UNIX Programmer's Manual

Part 2
System Calls and Standard Subroutines

N. Plat
editor

Delft University of Technology
Faculty of Mathematics and Informatics
March 1987

Western Electric

Patent Licensing

Guilford Center
P O Box [illegible]
Greensboro NC 27420
919 697 [illegible]

OCT 03 1980

TECHNISCHE HOGESCHOOL DELFT
Department of Mathematics
Juianalaan 132
2628 BL Delft
The Netherlands

Attn: Mr. P. J. van der Hoff

Gentlemen:

Re: May 1, 1979 Software Agreement Between Us
Relating to PWB/UNIX* Time Sharing Operating
System

In response to the August 22, 1980 request from Mr. van der Hoff, your institution may use the licensed software pursuant to the referenced agreement on the following additional CPU's:

LSI-11, Serial No. WM 1720
PDP 11/60, Serial No. AG 00073
Technische Hogeschool Delft
Department of Mathematics - Comp. Rm. 0.101
Julianalaan 132
2628 BL Delft
The Netherlands

Yours truly,

O. L. Wilson

O. L. WILSON
Patent Licensing Manager

*UNIX is a trademark of Bell Laboratories.

TABLE OF CONTENTS

2. System Calls

3. Subroutines

Chapter 2

System calls

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 superuser. It is also returned for attempts by ordinary users to do things allowed only to the superuser.

2 ENOENT No such file or directory
This error occurs when a 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.

35 ETPL Fatal error - tape position lost
A magtape driver has encountered a tape error that has caused it to loose track of tape position.

36 ETOL Tape unit off-line
An attempt has been made to access a tape drive that is not available, i.e., no tape loaded or off-line.

37 ETWL Tape unit write locked
A write operation was attempted on a tape without a write enable ring installed.

38 ETO Tape unit already open
Caused by an attempt to access a tape drive that is already in use or is hung for some reason.

SEE ALSO

intro(3)

RESTRICTIONS

The getfp(), bdflush(), errlog(), and ttlocl() system calls are intentionally not documented. These calls are reserved for use by ULTRIX-11 system programs and must not be called by any user process.

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.

On the Interdata 8/32, the system call arguments correspond well to the arguments of the C routines. The sequence is:

```
la      %2,errno
l       %0,&callno
svc     0,args
```

Thus register 2 points to a word into which the error number will be stored as needed; it is cleared if no error occurs. Register 0 contains the system call number; the nomenclature is identical to that on the PDP11. The argument of the *svc* is the address of the arguments, laid out in storage as in the C calling sequence. The return value is in register 2 (possibly 3 also, as in *pipe*) and is -1 in case of error. The overflow bit in the program status word is also set when errors occur.

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.

DIAGNOSTICS

Access to the file is denied if:

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

[EACCES] Search permission is denied on a component of the path prefix. The owner of a file has permission checked with respect to the 'owner' read, write, and execute mode bits. Members of the file's group other than the owner have permission checked with respect to the 'group' mode bits, and all others have permissions checked with respect to the 'other' mode bits.

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

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

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).

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

DIAGNOSTICS

Acct will fail if one of the following is true:

[EACCES]	*File* is not a regular file.
[EBUSY]	An attempt is made to turn on accounting when it is already on.
[EFAULT]	*File* points outside the process's allocated address space.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The caller is not the superuser.
[EROFS]	The file resides on a read-only file system.

RESTRICTIONS

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

SEE ALSO

acct(5), sa(1)

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*.

RETURN VALUE

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.

DIAGNOSTICS

Sbrk will fail and no additional memory will be allocated if:

[ENOMEM] The maximum possible size of a data segment (compiled into the system) was exceeded.

[ENOMEM] The maximum available memory for a user process was exceeded. For I/D process, text or data+stack was larger than 64KB. For non-I/D process, text+data+stack was larger than 64KB.

RESTRICTIONS

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

SEE ALSO

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

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

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 superuser.

RETURN VALUE

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.

DIAGNOSTICS

Chdir will fail and the current working directory will be unchanged if one or more of the following is true:

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

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

[ENFILE] Insufficient system space to contain i-node.

[ENOENT] The named directory, or an element within the named path, does not exist.

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

Chroot will also fail if:

[EPERM] The user is not the superuser.

SEE ALSO

cd(1)

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 superuser 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 superuser) may change the mode. Only the superuser can set the 1000 mode.

RETURN VALUE

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 superuser.

DIAGNOSTICS

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

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

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

[ENFILE] Insufficient system space to contain i-node.

[ENOENT] The named file, or an element within the named file, does not exist.

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

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

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

SEE ALSO

chmod(1)

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 superuser may execute this call, because if users were able to give files away, they could defeat the (nonexistent) file-space accounting procedures.

