with an output file, or if buffered data cannot be transferred to that file.

NAME

feof, ferror, clearerr, fileno — stream status inquiries

SYNOPSIS

```
#include <stdio.h>

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream

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

DESCRIPTION

`feof` returns non-zero when end of file is read on the named input *stream*, otherwise zero.

`ferror` returns non-zero when an error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by `clearerr`, the error indication lasts until the stream is closed.

`clrerr` resets the error indication on the named *stream*.

`fileno` returns the integer file descriptor associated with the *stream*, see **open(2)**.

These functions are implemented as macros; they cannot be redeclared.

SEE ALSO

fopen(3), open(2)

NAME

fabs, floor, ceil — absolute value, floor, ceiling functions

SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double(x);
```

DESCRIPTION

`fabs` returns the absolute value $|x|$.

`floor` returns the largest integer not greater than x.

`ceil` returns the smallest integer not less than x.

SEE ALSO

abs(3)

NAME

fopen, freopen, fdopen — open a stream

SYNOPSIS

```
#include <stdio.h>

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

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

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

DESCRIPTION

`fopen` opens the file named by *filename* and associates a stream with it. `fopen` returns a pointer to be used to identify the stream in subsequent operations.

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

`"r"` open for reading

`"w"` create for writing

`"a"` append: open for writing at end of file, or create for writing

`freopen` substitutes the named file in place of the open *stream*. It returns the original value of *stream*. The original stream is closed.

`freopen` is typically used to attach the preopened constant names, stdin, stdout, stderr, to specified files.

`fdopen` associates a stream with a file descriptor obtained from **open, dup, creat,** or **pipe(2)**. The *type* of the stream must agree with the mode of the open file.

SEE ALSO

open(2), fclose(3)

DIAGNOSTICS

`fopen` and `freopen` return the pointer NULL if *filename* cannot be accessed.

BUGS

`fdopen` is not portable to systems other than UNIX.

NAME

fread, fwrite — buffered binary input/output

SYNOPSIS

```
#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;

fwrite(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

DESCRIPTION

`fread` reads, into a block beginning at *ptr*, *nitems* of data of the type of **ptr* from the named input *stream*. It returns the number of items actually read.

`fwrite` appends at most *nitems* of data of the type of **ptr* beginning at *ptr* to the named output *stream*. It returns the number of items actually written.

SEE ALSO

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

DIAGNOSTICS

`fread` and `fwrite` return 0 upon end of file or error.

NAME

frexp, ldexp, modf — split into mantissa and exponent

SYNOPSIS

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

double ldexp(value, exp)
double value;

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

DESCRIPTION

`frexp` returns the mantissa of a double *value* as a double quantity, x, of magnitude less than 1 and stores an integer n such that $value = x2^n$ indirectly through *eptr*.

`ldexp` returns the quantity $value2^{exp}$.

`modf` returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*.

NAME

fseek, ftell, rewind — reposition a stream

SYNOPSIS

```
#include <stdio.h>

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

long ftell(stream)
FILE *stream;

rewind(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, according as *ptrname* has the value 0, 1, or 2.

`fseek` undoes any effects of **ungetc(3)**.

`ftell` returns the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes on UNIX; on some other systems it is a magic cookie, and the only foolproof way to obtain an *offset* for *fseek*.

`Rewind(stream)` is equivalent to `fseek(stream, 0L, 0)`.

SEE ALSO

lseek(2), fopen(3)

DIAGNOSTICS

`fseek` returns -1 for improper seeks.

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 from the named input *stream*.

`getchar()` is identical to `getc( stdin)`.

`fgetc` behaves like `getc`, but is a genuine function, not a macro; it may be used to save object text.

`getw` returns the next word from the named input *stream*. It returns the constant EOF upon end of file or error, but since that is a good integer value, **feof** and **ferror(3)** should be used to check the success of `getw`. `getw` assumes no special alignment in the file.

SEE ALSO

fopen(3), putc(3), gets(3), scanf(3), fread(3), ungetc(3)

DIAGNOSTICS

These functions return the integer constant EOF at end of file or upon read error.

A stop with message, `Reading bad file`, means an attempt has been made to read from a stream that has not been opened for reading by `fopen`.

BUGS

The end-of-file return from `getchar` is incompatible with that in previous editions of UNIX .

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

NAME

getenv — 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 *value* if such a string is present, otherwise 0 (NULL).

SEE ALSO

environ(5), exec(2)

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent — get group file entry

SYNOPSIS

```
#include <grp.h>
struct group *getgrent();
struct group *getgrgid(gid) int gid;
struct group *getgrnam(name) char *name;
int setgrent();
int endgrent();
```

DESCRIPTION

getgrent, getgrgid and getgrnam each return pointers to an object with the following structure containing the brokenout fields of a line in the group file.

```
struct group {/* see getgrent(3) */
        char *gr_name;
        char *gr_passwd;
        int gr_gid;
        char **gr_mem;
};
```

The members of this structure are:

gr_name The name of the group.

gr_passwd The encrypted password of the group.

gr_gid The numerical group-ID.

gr_mem Null-terminated vector of pointers to the individual member names.

getgrent simply reads the next line while getgrgid and getgrnam search until a matching *gid* or *name* is found (or until EOF is encountered). Each routine picks up where the others leave off so successive calls may be used to search the entire file.

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(3), getpwent(3), group(5)

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

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 userid is shared by several login names.

If `getlogin` is called within a process that is not attached to a typewriter, it returns NULL. The correct procedure for determining the login name is to first call `getlogin` and if it fails, to call `getpwuid`.

FILES

`/etc/utmp`

SEE ALSO

getpwent(3), getgrent(3), utmp(5)

DIAGNOSTICS

Returns NULL (0) if name not found.

BUGS

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

NAME

getpass — read a password

SYNOPSIS

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

DESCRIPTION

getpass reads a password from the file /dev/tty, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters.

FILES

/dev/tty

SEE ALSO

crypt(3)

BUGS

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

NAME

getpw — get name from UID

SYNOPSIS

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

DESCRIPTION

getpw searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

FILES

/etc/passwd

SEE ALSO

getpwent(3), passwd(5)

DIAGNOSTICS

Non-zero return on error.

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;
int setpwent();
int 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 password file.

```
struct passwd {/* see getpwent(3) */
        char *pw_name;
        char *pw_passwd;
        int pw_uid;
        int pw_gid;
        int pw_quota;
        char *pw_comment;
        char *pw_gecos;
        char *pw_dir;
        char *pw_shell;
};
```

The fields *pw_quota* and *pw_comment* are unused; the others have meanings described in **passwd(5)**.

getpwent reads the next line (opening the file if necessary); setpwent rewinds the file; endpwent closes it.

getpwuid and getpwnam search from the beginning until a matching *uid* or *name* is found (or until EOF is encountered).

FILES

/etc/passwd .

SEE ALSO

getlogin(3), getgrent(3), passwd(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

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;
FILE *stream;
```

DESCRIPTION

gets reads a string into *s* from the standard input stream stdin. The string is terminated by a newline character, which is replaced in *s* by a null character. gets returns its argument.

fgets reads *n*−1 characters, or up to a newline character, whichever comes first, from the *stream* into the string *s*. The last character read into *s* is followed by a null character. fgets returns its first argument.

SEE ALSO

puts(3), getc(3), scanf(3), fread(3), ferror(3)

DIAGNOSTICS

gets and fgets return the constant pointer NULL upon end of file or error.

BUGS

gets deletes a newline, fgets keeps it, all in the name of backward compatibility.

NAME

hypot, cabs — euclidean distance

SYNOPSIS

```
#include <math.h>

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

double cabs(z)
struct {double x, y;}z;
```

DESCRIPTION

hypot and cabs return $\sqrt{x^2+y^2}$, taking precautions against unwarranted overflows.

SEE ALSO

exp(3) for sqrt

NAME

j0, j1, jn, y0, y1, yn — bessel functions

SYNOPSIS

```
#include <math.h>

double x;

double j0(x)
double j1(x)
double jn(n, x);
double y0(x)
double y1(x)
double yn(n, x)
```

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

DIAGNOSTICS

Negative arguments cause y0, y1, and yn to return a huge negative value and set errno to EDOM.

NAME

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

SYNOPSIS

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

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

DESCRIPTION

l3tol converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

ltol3 performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance; disk addresses are three bytes long.

SEE ALSO

filsys(5)

NAME

malloc, free, realloc, calloc — main memory allocator

SYNOPSIS

```
char *malloc(size)
unsigned size;

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 beginning on a word boundary.

The argument to `free` is a pointer to a block previously allocated by `malloc`; this space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by `malloc` is overrun or if some random number is handed to `free`.

`malloc` allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls `sbrk` (see **break(2)**) to get more memory from the system when there is no suitable space already free.

`realloc` changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

`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 (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. `malloc` may be recompiled to check the arena very stringently on every transaction; see the source code.

BUGS

When `realloc` returns 0, the block pointed to by *ptr* may be destroyed.

NAME

mktemp — make a unique file name

SYNOPSIS

```
char *mktemp(template)
char *template;
```

DESCRIPTION

mktemp replaces *template* by a unique file name, and returns the address of the template. The template should look like a file name with six trailing X's, which will be replaced with the current process id and a unique letter.

SEE ALSO

getpid(2)

NAME

monitor — prepare execution profile

SYNOPSIS

```
monitor(lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)(), (*highpc)();
short buffer[];
```

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 of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option -p of **cc(1)** are 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(3)**.

To stop execution monitoring and write the results on the file `mon.out`, use

```
monitor(0);
```

then **prof(1)** can be used to examine the results.

FILES

`mon.out`

SEE ALSO

prof(1), profil(2), cc(1)

NAME

itom, madd, msub, mult, mdiv, min, mout, pow, gcd, rpow — multiple precision integer arithmetic

SYNOPSIS

```
typedef struct  int len; short *val;  mint;

madd(a, b, c)
msub(a, b, c)
mult(a, b, c)
mdiv(a, b, q, r)
min(a)
mout(a)
pow(a, b, m, c)
gcd(a, b, c)
rpow(a, b, c)
msqrt(a, b, r)
mint *a, *b, *c, *m, *q, *r;

sdiv(a, n, q, r)
mint *a, *q;
short *r;

mint *itom(n)
```

DESCRIPTION

These routines perform arithmetic on integers of arbitrary length. The integers are stored using the defined type `mint`. Pointers to a `mint` should be initialized using the function `itom`, which sets the initial value to *n*. After that space is managed automatically by the routines.

`madd`, `msub`, `mult`, assign to their third arguments the sum, difference, and product, respectively, of their first two arguments. `mdiv` assigns the quotient and remainder, respectively, to its third and fourth arguments. `sdiv` is like `mdiv` except that the divisor is an ordinary integer. `msqrt` produces the square root and remainder of its first argument. `rpow` calculates *a* raised to the power *b*, while pow calculates this reduced modulo *m*. `min` and `mout` do decimal input and output.

The functions are obtained with the loader option `-lmp`.

DIAGNOSTICS

Illegal operations and running out of memory produce messages and core images.

NAME

nlist — get entries from name list

SYNOPSIS

```
#include <a.out.h>
nlist(filename, nl)
char *filename;
struct nlist nl[ ];
```

DESCRIPTION

`nlist` examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See **a.out(5)** for the structure declaration.

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(5)

DIAGNOSTICS

All type entries are set to 0 if the file cannot be found or if it is not a valid namelist.

NAME

perror, sys_errlist, sys_nerr — system error messages

SYNOPSIS

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];
```

DESCRIPTION

perror produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable errno (see **intro(2)**), which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings sys_errlist is provided; errno can be used as an index in this table to get the message string without the newline. Sys_nerr is the number of messages 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

plot: openpl et al. — graphics interface

SYNOPSIS

```
openpl( ) erase( )
label(s) char s[ ];
line(x1, y1, x2, y2)
circle(x, y, r)
arc(x, y, x0, y0, x1, y1)
move(x, y)
cont(x, y)
point(x, y)
linemod(s) char s[ ];
space(x0, y0, x1, y1)
closepl( )
```

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. See **plot(5)** for a description of their effect. `openpl` must be used before any of the others to open the device for writing. `closepl` flushes the output.

String arguments to `label` and `linemod` are null-terminated, and do not contain newlines.

Various flavors of these functions exist for different output devices. They are obtained by the following **ld(1)** options:

`-lplot`	device-independent filter to standard output for **plot(1)**
`-l300`	GSI 300 terminal
`-l300s`	GSI 300S terminal
`-l450`	DASI 450 terminal
`-l4014`	Tektronix 4014 terminal

SEE ALSO

plot(5), plot(1), graph(1)

NAME

popen, pclose — initiate I/O to/from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen(command, type)
char *command, *type;

pclose(stream)
FILE *stream;
```

DESCRIPTION

The arguments to popen are pointers to null-terminated strings containing respectively a shell command line and an I/O mode, either "r" for reading or "w" for writing. It creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

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), fopen(3), fclose(3), system(3), wait(2)

DIAGNOSTICS

popen returns a null pointer if files or processes cannot be created, or the Shell cannot be accessed.

pclose returns -1 if *stream* is not associated with a 'popened' command.

BUGS

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, e.g. with `fflush`, see **fclose(3)**.

NAME

printf, fprintf, sprintf — formatted output conversion

SYNOPSIS

```
#include <stdio.h>

printf(format [, arg ] ... )
char *format;

fprintf(stream, format [, arg ] ... )
FILE *stream;
char *format;

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" in the string *s*, followed by the character "\0".

Each of these functions converts, formats, and prints its arguments after the first under control of the first argument. The first argument is a character string which ntains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive *arg* `printf`.

Each conversion specification is introduced by the character %. Following the %, there may be

- an optional minus sign "-" which specifies *left adjustment* of the converted value in the indicated field;
- an optional digit string specifying a *field width*; if the converted value has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank padding;
- an optional period "." which serves to separate the field width from the next digit string;

- an optional digit string specifying a *precision* which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- the character l specifying that a following d, o, x, or u corresponds to a long integer *arg*. (A capitalized conversion code accomplishes the same thing.)
- a character which indicates the type of conversion to be applied.

A field width or precision may be "*" instead of a digit string. In this case an integer *arg* supplies the field width or precision.

The conversion characters and their meanings are

dox The integer *arg* is converted to decimal, octal, or hexadecimal notation respectively.

f The float or double *arg* is converted to decimal notation in the style `[-]ddd.ddd` where the number of *d*'s after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e The float or double *arg* is converted in the style "`[-]d.ddde±dd`" where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

g The float or double *arg* is printed in style d, in style f, or in style e, whichever gives full precision in minimum space.

c The character *arg* is printed. Null characters are ignored.

s *Arg* is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

u The unsigned integer *arg* is converted to decimal and printed (the result will be in the range 0 to 65535).

% Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by `printf` are printed by **putc(3)**.

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, %02d:%02d", weekday, month, day, hour, min);
```

To print pi to 5 decimals:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

putc(3), scanf(3), ecvt(3)

BUGS

Very wide fields (>128 characters) fail.

NAME

putc, putchar, fputc, putw — put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

putchar(c)

fputc(c, stream)
FILE *stream;

putw(w, stream)
FILE *stream;
```

DESCRIPTION

putc appends the character *c* to the named output *stream*. It returns the character written.

putchar(*c*) is defined as putc(*c*, *stdout*).

fputc behaves like putc, but is a genuine function rather than a macro. It may be used to save on object text.

putw appends word (i.e. int) *w* to the output *stream*. It returns the word written. putw neither assumes nor causes special alignment in the file.

The standard stream stdout is normally buffered if and only if the output does not refer to a terminal; this default may be changed by **setbuf(3)**. The standard stream stderr is by default unbuffered unconditionally, but use of freopen (see **fopen(3)**) will cause it to become buffered; setbuf, again, will set the state to whatever is desired. When an output stream is unbuffered information appears on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block. fflush (see **fclose(3)**) may be used to force the block out early.

SEE ALSO

fopen(3), fclose(3), getc(3), puts(3), printf(3), fread(3)

DIAGNOSTICS

These functions return the constant EOF upon error. Since this is a good integer, **ferror(3)** should be used to detect putw errors.

BUGS

Because it is implemented as a macro, putc treats a *stream* argument with side effects improperly. In particular "putc(*c*, **f++*);" doesn't work sensibly.

NAME

puts, fputs — put a string on a stream

SYNOPSIS

```
#include <stdio.h>

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

puts copies the null-terminated string *s* to the standard output stream stdout and appends a newline character.

fputs copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminal null character.

SEE ALSO

fopen(3), gets(3), putc(3), printf(3), ferror(3) fread(3) for fwrite

BUGS

puts appends a newline, fputs does not, all in the name of backward compatibility.

NAME

qsort — quicker sort

SYNOPSIS

```
qsort(base, nel, width, compar)
char *base;
int (*compar)( );
```

DESCRIPTION

qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort(1)

NAME

rand, srand — random number generator

SYNOPSIS

```
srand(seed)
int seed;

rand( )
```

DESCRIPTION

rand uses a multiplicative congruential random number generator with period 2^32 to return successive pseudo-random numbers in the range from 0 to $2^15 - 1$.

The generator is reinitialized by calling srand with 1 as argument. It can be set to a random starting point by calling srand with whatever you like as argument.

NAME

scanf, fscanf, sscanf — formatted input conversion

SYNOPSIS

```
#include <stdio.h>

scanf(format [ , pointer ] . . . )
char *format;

fscanf(stream, format [ , pointer ] . . . )
FILE *stream;
char *format;

sscanf(s, format [ , pointer ] . . . )
char *s, *format;
```

DESCRIPTION

scanf reads from the standard input stream stdin. fscanf reads from the named input *stream*. sscanf reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects as arguments a control string *format*, described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which match optional white space in the input.
2. An ordinary character (not %) which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters 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.

`o` an octal integer is expected; the corresponding argument should be a integer pointer.

`x` a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

`s` a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating "\0", which will be added. The input field is terminated by a space character or a newline.

`c` a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, try "`%1s`". If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

`e,f` 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 `e` followed by an optionally signed integer.

`[` indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (`^`), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is `^`, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters `d`, `o`, and `x` may be capitalized or preceeded by `l` to indicate that a pointer to `long` rather than to `int` is in the argument list. Similarly, the conversion characters `e` or `f` may be capitalized or preceded by `l` to indicate a pointer to `double` rather than to `float`. The conversion characters `d`, `o`, and `x` may be preceeded by `h` to indicate a pointer to `short` rather than to `int`.

The scanf functions return the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant EOF is returned upon end of input; note that this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

For example, 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 to i the value 25, x the value 5.432, and name will contain "thompson\0". Or,

```
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &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 will return "a".

SEE ALSO

atof(3), getc(3), printf(3)

DIAGNOSTICS

The scanf 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>

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 *buf* to be used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be com- pletely unbuffered.

A manifest constant BUFSIZ tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from **malloc(3)** upon the first `getc` or **putc(3)** on the file, except that output streams directed to terminals, and the standard error stream `stderr` are normally not buffered.

SEE ALSO

fopen(3), getc(3), putc(3), malloc(3)

NAME

setjmp, longjmp — non-local goto

SYNOPSIS

```
#include <setjmp.h>

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
```

DESCRIPTION

These routines 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`. It returns value 0.

`longjmp` restores the environment saved by the last call of `setjmp`. It then returns in such a way that execution continues as if the call of `setjmp` had just returned the value *val* to the function that invoked `setjmp`, which must not itself have returned in the interim. All accessible data have values as of the time `longjmp` was called.

SEE ALSO

signal(2)

NAME

sin, cos, tan, asin, acos, atan, atan2 — trigonometric functions

SYNOPSIS

```
#include <math.h>

double x;

double sin(x)
double cos(x)
double asin(x)
double acos(x)
double atan(x)
double atan2(x, y)
```

DESCRIPTION

sin, cos and tan return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

asin returns the arc sin in the range $-\pi/2$ to $\pi/2$.

acos returns the arc cosine in the range 0 to π.

atan returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$.

atan2 returns the arc tangent of x/y in the range $-\pi$ to π .

DIAGNOSTICS

Arguments of magnitude greater than 1 cause asin and acos to return value 0. errno is set to EDOM. The value of tan at its singular points is a huge number, and errno is set to ERANGE.

BUGS

The value of tan for arguments greater than about 2^31 is garbage.

NAME

sinh, cosh, tanh — hyperbolic functions

SYNOPSIS

```
#include <math.h>

double x;

double sinh(x)
double cosh(x)
double tanh(x)
```

DESCRIPTION

These functions compute the designated hyperbolic functions for real arguments.

DIAGNOSTICS

`sinh` and `cosh` return a huge value of appropriate sign when the correct value would overflow.

NAME

sleep — suspend execution for interval

SYNOPSIS

```
sleep(seconds)
unsigned seconds;
```

DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and an arbitrary amount longer because of other activity in the system.

The routine is implemented by setting an alarm clock signal and pausing until it occurs. The previous state of this signal is saved and restored. If the sleep time exceeds the time to the alarm signal, the process sleeps only until the signal would have occurred, and the signal is sent 1 second later.

SEE ALSO

alarm(2), pause(2)

NAME

stdio — standard buffered input/output package

SYNOPSIS

```
#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;
```

DESCRIPTION

The functions described in Sections 3S constitute an efficient user-level buffering scheme. The in-line macros `getc` and **putc(3)** handle characters quickly. The higher level routines `gets`, `fgets`, `scanf`, `fscanf`, `fread`, `puts`, `fputs`, `printf`, `fprintf`, `fwrite` 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(3)** creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

`stdin`	standard input file
`stdout`	standard output file
`stderr`	standard error file

A constant "pointer" NULL (0) designates no stream at all.

An integer constant EOF (-1) is returned upon end of file or error by integer functions that deal with streams.

Any routine that uses the standard input/output package must include the header file <stdio.h> of pertinent macro definitions. The functions and constants mentioned in sections labeled 3S are declared in the include file and need no further declaration. The constants, and the following 'functions' are implemented as macros; redeclaration of these names is perilous: `getc`, `getchar`, `putc`, `putchar`, `feof`, `ferror`, `fileno`.

SEE ALSO

open(2), close(2), read(2), write(2)

DIAGNOSTICS

The value EOF is returned uniformly to indicate that a FILE pointer has not been initialized with fopen, input (output) has been attempted on an output (input) stream, or a FILE pointer designates corrupt or otherwise unintelligible FILE data.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex — string operations

SYNOPSIS

```
char *strcat(s1, s2); char *s1, *s2;

char *strncat(s1, s2, n); char *s1, *s2;

strcmp(s1, s2); char *s1, *s2;

strncmp(s1, s2, n); char *s1, *s2;

char *strcpy(s1, s2); char *s1, *s2;

char *strncpy(s1, s2, n); char *s1, *s2;

strlen(s); char *s;

char *index(s, c); char *s, c;

char *rindex(s, c); char *s;
```

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

`strcat` appends a copy of string *s2* to the end of string *s1*. `strncat` copies at most *n* characters. Both return a pointer to the null-terminated result.

`strcmp` compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. `strncmp` makes the same comparison but looks at at most *n* characters.

`strcpy` copies string *s2* to *s1*, stopping after the null character has been moved. `strncpy` copies exactly *n* characters, truncating or null-padding *s2*; the target may not be null-terminated if the length of *s2* is *n* or more. Both return *s1*.

`strlen` returns the number of non-null characters in *s*.

`index` (*rindex*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

BUGS

`strcmp` uses native character comparison, which is signed on PDP11's, unsigned on other machines.

NAME

swab — swap bytes

SYNOPSIS

```
swab(from, to, nbytes)
char *from, *to;
```

DESCRIPTION

swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP11's and other machines. *nbytes* should be even.

NAME

system — issue a shell command

SYNOPSIS

```
system(string)
char *string;
```

DESCRIPTION

system causes the *string* to be given to sh(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

SEE ALSO

popen(3), exec(2), wait(2)

DIAGNOSTICS

Exit status 127 indicates the shell couldn't be executed.

NAME

ttyname, isatty, ttyslot — find name of a terminal

SYNOPSIS

```
char *ttyname(fildes)
isatty(fildes)
ttyslot()
```

DESCRIPTION

`ttyname` returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *fildes*.

`isatty` returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

`ttyslot` returns the number of the entry in the ttys(5) file for the control terminal of the current process.

FILES

```
/dev/*
/etc/ttys
```

SEE ALSO

ioctl(2), ttys(5)

DIAGNOSTICS

`ttyname` returns a null pointer (0) if *fildes* does not describe a terminal device in directory "`/dev`".

`ttyslot` returns 0 if "`/etc/ttys`" is inaccessible or if it cannot determine the control terminal.

BUGS

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

NAME

ungetc — push character back into input stream

SYNOPSIS

```
#include <stdio.h>

ungetc(c, stream)
FILE *stream;
```

DESCRIPTION

ungetc pushes the character *c* back on an input stream. That character will be returned by the next getc call on that stream. ungetc returns *c*.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push EOF are rejected.

fseek(3) erases all memory of pushed back characters.

SEE ALSO

getc(3), setbuf(3), fseek(3)

DIAGNOSTICS

ungetc returns EOF if it can't push a character back.

NAME

mem, kmem — core memory

DESCRIPTION

mem is a special file that is an image of the core memory of the computer. It may be used, for example, to examine, and even to patch the system. kmem is the same as mem except that kernel virtual memory rather than physical memory is accessed.

Byte addresses are interpreted as memory addresses. References to non-existent locations return errors.

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present.

FILES

/dev/mem
/dev/kmem

BUGS

On PDP11's, memory files are accessed one byte at a time, an inappropriate method for some device registers.

NAME

null — data sink

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

`/dev/null`

Plexus Terminal I/O

(1) We have two ICP's (sioc's in dd jargon) They have major # 5 and 6. The minor numbers are 0-7 for serial lines and 8 for the parallel port. Device 9 links to the ICP's

(2) The ICP's are loaded at multiuser initiation by commands in /etc/rc

(3) The console is driven from the main cpu board directly

(4) There is no way to check for characters received on a serial line.

Plexus Terminal I/O

(1) We have two ICP's (stock in [illegible]). They have major # [illegible] in. The [illegible] numbers are 0-7 for serial lines and 8 for the parallel port. Device 8 links to the ICP's

(2) The ICP's are loaded at multiuser initiation by commands in /etc/rc.

(3) The console is driven from the main cpu board directly.

(4) There is no way to check the [illegible] [illegible] on Console [illegible]

NAME

acct — execution accounting file

SYNOPSIS

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

DESCRIPTION

acct(2) causes entries to be made into an accounting file for each process that terminates. The accounting file is a sequence of entries whose layout, as defined by the include file is:

```
typedef unsigned short comp_t
                /* "floating pt": 3 bits base 8 exp, 13 bits fraction */

struct acct
{
        char    ac_comm[10];    /* Accounting command name */
        comp_t  ac_utime;       /* Accounting user time */
        comp_t  ac_stime;       /* Accounting system time */
        comp_t  ac_etime;       /* Accounting elapsed time */
        time_t  ac_btime;       /* Beginning time */
        short   ac_uid;         /* Accounting user ID */
        short   ac_gid;         /* Accounting group ID */
        short   ac_mem;         /* average memory usage */
        comp_t  ac_io;          /* number of disk IO blocks */
        dev_t   ac_tty;         /* control typewriter */
        char    ac_flag;        /* Accounting flag */
};
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 super-user privileges */
```

If the process does an **exec(2)**, the first 10 characters of the filename appear in ac_comm. The accounting flag contains bits indicating whether **exec(2)** was ever accomplished, and whether the process ever had super-user privileges.

SEE ALSO

acct(2), sa(1)

NAME

a.out — assembler and link editor output

SYNOPSIS

```
#include <a.out.h>
```

DESCRIPTION

a.out is the output file of the assembler as(1) and the link editor ld(1). Both programs make a.out executable if there were no errors and no unresolved external references. Layout information as given in the include file for the PDP11 is:

```
struct  exec {                  /* a.out header */
        int     a_magic;        /* magic number */
        unsigned a_text;        /* size of text segment */
        unsigned a_data;        /* size of initialized data */
        unsigned a_bss;         /* size of unitialized data */
        unsigned a_syms;        /* size of symbol table */
        unsigned a_entry;       /* entry point */
        unsigned a_unused;      /* not used */
        unsigned a_flag;        /* relocation info stripped */
};

#define A_MAGIC1 0407           /* normal */
#define A_MAGIC2 0410           /* read-only text */
#define A_MAGIC3 0411           /* separated I&D */
#define A_MAGIC4 0405           /* overlay */

struct  nlist {                 /* symbol table entry */
        char    n_name[8];      /* symbol name */
        int     n_type;         /* type flag */
        unsigned n_value;       /* value */
};

                                /* values for type flag */
#define N_UNDF  0               /* undefined */

#define N_ABS   01              /* absolute */
#define N_TEXT  02              /* text symbol */
```

```
#define N_DATA    03        /* data symbol */
#define N_BSS     04        /* bss symbol */
#define N_TYPE    037
#define N_REG     024       /* register name */
#define N_FN      037       /* file name symbol */
#define N_EXT     040       /* external bit, or'ed in */
#define FORMAT    "%06o"    /* to print a value */
```

The file has four sections: a header, the program and data text, relocation information, and a symbol table (in that order). The last two may be empty if the program was loaded with the "-s" option of ld or if the symbols and relocation have been removed by **strip(1)**.

In the header the sizes of each section are given in bytes, but are even. The size of the header is not included in any of the other sizes.

When an a.out file is loaded into core for execution, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at 0 in the core image; the header is not loaded. If the magic number in the header is 0407_8, it indicates that the text segment is not to be write-protected and shared, so the data segment is immediately contiguous with the text segment. If the magic number is 0410_8, the data segment begins at the first 0 mod 8K byte boundary following the text segment, and the text segment is not writable by the program. If other processes are executing the same file, they will share the text segment. If the magic number is 411_8, the text segment is again pure, write-protected, and shared, and moreover instruction and data space are separated, the text and data segment both begin at location 0. If the magic number is 0405, the text segment is overlaid on an existing (0411 or 0405) text segment and the existing data segment is preserved.

The stack will occupy the highest possible locations in the core image: from 0177776_8 and growing downwards. The stack is automatically extended as required. The data segment is only extended as requested by **brk(2)**.

The start of the text segment in the file is 020_8; the start of the data segment is $020+S_t$ (the size of the text) the start of the relocation information is $020+S_t+S_d$; the start of the symbol table is $020+2\|(S_t+S_d)$ if the relocation information is present, $020+S_t+S_d$ if not.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file. Other flag values may occur if an assembly language program defines machine instructions.

If a symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader ld as the name of a common region whose size is indicated by the value of the symbol.

The value of a word in the text or data portions which is not a reference to an undefined external symbol is exactly that value which will appear in core when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation information for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

If relocation information is present, it amounts to one word per word of program text or initialized data. There is no relocation information if the "`relocation info stripped`" flag in the header is on.

Bits 3-1 of a relocation word indicate the segment referred to by the text or data word associated with the relocation word:

000	absolute number
002	reference to text segment
004	reference to initialized data
006	reference to uninitialized data (bss)
010	reference to undefined external symbol

Bit 0 of the relocation word indicates, if 1, that the reference is relative to the pc (`clr x`). If 0, the reference is to the actual symbol (`clr *$x`).

The remainder of the relocation word (bits 15-4) contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

SEE ALSO

as(1), ld(1), nm(1)

NAME

ar — archive (library) file format

SYNOPSIS

```
#include <ar.h>
```

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.

A file produced by ar has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

```
#define ARMAG 0177545
struct ar_hdr {
        char    ar_name[14];
        long    ar_date;
        char    ar_uid;
        char    ar_gid;
        int     ar_mode;
        long    ar_size;
};
```

The name is a null-terminated string; the date is in the form of **time(2)**; the user ID and group ID are numbers; the mode is a bit pattern per **chmod(2)**; the size is counted in bytes.

Each file begins on a word boundary; a null byte 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.

SEE ALSO

ar(1), ld(1), nm(1)

BUGS

Coding user and group IDs as characters is a botch.

NAME

core — format of core image file

DESCRIPTION

UNIX writes out a core image of a terminated process when any of various errors occur. See **signal(2)** for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called "core" and is written in the process's working directory (provided it can be; normal access controls apply).

The first 1024 bytes of the core image are a copy of the system's per-user data for the process, including the registers as they were at the time of the fault; see the system listings for the format of this area. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is write-protected and shared, it is not dumped; otherwise the entire address space is dumped.

In general the debugger **adb(1)** is sufficient to deal with core images.

SEE ALSO

adb(1), signal(2)

NAME

dir — format of directories

SYNOPSIS

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

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry see, **filsys(5)**. The structure of a directory entry as given in the include file is:

```
#ifndef DIRSIZ
#define DIRSIZ14
#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 and for the root directories of removable file systems. In the first case, there is no parent, and in the second, the system does not permit off-device references. Therefore in both cases ".." has the same meaning as ".".

SEE ALSO

filsys(5)

NAME

dump, ddate — incremental dump format

SYNOPSIS

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

DESCRIPTION

Tapes used by dump and **restor(1)** contain:

a header record
two groups of bit map records
a group of records describing directories
a group of records describing files

The format of the header record and of the first record of each description as given in the include file <dumprestor.h> is:

```
#define NTREC     8
#define MLEN      16
#define MSIZ      4096

#define TS_TAPE   1
#define TS_INODE  2
#define TS_BITS   3
#define TS_ADDR   4
#define TS_END    5
#define TS_CLRI   6
#define MAGIC     (int)60011
#define CHECKSUM  (int)84446

struct  spcl      {
        int       c_type;
        time_t    c_date;
        time_t    c_ddate;
        int       c_volume;
        daddr_t   c_tapea;
        ino_t     c_inumber;
```

```
        int     c_magic;
        int     c_checksum;
        struct  dinodec_dinode;
        int     c_count;
        char    c_addr[BSIZE];
} spcl;

struct idates   {
        char    id_name[16];
        char    id_incno;
        time_t  id_ddate;
};
```

NTREC is the number of 512 byte records in a physical tape block. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS_ entries are used in the c_type field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TAPE	Tape volume label
TS_INODE	A file or directory follows. The c_dinode field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS	A bit map follows. This bit map has a one bit for each inode that was dumped.
TS_ADDR	A subrecord of a file description. See c_addr below.
TS_END	End of tape record.
TS_CLRI	A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped.
MAGIC	All header records have this number in c_magic.
CHECKSUM	Header records checksum to this value.

The fields of the header structure are as follows:

c_type	The type of the header.
c_date	The date the dump was taken.
c_ddate	The date the file system was dumped from.
c_volume	The current volume number of the dump.
c_tapea	The current number of this (512-byte) record.
c_inumber	The number of the inode being dumped if this is of type TS_INODE.
c_magic	This contains the value MAGIC above, truncated as needed.
c_checksum	This contains whatever value is needed to make the record sum to CHECKSUM.
c_dinode	This is a copy of the inode as it appears on the file system; see **filsys(5)**.

c_count	The count of characters in c_addr.
c_addr	An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is nonzero. If the block was not present on the file system, no block was dumped; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, TS_ADDR records will be scattered through the file, each one picking up where the last left off.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a TS_END record and then the tapemark.

The structure idates describes an entry of the file /etc/ddate where dump history is kept. The fields of the structure are:

id_name	The dumped filesystem is "/dev/id_nam".
id_incno	The level number of the dump tape; see **dump(1)**.
id_ddate	The date of the incremental dump in system format see **types(5)**.

FILES

/etc/ddate

SEE ALSO

dump(1), dumpdir(1), restor(1), filsys(5), types(5)

NAME

environ — user environment

SYNOPSIS

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

DESCRIPTION

An array of strings called the "environment" is made available by **exec(2)** when a process begins. By convention these strings have the form "`name=value`". The following names are used by various commands:

`PATH` The sequence of directory prefixes that **sh**, **time**, **nice(1)**, etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by ":". **Login(1)** sets `PATH=:/bin:/usr/bin`.

`HOME` A user's login directory, set by **login(1)** from the password file **passwd(5)**.

`TERM` The kind of terminal for which output is to be prepared. This information is used by commands, such as **nroff** or **plot(1)**, which may exploit special terminal capabilities. See **term(7)** for a list of terminal types.

Further names may be placed in the environment by the `export` command and "`name=value`" arguments in **sh(1)**, or by **exec(2)**. It is unwise to conflict with certain Shell variables that are frequently exported by "`.profile`" files: `MAIL`, `PS1`, `PS2`, and `IFS`.

SEE ALSO

exec(2), sh(1), term(7), login(1)

NAME

filsys, flblk, ino — format of file system volume

SYNOPSIS

```
#include <sys/types.h>
#include <sys/flbk.h>
#include <sys/filsys.h>
#include <sys/ino.h>
```

DESCRIPTION

Every file system storage volume (e.g. RF disk, RK disk, RP disk, DECtape reel) has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the super block. The layout of the super block as defined by the include file <sys/filsys.h> is:

```
        /* structure of the super-block */

 struct filsys   {
        unsigned short s_isize;     /* size in blocks of i-list */
        daddr_t  s_fsize;           /* size in blocks of entire volume */
        short    s_nfree;           /* number of addresses in s_free */
        daddr_t  s_free[NICFREE];   /* free block list */
        short    s_ninode;          /* number of i-nodes in s_inode */
        ino_t    s_inode[NICINOD];  /* free i-node list */
        char     s_flock;           /* lock during free list manipulation */
        char     s_ilock;           /* lock during i-list manipulation */
        char     s_fmod;            /* super block modified flag */
        char     s_ronly;           /* mounted read-only flag */
        time_t   s_time;            /* last super block update */

        /* remainder not maintained by this version of the system */

        daddr_t  s_tfree;           /* total free blocks*/
        ino_t    s_tinode;          /* total free inodes */
        short    s_m;               /* interleave factor */
        short    s_n;               /* " " */
```

```
        char    s_fname[6];     /* file system name */
        char    s_fpack[6];     /* file system pack name */
};
```

s_isize is the address of the first block after the i-list, which starts just after the super-block, in block 2. Thus is i-list is s_isize-2 blocks long. s_fsize is the address of the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block addresses; if an 'impossible' block address 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 NICFREE free block numbers. NICFREE is a configuration constant. s_free[0] is the block address of the head of a chain of blocks constituting the free list. The layout of each block of the free chain as defined in the include file <sys/fblk.h> is:

```
struct fblk
{
        int     df_nfree;
        daddr_t df_free[NICFREE];
};
```

The fields df_nfree and df_free in a free block are used exactly like s_nfree and s_free in the super block. To allocate a block: decrement s_nfree, and the new block number is s_free[s_nfree]. If the new block address is 0, there are no blocks left, so give an error. If s_nfree became 0, read the new block into s_nfree and s_free. To free a block, check if s_nfree is NICFREE; 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 address and increment s_nfree.

s_ninode is the number of free i-numbers in the s_inode array. To allocate an i-node: if s_ninode is greater than 0, decrement it and return s_inode[s_ninode]. If it was 0, read the i-list and place the numbers of all free inodes (up to NICINOD) into the s_inode array, then try again. To free an i-node, provided s_ninode is less than NICINODE, place its number into s_inode[s_ninode] and increment s_ninode. If s_ninode is already NICINODE, don't bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

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 write-protection indicator; its disk value is also immaterial.

s_time is the last time the super-block of the file system was changed. During a reboot, s_time of the super-block for the root file system is used to set the system's idea of the time.

The fields s_tfree, s_tinode, s_fname and s_fpack are not currently maintained.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. I-nodes are 64 bytes long, so 8 of them fit into a block. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an i-node as given in the include file <sys/ino.h> is:

```
/* Inode structure as it appears on a disk block.  */
struct dinode
{
unsigned short di_mode; /* mode and type of file */
        short     di_nlink;       /* number of links to file */
        short     di_uid;         /* owner's user id */
        short     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 */
};
#define INOPB 8 /* 8 inodes per block */
/* the 40 address bytes:
 * 39 used; 13 addresses of 3 bytes each */
```

di_mode tells the kind of file; it is encoded identically to the st_mode field of stat(2). di_nlink is the number of directory entries (links) that refer to this i-node. di_uid and di_gid are the owner's user and group IDs. size is the number of bytes in the file. di_atime and di_mtime are the times of last access and modification of the file contents (read, write or create) (see times(2)); di_ctime records the time of last modification to the inode or to the file, and is used to determine whether it should be dumped.

Special files are recognized by their modes and not by i-number. A block-type special file is one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character oriented. For special files, the di_addr field is occupied by the device code (see types(5)). The device codes of block and character special files overlap.

Disk addresses of plain files and directories are kept in the array di_addr packed

into 3 bytes each. The first 10 addresses specify device blocks directly. The last 3 addresses are singly, doubly, and triply indirect and point to blocks of 128 block pointers. Pointers in indirect blocks have the type daddr_t (see **types(5)**).

For block b in a file to exist, it is not necessary that all blocks less than b exist. A zero block number either in the address words of the i-node or in an indirect block indicates that the corresponding block has never been allocated. Such a missing block reads as if it contained all zero words.

SEE ALSO

icheck(1), dcheck(1), dir(5), mount(1), stat(2), types(5)

NAME

group — group file

DESCRIPTION

group contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; Each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in 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 ID's to names.

FILES

/etc/group

SEE ALSO

newgrp(1), crypt(3), passwd(1), passwd(5)

NAME

mpxio — multiplexed i/o

SYNOPSIS

```
#include <sys/mx.h>
#include <sgtty.h>
```

DESCRIPTION

Data transfers on mpx files (see **mpx(2)**) are multiplexed by imposing a record structure on the io stream. Each record represents data from/to a particular channel or a control or status message associated with a particular channel.

The prototypical data record read from an mpx file is as follows

```
struct input_record {
        short   index;
        short   count;
        short   ccount;
        char    data[];
};
```

where `index` identifies the channel, and `count` specifies the number of characters in `data`. If `count` is zero, `ccount` gives the size of `data`, and the record is a control or status message. Although `count` or `ccount` might be odd, the operating system aligns records on short (i.e. 16-bit) boundaries by skipping bytes when necessary.

Data written to an mpx file must be formatted as an array of record structures defined as follows

```
struct output_record {
        short   index;
        short   count;
        short   ccount;
        char    *data;
};
```

where the data portion of the record is referred to indirectly and the other cells have the same interpretation as in `input_record`.

The control messages listed below may be read from a multiplexed file descriptor. They are presented as two 16-bit integers: the first number is the message code (defined in <sys/mx.h>), the second is an optional parameter meaningful only with M_WATCH and M_BLK.

M_WATCH — a process 'wants to attach' on this channel. The second parameter is the 16-bit user-id of the process that executed the open.

M_CLOSE — the channel is closed. This message is generated when the last file descriptor referencing a channel is closed. The `detach` command (see **mpx(2)** should be used in response to this message.

M_EOT — indicates logical end of file on a channel. If the channel is joined to a typewriter, EOT (control-d) will cause the M_EOT message under the conditions specified in **tty(4)** for end of file. If the channel is attached to a process, M_EOT will be generated whenever the process writes zero bytes on the channel.

M_BLK — if non-blocking mode has been enabled on an mpx file descriptor `xd` by executing `ioctl(xd, MXNBLK, 0)`, write operations on the file are truncated in the kernel when internal queues become full. This is done on a per-channel basis: the parameter is a count of the number of characters not transferred to the channel on which M_BLK is received.

M_UBLK — is generated for a channel after M_BLK when the internal queues have drained below a threshold.

Two other messages may be generated by the kernel. As with other messages, the first 16-bit quantity is the message code.

M_OPEN — is generated in conjunction with 'listener' mode (see `mpx(2)`). The uid of the calling process follows the message code as with M_WATCH. This is followed by a null-terminated string which is the name of the file being opened. M_IOCTL — is generated for a channel connected to a process when that process executes the `ioctl(fd, cmd, &vec)` call on the channel file descriptor. The M_IOCTL code is followed by the `cmd` argument given to `ioctl` followed by the contents of the structure `vec`. It is assumed, not needing a better compromise at this time, that the length of `vec` is determined by `sizeof (struct sgttyb)` as declared in <sgtty.h>.

Two control messages are understood by the operating system. M_EOT may be sent through an mpx file to a channel. It is equivalent to propagating a zero-length record through the channel; i.e. the channel is allowed to drain and the process or device at the other end receives a zero-length transfer before data starts flowing through the channel again. M_IOCTL can also be sent through a channel. The format is identical to that described above.

NAME

mtab — mounted file system table

DESCRIPTION

`mtab` resides in directory `/etc` and contains a table of devices mounted by the `mount` command. `umount` removes entries.

Each entry is 64 bytes long; the first 32 are the null-padded name of the place where the special file is mounted; the second 32 are the null-padded name of the special file. The special file has all its directories stripped away; that is, everything through the last / is thrown away.

This table is present only so people can look at it. It does not matter to `mount` if there are duplicated entries nor to `umount` if a name cannot be found.

FILES

`/etc/mtab`

SEE ALSO

mount(1)

NAME

passwd — password file

DESCRIPTION

passwd contains for each user the following information:

name (login name, contains no upper case)
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 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 ID's to names.

FILES

`/etc/passwd`

SEE ALSO

getpwent(3), login(1), crypt(3), passwd(1), group(5)

NAME

plot — graphics interface

DESCRIPTION

Files of this format are produced by routines described in **plot(3)**, and are interpreted for various devices by commands described in **plot(1)**. 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 four 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(3)**.

- `m` move: The next four bytes give a new current point.
- `n` cont: Draw a line from the current point to the point given by the next four bytes. See **plot(1)**.
- `p` point: Plot the point given by the next four bytes.
- `l` line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.
- `t` label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a newline.
- `a` arc: The first four bytes give the center, the next four give the starting point, and the last four give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise.
- `c` circle: The first four bytes give the center of the circle, the next two the radius.
- `e` erase: Start another frame of output.
- `f` linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are 'dotted,' 'solid,' 'longdashed,' 'shortdashed,' and 'dotdashed.' Effective only in **plot 4014** and `plot ver`.
- `s` space: The next four bytes give the lower left corner of the plotting area; the following four 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 **plot(1)**. 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 isn't square.

```
4014        space(0, 0, 3120, 3120);
```

```
ver         space(0, 0, 2048, 2048);
300, 300s   space(0, 0, 4096, 4096);
450         space(0, 0, 4096, 4096);
```

SEE ALSO

plot(1), plot(3), graph(1)

NAME

tp — DEC/mag tape formats

DESCRIPTION

The command tp dumps files to and extracts files from DECtape and magtape. The formats of these tapes are the same except that magtapes have larger directories. Block zero contains a copy of a stand-alone bootstrap program. See **bproc(8)**. Blocks 1 through 24 for DECtape (1 through 62 for magtape) contain a directory of the tape. There are 192 (resp. 496) entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

```
struct {
        char    pathname[32];
        int     mode;
        char    uid;
        char    gid;
        char    unused1;
        char    size[3];
        long    modtime;
        int     tapeaddr;
        char    unused2[16];
        int     checksum;
};
```

The path name entry is the path name of the file when put on the tape. If the pathname starts with a zero word, the entry is empty. It is at most 32 bytes long and ends in a null byte. Mode, uid, gid, size and time modified are the same as described under i-nodes (see file system **filsys(5)**). The tape address is the tape block number of the start of the contents of the file. Every file starts on a block boundary. The file occupies (size+511)/512 blocks of continuous tape. The checksum entry has a value such that the sum of the 32 words of the directory entry is zero.
Blocks above 25 (resp. 63) are available for file storage.
A fake entry has a size of zero.

SEE ALSO

filsys(5), tp(1)

BUGS

The `pathname`, `uid`, `gid`, and `size` fields are too small.

NAME

ttys — terminal initialization data

DESCRIPTION

The `ttys` file is read by the `init` program and specifies which terminal special files are to have a process created for them which will allow people to log in. It contains one line per special file.

The first character of a line is either "0" or "1"; the former causes the line to be ignored, the latter causes it to be effective. The second character is used as an argument to **getty(8)**, which performs such tasks as baud-rate recognition, reading the login name, and calling `login`. For normal lines, the character is "0"; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. (Getty will have to be fixed in such cases.) The remainder of the line is the terminal's entry in the device directory, /dev.

FILES

`/etc/ttys`

SEE ALSO

init(8), getty(8), login(1)

NAME

types — primitive system data types

SYNOPSIS

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

DESCRIPTION

The data types defined in the include file are used in UNIX system code; some data of these types are accessible to user code:

```
typedef    long            daddr_t;      /* disk address */
typedef    char *          caddr_t;      /* core address */
typedef    unsigned int    ino_t;        /* i-node number */
typedef    long            time_t;       /* a time */
typedef    int             label_t[6];   /* program status */
typedef    int             dev_t;        /* device code */
typedef    long            off_t;        /* offset in file */
        /* selectors and constructor for device code */
#define    major(x)        (int)(((unsigned)x>>8))
#define    minor(x)        (int)(x&0377)
#define    makedev(x,y)    (dev_t)((x)<<8|(y))
```

The form `daddr_t` is used for disk addresses except in an inode on disk, see **filsys(5)**. 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

filsys(5), time(2), lseek(2), adb(1)

NAME

utmp, wtmp — login records

SYNOPSIS

```
#include <utmp.h>
```

DESCRIPTION

The utmp file allows one to discover information about who is currently using UNIX. The file is a sequence of entries with the following structure declared in the include file:

```
struct utmp {
        char ut_line[8];    /* tty name */
        char ut_name[8];    /* user id */
        long ut_time;       /* time on */
};
```

This structure gives the name of the special file associated with the user's terminal, the user"s login name, and the time of the login in the form of **time(2)**.

The wtmp file records all logins and logouts. Its format is exactly like utmp except that a null user name indicates a logout on the associated terminal. Furthermore, the terminal name "~" indicates that the system was rebooted at the indicated time; the adjacent pair of entries with terminal names "|" and "}" indicate the system-maintained time just before and just after a date command has changed the system's idea of the time.

Wtmp is maintained by **login(1)** and **init(8)**. Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by **ac(1)**.

FILES

```
/etc/utmp
/usr/adm/wtmp
```

SEE ALSO

login(1), init(8), who(1), ac(1)

NAME

arithmetic — provide drill in number facts

SYNOPSIS

`/usr/games/arithmetic [ +-x/ ] [ range ]`

DESCRIPTION

`arithmetic` types out simple arithmetic problems, and waits for an answer to be typed in. If the answer is correct, it types back "Right!", and a new problem. If the answer is wrong, it replies "What?", and waits for another answer. Every twenty problems, it publishes statistics on correctness and the time required to answer.

To quit the program, type an interrupt (delete).

The first optional argument determines the kind of problem to be generated. `+ - x /` respectively cause addition, subtraction, multiplication, and division problems to be generated. One or more characters can be given; if more than one is given, the different types of problems will be mixed in random order; default is `+ -`

range is a decimal number; all addends, subtrahends, differences, multiplicands, divisors, and quotients will be less than or equal to the value of *range*. Default *range* is 10.

At the start, all numbers less than or equal to *range* are equally likely to appear. If the respondent makes a mistake, the numbers in the problem which was missed become more likely to reappear.

As a matter of educational philosophy, the program will not give correct answers, since the learner should, in principle, be able to calculate them. Thus the program is intended to provide drill for someone just past the first learning stage, not to teach number facts de novo. For almost all users, the relevant statistic should be time per problem, not percent correct.

NAME

backgammon — the game

SYNOPSIS

`/usr/games/backgammon`

DESCRIPTION

This program does what you expect. It will ask whether you need instructions.

NAME

quiz — test your knowledge

SYNOPSIS

`/usr/games/quiz [ -i` *file* `] [ -t ] [` *category1 category2* `]`

DESCRIPTION

`quiz` gives associative knowledge tests on various subjects. It asks items chosen from *category1* and expects answers from *category2*. If no categories are specified, `quiz` gives instructions and lists the available categories.

`quiz` tells a correct answer whenever you type a bare newline. At the end of input, upon interrupt, or when questions run out, `quiz` reports a score and terminates.

The `-t` flag specifies "`tutorial`" mode, where missed questions are repeated later, and material is gradually introduced as you learn.

The `-i` flag causes the named file to be substituted for the default index file. The lines of these files have the syntax:

```
line      = category newline | category ":" line
category  = alternate | category "|" alternate
alternate = empty | alternate primary
primary   = character | "[" category "]" | option
option    = "{" category "}"
```

The first category on each line of an index file names an information file. The remaining categories specify the order and contents of the data in each line of the information file. Information files have the same syntax. Backslash "\" is used as with **sh(1)** to quote syntactically significant characters or to insert transparent newlines into a line. When either a question or its answer is empty, `quiz` will refrain from asking it.

FILES

`/usr/games/quiz.k/*`

BUGS

The construct "`a|ab`" doesn't work in an information file. Use "`a{b}`".

NAME

wump — the game of hunt-the-wumpus

SYNOPSIS

`/usr/games/wump`

DESCRIPTION

wump plays the game of "`Hunt the Wumpus`." A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

The program asks various questions which you answer one per line; it will give a more detailed description if you want.

This program is based on one described in *People's Computer Company*, *2*, 2 (November 1973).

BUGS

It will never replace Space War.

NAME

ascii — map of ASCII character set

SYNOPSIS

```
cat /usr/pub/ascii
```

DESCRIPTION

ascii is a map of the ASCII character set, to be printed as needed. It contains:

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 Ø	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

FILES

/usr/pub/ascii

NAME

eqnchar — special character definitions for eqn

SYNOPSIS

`eqn /usr/pub/eqnchar [ *files* ] | troff [ *options* ]`

`neqn /usr/pub/eqnchar [ *files* ] | nroff [ *options* ]`

DESCRIPTION

eqnchar contains **troff** and **nroff** character definitions for constructing characters that are not available on the Graphic Systems typesetter. These definitions are primarily intended for use with eqn and neqn. It contains definitions for the following characters

ciplus	⊕	||	‖	square	□
citimes	⊗	langle	⟨	circle	∘
wig	∼	rangle	⟩	blot	■
-wig	≃	hbar	ℏ	bullet	•
>wig	≳	ppd	⊥	prop.	prop
<wig	≲	<·>	↔	empty	∅
=wig	≅	<=>	⇔	member	∈
star	*	|	∤	nomem	∉
bigstar	bigstar	|>	⊳	cup	∪
=dot	≐	ang	∠	cap	∩
orsign	∨	rang	rang	incl	⊆
andsign	∧	3dot	3dot	subset	⊂
=del	≜	thf	thf	supset	⊃
oppA	∀	quarter	1/4	!subset	⊄
oppE	∃	3quarter	3/4	!supset	⊅
angstrom	Å	degree	°		

FILES

/usr/pub/eqnchar

SEE ALSO

troff(1), eqn(1)

NAME

greek — graphics for extended TTY-37 type-box

SYNOPSIS

`cat /usr/pub/greek [ | greek -T`*terminal* `]`

DESCRIPTION

greek gives the mapping from ascii to the 'shift out' graphics in effect between SO and SI on model 37 Teletypes with a 128-character type-box. These are the default greek characters produced by **nroff**. The filters of **greek(1)** attempt to prints them on various other terminals. The file contains:

alpha	α	A	beta	β	B	gamma	γ	\
GAMMA	Γ	G	delta	δ	D	DELTA	Δ	W
epsilon	ϵ	S	zeta	ζ	Q	eta	η	N
THETA	Θ	T	theta	θ	O	labda	λ	L
xi	ξ	X	pi	π	J	PI	Π	P
rho	ρ	K	sigma	σ	Y	SIGMA	Σ	R
tau	τ	I	phi	ϕ	U	PHI	Φ	F
psi	ψ	V	PSI	Ψ	H	omega	ω	C
OMEGA	Ω	Z	nabla	∇	[	not	$\neg$	_
partial	∂	]	integral	$\int$	^			

SEE ALSO

troff(1), greek(1)

NAME

hier — file system hierarchy

DESCRIPTION

The following outline gives a quick tour through a representative directory hierarchy.

```
/       root
/dev/   devices (4)
        console
                main console, tty(4)
        tty*    terminals, tty(4)
        cat     phototypesetter cat(4)
        rp*     disks, rp, hp(4)
        rrp*    raw disks, rp, hp(4)
        ...
/bin/   utility programs, cf /usr/bin/ (1)
        as      assembler first pass, cf /usr/lib/as2
        cc      C compiler executive, cf /usr/lib/c[012]
        ...
/lib/   object libraries and other stuff, cf /usr/lib/
        libc.a  system calls, standard I/O, etc. (2,3,3S)
        libm.a  math routines (3M)
        libplot.a
                plotting routines, plot(3)
        libF77.a
                Fortran runtime support
        libI77.a
                Fortran I/O
        ...
        as2     second pass of as(1)
        c[012]  passes of cc(1)
        ...
/etc/   essential data and dangerous maintenance utilities
        passwd  password file, passwd(5)
        group   group file, group(5)
        motd    message of the day, login(1)
        mtab    mounted file table, mtab(5)
        ddate   dump history, dump(1)
```

```
        ttys    properties of terminals, ttys(5)
        getty   part of login, getty(8)
        init    the father of all processes, init(8)
        rc      shell program to bring the system up
        cron    the clock daemon, cron(8)
        mount   mount(1)
        wall    wall(1)
        ...
/tmp/   temporary files, usually on a fast device, cf /usr/tmp/
        e*      used by ed(1)
        ctm*    used by cc(1)
        ...
/usr/   general-pupose directory, usually a mounted file system
        adm/    administrative information
                wtmp    login history, utmp(5)
                messageshardware error messages
                tracct  phototypesetter accounting, troff(1)
                vpacct  line printer accounting lpr(1)
/usr /bin
        utility programs, to keep /bin/ small
        tmp/    temporaries, to keep /tmp/ small
                stm*    used by sort(1)
                raster  used by plot(1)
        dict/
                word    lists, etc.
                words   principal word list, used by look(1)
                spellhist
                        history file for spell(1)
        games/
                bj      blackjack
                hangman
                quiz.k/ what quiz(6) knows
                        index   category index
                        africa  countries and capitals
                        ...
                ...
        include/
                standard #include files
                a.out.h object file layout, a.out(5)
                stdio.h standard I/O, stdio(3)
                math.h  (3M)
                ...
                sys/    system-defined layouts, cf /usr/sys/h
```

```
                acct.h  process accounts, acct(5)
                buf.h   internal system buffers
                ...
        lib/    object libraries and stuff, to keep /lib/ small
                lint[12]
                        subprocesses for lint(1)
                llib-lc dummy declarations for /lib/libc.a, used by lint(1)
                llib-lm dummy declarations for /lib/libc.m
                atrun   scheduler for at(1)
                struct/ passes of struct(1)
                ...
                tmac/   macros for troff(1)
                        tmac.an macros for man(7)
                        tmac.s  macros for ms(7)
                        ...
                font/   fonts for troff(1)
                        R       Times Roman
                        B       Times Bold
                        ...
                uucp/   programs and data for uucp(1)
                        L.sys   remote system names and numbers
                        uucico  the real copy program
                        ...
                suftab  table of suffixes for hyphenation, used by troff(1)
                units   conversion tables for units(1)
                eign    list of English words to be ignored by ptx(1)
/usr/man/
        volume 1 of this manual, man(1)
                man0/   general
                        intro   introduction to volume 1, ms(7) format
                        xx      template for manual page
                man1/   chapter 1
                        as.1
                        mount.1m
                        ...
                cat1/   preprinted pages for man1/
                        as.1
                        mount.1m
                ...
        spool/  delayed execution files
                at/     used by at(1)
                lpd/    used by lpr(1) lock present when line printer is active
                        cf*     copy of file to be printed, if necessary
```

```
                df*     daemon control file, lpd(8)
                tf*     transient control file, while lpr is working
        uucp/   work files and staging area for uucp(1)
                LOGFILE summary log
                LOG.*   log file for one transaction
mail/   mailboxes for mail(1)
        uid     mail file for user uid
        uid.lockdock file while uid is receiving mail
wd      initial working directory of a user, typically wd is the user's login
        name
        .profile
                set environment for sh(1), environ(5)
        calendar
                user's datebook for calendar(1)
doc/    papers, mostly in volume 2 of this manual, typically in ms(7) format
        as/     assembler manual
        c       C manual
        ....
sys/    system source
        dev/    device drivers
                bio.c   common code
                cat.c   cat(4)
                dh.c    DH11, tty(4)
                tty     tty(4)
                ...
        conf/   hardware-dependent code
                mch.s   assembly language portion
                conf    configuration generator
                ...
        h/      header (include) files
                acct.h  acct(5)
                stat.h  stat(2)
                ...
        sys/    source for system proper
                main.c
                pipe.c
                sysent.c
                        system entry points
                ...
/usr/   src/ source programs for utilities, etc.
                cmd/    source of commands
                        as/     assembler
                                makefile
```

```
                recipe for rebuilding the assembler
        as1?.s  source of pass1
    ar.c    source for ar(1)
    ...
    troff/  source for nroff and troff(1)
        nmake   makefile for nroff
        tmake   makefile for troff
        font/   source for font tables, /usr/lib/font/
            ftR.c   Roman
            ...
        term/   terminal characteristics tables,
            /usr/lib/term/
            tab300.c
                DASI 300
            ...
        ...
libc/   source for functions in /lib/libc.a
    crt/    C runtime support
        ldiv.s  division into a long
        lmul.s  multiplication to produce long
        ...
    csu/    startup and wrapup routines needed with every C
        program
        crt0.s  regular startup
        mcrt0.s modified startup for cc -p
    sys/    system calls (2)
        access.s
        alarm.s
        ...
    stdio/  standard I/O functions (3S)
        fgets.c
        fopen.c
        ...
    gen/    other functions in (3)
        abs.c
        atof.c
        ...
    compall shell procedure to compile libc
    mklib   shell procedure to make /lib/libc.a
libI77/ source for /lib/libI77
libF77/
...
games/  source for /usr/games
```

SEE ALSO

ls(1), ncheck(1), find(1), grep(1)

BUGS

The position of files is subject to change without notice.

NAME

man — macros to typeset manual

SYNOPSIS

`nroff -man` *file* ...

`troff -man` *file* ...

DESCRIPTION

These macros are used to lay out manual pages. A skeleton page may be found in the file `/usr/man/man0/xx`.

Any text argument *t* may be zero to six words. Quotes may be used to include blanks in a 'word'. If *text* is empty, the special treatment is applied to the next input line with text to be printed. In this way .I may be used to italicize a whole line, or .SM followed by .B to make small bold letters.

A prevailing indent distance is remembered between successive indented paragraphs, and is reset to default value upon reaching a non-indented paragraph. Default units for indents *i* are ens.

Type font and size are reset to default values before each paragraph, and after processing font and size setting macros.

These strings are predefined by `-man`:

*S Change to default type size.

FILES

`/usr/lib/tmac/tmac.an`
`/usr/man/man0/xx`

SEE ALSO

troff(1), man(1)

BUGS

Relative indents don't nest.

REQUESTS

Request	Cause Break	If no Argument	Explanation
.B *t*	no	*t*=*n*.t.l.*	Text *t* is bold.
.BI *t*	no	*t*=*n*.t.l.	Join words of *t* alternating bold and italic.
.BR *t*	no	*t*=*n*.t.l.	Join words of *t* alternating bold and Roman.
.DT	no	.5i 1i...	Restore default tabs.
.HP *i*	yes	*i*=*p*.i.*	Set prevailing indent to *i*. Begin paragraph with hanging indent.
.I *t*	no	*t*=*n*.t.l.	Text *t* is italic.
.IB *t*	no	*t*=*n*.t.l.	Join words of *t* alternating italic and bold.
.IP *x i*	yes	*x*=	Same as .TP with tag *x*.
.IR *t*	no	*t*=*n*.t.l.	Join words of *t* alternating italic and Roman.
.LP	yes	-	Same as .PP.
.PD *d*	no	*d*=*4v*	Interparagraph distance is *d*.
.PP	yes	-	Begin paragraph. Set prevailing indent to .5i.
.RE	yes	-	End of relative indent. Set prevailing indent to amount of starting .RS.
.RB *t*	no	*t*=*n*.t.l.	Join words of *t* alternating Roman and bold.
.RI *t*	no	*t*=*n*.t.l.	Join words of *t* alternating Roman and italic.
.RS *i*	yes	*i*=*p*.i.	Start relative indent, move left margin in distance *i*. Set prevailing indent to .5i for nested indents.
.SH *t*	yes	*t*=*n*.t.l.	Subhead.
.SM *t*	no	*t*=n.t.l.	Text *t* is small.
.TH *n c x*	yes	-	Begin page named *n* of chapter *c*; *x* is extra commentary, e.g. 'local', for page foot. Set prevailing indent and tabs to .5i.
.TP *i*	yes	*i*=p.i.	Set prevailing indent to *i*. Begin indented paragraph with hanging tag given by next text line. If tag doesn't fit, place it on separate line.

* n.t.l.= next text line; p.i. = prevailing indent

NAME

ms — macros for formatting manuscripts

SYNOPSIS

`nroff -ms [` *options* `]` *file* ... `troff -ms [` *options* `]` *file* ...

DESCRIPTION

This package of *nroff* and *troff* macro definitions provides a canned formatting facility for technical papers in various formats. When producing 2-column output on a terminal, filter the output through col(1).

The macro requests are defined below. Many `nroff` and `troff` requests are unsafe in conjunction with this package, however these requests may be used with impunity after the first .PP:

`.bp`	begin new page
`.br`	break output line here
`.sp n`	insert n spacing lines
`.ls n`	(line spacing) n=1 single, n=2 double space
`.na`	no alignment of right margin

Output of the **eqn**, **neqn**, **refer**, and tbl(1) preprocessors for equations and tables is acceptable as input.

FILES

`/usr/lib/tmac/tmac.s`

SEE ALSO

eqn(1), troff(1), refer(1), tbl(1)

REQUESTS

Request	Initial Value	Cause Break	Explanation
.1C	yes	yes	One column format on a new page.
.2C	no	yes	Two column format.
.AB	no	yes	Begin abstract.
.AE	-	yes	End abstract.
.AI	no	yes	Author's institution follows. Suppressed in TM.
.AT	no	yes	Print 'Attached' and turn off line filling.
.AU *x y*	no	yes	Author's name follows. *x* is location and *y* is extension, ignored except in TM.
.B *x*	no	no	Print *x* in boldface; if no argument switch to boldface.
.B1	no	yes	Begin text to be enclosed in a box.
.B2	no	yes	End text to be boxed . print it.
.BT	date	no	Bottom title, automatically invoked at foot of page. May be redefined.
.BX *x*	no	no	Print *x* in a box.
.CS *x...*	-	yes	Cover sheet info if TM format, suppressed otherwise. Arguments are number of text pages, other pages, total pages, figures, tables, references.
.CT	no	yes	Print 'Copies to' and enter no-fill mode.
.DA *x*	nroff	no	'Date line' at bottom of page is *x*. Default is today.
.DE	-	yes	End displayed text. Implies .KE.
.DS *x*	no	yes	Start of displayed text, to appear verbatim line-by-line. *x*=I for indented display (default), *x*=L for left-justified on the page, *x*=C for centered, *x*=B for make left-justified block, then center whole block. Implies .KS.
.EG	no	-	Print document in BTL format for 'Engineer's Notes.' Must be first.
.EN	-	yes	Space after equation produced by eqn or neqn.
.EQ *x y*	-	yes	Precede equation; break out and add space. Equation number is *y*. The optional argument *x* may be *I* to indent equation (default), *L* to left-adjust the equation, or *C* to center the equation.
.FE	-	yes	End footnote.

.FS	no	no	Start footnote. The note will be moved to the bottom of the page.
.HD	-	no	'Bell Laboratories, Holmdel, New Jersey 07733'.
.I *x*	no	no	Italicize *x*; if *x* missing, italic text follows.
.IH	no	no	'Bell Laboratories, Naperville, Illinois 60540'
.IM	no	no	Print document in BTL format for an internal memorandum. Must be first.
.IP *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.KE	-	yes	End keep. Put kept text on next page if not enough room.
.KF	no	yes	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
.KS	no	yes	Start keeping following text.
.LG	no	no	Make letters larger.
.LP	yes	yes	Start left-blocked paragraph.
.MF	-	-	Print document in BTL format for 'Memorandum for File.' Must be first.
.MH	-	no	'Bell Laboratories, Murray Hill, New Jersey 07974'.
.MR	-	-	Print document in BTL format for 'Memorandum for Record.' Must be first.
.ND *date*	troff	no	Use date supplied (if any) only in special BTL format positions; omit from page footer.
.NH *n*	-	yes	Same as .SH, with section number supplied automatically. Numbers are multilevel, like 1.2.3, where *n* tells what level is wanted (default is 1).
.NL	yes	no	Make letters normal size.
.OK	-	yes	'Other keywords' for TM cover sheet follow.
.PP	no	yes	Begin paragraph. First line indented.
.PT	pg #	-	Page title, automatically invoked at top of page. May be redefined.
.PY	-	no	'Bell Laboratories, Piscataway, New Jersey 08854'
.QE	-	yes	End quoted (indented and shorter) material.
.QP	-	yes	Begin single paragraph which is indented and shorter.
.QS	-	yes	Begin quoted (indented and shorter) material.
.R	yes	no	Roman text follows.
.RE	-	yes	End relative indent level.
.RP	no	-	Cover sheet and first page for released paper. Must

			precede other requests.
.RS	-	yes	Start level of relative indentation. Following .IP's are measured from current indentation.
.SG *x*	no	yes	Insert signature(s) of author(s), ignored except in TM. *x* is the reference line (initials of author and typist).
.SH	-	yes	Section head follows, font automatically bold.
.SM	no	no	Make letters smaller.
.TA *x*...	5...	no	Set tabs in ens. Default is 5 10 15 ...
.TE	-	yes	End table.
.TH	-	yes	End heading section of table.
.TL	no	yes	Title follows.
.TM *x*...	no	-	Print document in BTL technical memorandum format. Arguments are TM number, (quoted list of) case number(s), and file number. Must precede other requests.
.TR *x*	-	-	Print in BTL technical report format; report number is *x*. Must be first.
.TS *x*	-	yes	Begin table; if *x* is *H* table has repeated heading.
.UL *x*	-	no	Underline argument (even in troff).
.UX	-	no	'UNIX'; first time used, add footnote 'UNIX is a trademark of Bell Laboratories.'
.WH	-	no	'Bell Laboratories, Whippany, New Jersey 07981'.

NAME

terminals — conventional names

DESCRIPTION

These names are used by certain commands and are maintained as part of the shell environment (see **sh(1),environ(5)**).

1620	DIABLO 1620 (and others using HyType II)
1620-12	same, in 12-pitch mode
300	DASI/DTC/GSI 300 (and others using HyType I)
300-12	same, in 12-pitch mode
300s	DASI/DTC 300/S
300s-12	same, in 12-pitch mode
33	TELETYPE Model 33
37	TELETYPE Model 37
40-2	TELETYPE Model 40/2
43	TELETYPE Model 43
450	DASI 450 (same as Diablo 1620)
450-12	same, in 12-pitch mode
450-12-8	same, in 12-pitch, 8 lines/inch mode
735	Texas Instruments TI735 (and TI725)
745	Texas Instruments TI745
dumb	terminals with no special features
hp	Hewlett-Packard HP264? series terminals
4014	Tektronix 4014
tn1200	General Electric TermiNet 1200
tn300	General Electric TermiNet 300
vt05	Digital Equipment Corp. VT05

Commands whose behavior may depend on the terminal accept arguments of the form *-Tterm*, where *term* is one of the names given above. If no such argument is present, a command may consult the shell environment for the terminal type.

SEE ALSO

stty(1), tabs(1), plot(1), sh(1), environ(5) troff(1) for `nroff`

BUGS

The programs that ought to adhere to this nomenclature do so only fitfully.

NAME

boot — standalone program loading and startup procedures

DESCRIPTION

`boot` is a relocating standalone loader for the PLEXUS environment. It may be invoked either from a system distribution tape or a formatted and initialized disk. `boot` will load type 407, 410, and 411 files. If the files contain relocation information the program will be relocated to 2000 hex, and the prom will remain in the low address space. If the file contains no relocation information it will be loaded at address zero. See standalone(8sa) for more information about the standalone environment.

`boot` sets up memory management, relocates itself into high memory, and types a ':' on the console. Then it reads from the console a device specification. `boot` finds the corresponding file on the given device, loads that file into memory location zero, sets up memory management as required, and calls the program. Control-h (erase) and delete/rubout (kill) line editing characters can be used.

Conventionally, the name of the current version of the system is "`/plex`". Then, the recipe is:

1) Get into the resident prom routines by reset or power off/on.

2) Type <cr> (return key) to load boot.

3) When the prompt is given, type a newline.

When the system is running, it types a '#' prompt. After doing any file system checks and setting the date (**date(1)**) a multi-user system is brought up by typing an EOT (control-d) in response to the '#' prompt.

FILES

`/plex` — system code

SEE ALSO

init(8)

NAME

crash — what to do when the system crashes

DESCRIPTION

This section gives at least a few clues about how to proceed if the system crashes. It can't pretend to be complete.

Bringing it back up. In restarting after a crash, always bring up the system single-user. This is accomplished by following the directions in **boot(8)** as modified for your particular installation; a single-user system is the shell spawned as the system starts executing. When it is running, perform a **dcheck** and **icheck(1)** on all file systems which could have been in use at the time of the crash. If any serious file system problems are found, they should be repaired. When you are satisfied with the health of your disks, check and set the date if necessary, then come up multi-user. This is most easily accomplished by logging out (typing an EOT).

To boot UNIX, three files (and the directories leading to them) must be intact. First, the initialization program `/etc/init` must be present and executable. If it is not, the CPU will loop in user mode. For `init` to work correctly, `/dev/console` and `/bin/sh` must be present. If either does not exist, the symptom is best described as thrashing. `Init` will go into a `fork/exec` loop trying to create a Shell with proper standard input and output.

If you cannot get the system to boot, a runnable system must be obtained from a backup medium. The root file system may then be doctored as a mounted file system as described below. If there are any problems with the root file system, it is probably prudent to go to a backup system to avoid working on a mounted file system.

Repairing disks. The first rule to keep in mind is that an addled disk should be treated gently; it shouldn't be mounted unless necessary, and if it is very valuable yet in quite bad shape, perhaps it should be dumped before trying surgery on it. This is an area where experience and informed courage count for much.

The problems reported by `icheck` typically fall into two kinds. There can be problems with the free list: duplicates in the free list, or free blocks also in files. These can be cured easily with an `icheck -s`. If the same block appears in more than one file or if a file contains bad blocks, the files should be deleted, and the free list reconstructed. The best way to delete such a file is to use **clri(1)**, then remove its directory entries. If any of the affected files is really precious, you can try to copy it to another device first.

Dcheck may report files which have more directory entries than links. Such situations are potentially dangerous; clri discusses a special case of the problem. All the directory entries for the file should be removed. If on the other hand there are more links than directory entries, there is no danger of spreading infection, but merely some disk space that is lost for use. It is sufficient to copy the file (if it has any entries and is useful) then use clri on its inode and remove any directory entries that do exist.

Finally, there may be inodes reported by dcheck that have 0 links and 0 entries. These occur on the root device when the system is stopped with pipes open, and on other file systems when the system stops with files that have been deleted while still open. A clri will free the inode, and an icheck -s will recover any missing blocks.

Why did it crash? UNIX types a message on the console typewriter when it voluntarily crashes. Here is the current list of such messages, with enough information to provide a hope at least of the remedy. The message has the form 'panic: ...', possibly accompanied by other information. Left unstated in all cases is the possibility that hardware or software error produced the message in some unexpected way.

blkdev — The getblk routine was called with a nonexistent major device as argument. Definitely hardware or software error.

devtab — Null device table entry for the major device used as argument to getblk. Definitely hardware or software error.

iinit — An I/O error reading the super-block for the root file system during initialization.

out of inodes — A mounted file system has no more i-nodes when creating a file. Sorry, the device isn't available; the icheck should tell you.

no fs — A device has disappeared from the mounted-device table. Definitely hardware or software error.

no imt — Like 'no fs', but produced elsewhere.

no inodes — The in-core inode table is full. Shouldn't be a panic, just a user error.

no clock — During initialization, neither the line nor programmable clock was found to exist.

swap error — An unrecoverable I/O error during a swap. Really shouldn't be a panic, but it is hard to fix.

unlink - iget — The directory containing a file being deleted can't be found. Hardware or software.

`out of swap space`
: A program needs to be swapped out, and there is no more swap space. It has to be increased. This really shouldn't be a panic, but there is no easy fix.

`out of text`
: A pure procedure program is being executed, and the table for such things is full. This shouldn't be a panic.

`trap`
: An unexpected trap has occurred within the system. This is accompanied by three numbers: containing cpu state; and a 'trap type' which encodes which trap occurred. The trap types are:

1	illegal instruction
3	PRIVOP
6	recursive system call
7	memory parity
9	NMI - power fail or page fault

In some of these cases it is possible for hex 10 to be added into the trap type; this indicates that the processor was in user mode when the trap occurred.

A more complete discussion of system debugging is impossible here.

SEE ALSO

clri(1), icheck(1), dcheck(1), boot(8)

BUGS

This document is generally aimed at an experienced systems

NAME

cron - clock daemon

SYNOPSIS

/etc/cron

DESCRIPTION

cron executes commands at specified dates and times according to the instructions in the file `/usr/lib/crontab`. Since cron never exits, it should only be executed once. This is best done by running cron from the initialization process through the file /etc/rc; see **init(8)**.

Crontab consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (1-7 with 1= monday). Each of these patterns may contain a number in the range above; two numbers separated by a minus meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field is translated to a new-line character. Only the first line (up to a command field is executed by the Shell. The other lines are made available to the command as standard input.

Crontab is examined by cron every minute.

FILES

`/usr/lib/crontab`

NAME

getty — set typewriter mode

SYNOPSIS

/etc/getty [*char*]

DESCRIPTION

getty is invoked by init(8) immediately after a typewriter is opened following a dial-up. It reads the user's login name and calls login(1) with the name as argument. While reading the name getty attempts to adapt the system to the speed and type of terminal being used.

Init calls getty with a single character argument taken from the ttys(5) file entry for the terminal line. This argument determines a sequence of line speeds through which getty cycles, and also the 'login:' greeting message, which can contain character sequences to put various kinds of terminals in useful states.

The user's name is terminated by a new-line or carriage return character. In the second case CRMOD mode is set (see ioctl(2)).

The name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is nonempty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

If the terminal's 'break' key is depressed, getty cycles to the next speed appropriate to the type of line and prints the greeting message again.

Finally, login is called with the user's name as argument.

The following arguments from the ttys file are understood.

0 Cycles through 300-1200-150-110 baud. Useful as a default for dialup lines accessed by a variety of terminals.

\- Intended for an on-line Teletype model 33, for example an operator's console.

1 Optimized for a 150-baud Teletype model 37.

2 Intended for an on-line 9600-baud terminal, for example the Textronix 4104.

1200only 3 Starts at 1200 baud, cycles to 300 and back. Useful with 212 datasets where most terminals run at 1200 speed.

5 Same as '3' but starts at 300.

4	Useful for on-line console DECwriter (LA36).
a	50 baud, standard modes.
b	75 baud, standard modes.
c	110 baud, standard modes.
d	134 baud, standard modes.
e	150 baud, standard modes.
f	200 baud, standard modes.
g	300 baud, standard modes.
h	600 baud, standard modes.
i	1200 baud, standard modes.
j	1800 baud, standard modes.
k	2400 baud, standard modes.
l	4800 baud, standard modes.
m	9600 baud, standard modes.
n	EXTA, standard modes.
o	EXTB, standard modes.
x	EXTB, standard modes (console speed).

Standard modes are any parity, expand tabs, echo, and crlf mode.

SEE ALSO

init(8), login(1), ioctl(2), ttys(5)

NAME

init, rc — process control initialization

SYNOPSIS

```
/etc/init
/etc/rc
```

DESCRIPTION

`init` is invoked as the last step of the boot procedure (see **boot(8)**). Generally its role is to create a process for each typewriter on which a user may log in.

When `init` first is executed the console typewriter `/dev/console`. is opened for reading and writing and the shell is invoked immediately. This feature is used to bring up a single-user system. If the shell terminates, `init` comes up multi-user and the process described below is started.

When `init` comes up multiuser, it invokes a shell, with input taken from the file `/etc/rc`. This command file performs housekeeping like removing temporary files, mounting file systems, and starting daemons.

Then `init` reads the file `/etc/ttys` and forks several times to create a process for each typewriter specified in the file. Each of these processes opens the appropriate typewriter for reading and writing. These channels thus receive file descriptors 0, 1 and 2, the standard input, output and error files. Opening the typewriter will usually involve a delay, since the `open` is not completed until someone is dialed up and carrier established on the channel. Then /etc/getty is called with argument as specified by the last character of the `ttys` file line. `getty` reads the user's name and invokes **login(1)** to log in the user and execute the shell.

Ultimately the shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of `init`, which has been waiting for such an event, wakes up and removes the appropriate entry from the file `utmp`, which records current users, and makes an entry in /usr/adm/wtmp, which maintains a history of logins and logouts. Then the appropriate typewriter is reopened and `getty` is reinvoked.

`Init` catches the hangup signal SIGHUP and interprets it to mean that the system should be brought from multi user to single user. Use "`kill -1 1`" to send the hangup signal.

FILES

/dev/tty?,
/etc/utmp,
/usr/adm/wtmp,
/etc/ttys,
/etc/rc

SEE ALSO

login(1), kill(1), sh(1), ttys(5), getty(8)

NAME

makekey — generate encryption key

SYNOPSIS

`/usr/lib/makekey`

DESCRIPTION

`makekey` improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (i.e. to require a substantial fraction of a second).

The first eight input bytes (the `input key`) can be arbitrary ASCII characters. The last two (the `salt`) are best chosen from the set of digits, upper- and lower-case letters, and '.' and '/'. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the `output key`.

The transformation performed is essentially the following: the salt is used to select one of 4096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but modified in 4096 different ways. Using the input key as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 useful key bits in the result.

`makekey` is intended for programs that perform encryption (e.g. **ed** and **crypt(1)**). Usually its input and output will be pipes.

SEE ALSO

crypt(1), ed(1)