RETURN VALUE

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

DIAGNOSTICS

Chown will fail and the file will be unchanged if:

[EACCES]	Search permission is denied on a component of the path prefix.
[EFAULT]	*Name* points outside the process's allocated address space.
[ENFILE]	Insufficient system space to contain i-node.
[ENOENT]	The named file, or an element within path *name*, does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the superuser.
[EROFS]	The named file resides on a read-only file system.

SEE ALSO

chown(1), passwd(5)

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)).

RETURN VALUE

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

DIAGNOSTICS

Close will fail if:

[EBADF] *Fildes* is not an active descriptor.

SEE ALSO

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

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.

RETURN VALUE

Zero is returned if the file is created; –1 is returned otherwise.

DIAGNOSTICS

Creat will fail and the file will not be created if:

[EACCES]	A component of the path prefix denies search permission.
[EACCES]	The directory in which the file is to be created is not writable by the user.
[EFAULT]	Name points outside the process's allocated address space.
[EISDIR]	The named file is a directory.
[EMFILE]	The maximum number of file descriptors allowed are already open.
[ENFILE]	No more system file descriptors are available, or there is insufficient system space to contain the i-node.
[ENOENT]	Element within path name does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENXIO]	Device special file within name is not for the current system (major device number is greater than "nchrdev").
[EROFS]	The directory in which the file is to be created is on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.

SEE ALSO

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

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.

DIAGNOSTICS

Dup and *dup2* will fail if:

[EBADF] *Fildes or fildes2* is not a valid active descriptor.

[EMFILE] Too many descriptors are active.

SEE ALSO

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

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 arg [0], arg [1] ... address null-terminated strings. Conventionally arg [0] 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*

RETURN VALUE

If *exec* returns to the calling process an error has occurred; the return value will be -1 and the global variable *errno* will contain an error code.

DIAGNOSTICS

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

[EACCES]	Search permission is denied for a directory listed in the new process file's path prefix.
[EACCES]	The new process file mode denies execute permission.
[EACCES]	The new process file is not a regular file.
[EFAULT]	*Name* points outside the process's allocated address space.
[EFAULT]	*Name*, *argv*, or *envp* point to an illegal address (outside of the user's image).
[ENFILE]	Insufficient system space to contain i-node.
[ENOENT]	One or more components of the new process file's path name does not exist.
[ENOENT]	The target file is a directory.
[ENOEXEC]	The new process file has the appropriate access permission but the file header contains an invalid magic number.
[ENOEXEC]	The text or data segment is null.
[ENOEXEC]	An overlay is larger than the overlay max size.
[ENOEXEC]	Unable to read file header.
[ENOEXEC]	Found a bad magic number in the file header.
[ENOMEM]	The new process requires more memory than is allowed.
[ENOTDIR]	A component of the new process path *name* is not a directory.
[ETXTBSY]	The new process file is a pure procedure (shared text) file that is currently open for reading or writing by some process.

RESTRICTIONS

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.

SEE ALSO

fork(2), environ(5)

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.

RETURN VALUE

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent process, no child process is created, and the global variable *errno* is set to indicate the error.

Only the superuser can take the last process-table slot.

DIAGNOSTICS

Fork will fail and no child process will be created if:

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

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

[ENOMEM] There is no swap space available for "maxmem".

SEE ALSO

wait(2), exec(2)

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

fperr – get floating point error status

SYNOPSIS

```
fperr(&fpstat)
int fpstat[3];   /* FP status register */
                 /* FP error code */
                 /* FP error address */
```

DESCRIPTION

The *fperr* system call loads the contents of the floating point hardware status and error registers into *fpstat*. This call allows user processes to handle floating point exceptions.

NAME

fpsim – report or change the status of floating point simulation

SYNOPSIS

fpsim(flag)

DESCRIPTION

Fpsim is used to enable, disable, or get the status of the kernel floating point simulator. If *flag* is 2, the current status of the simulator is returned. The return value will be either 0, 1, or 2, which respectively means that the simulator is disabled, enabled, or not configured into the current system.

If *flag* is 0 or 1 the simulator is respectively disabled or enabled, and the old status of the simulator is returned. Only the super user is allowed to change the status of the simulator.

ERRORS

fpsim will fail and the status of the simulator will remain unchanged if:

[EPERM]	The effective user ID of the calling process is not the superuser.
[ENODEV]	The simulator is not configured into the current system.
[EINVAL]	A mode other than 0, 1 or 2 was specified.

SEE ALSO

fpsim(1)

RESTRICTIONS

This call is only temporary, it is intended only for use if the kernel floating point simulator should blow up. Once it has been more thoroughly tested and verified there will no longer be a need to be able to turn it off on a running system.

NAME

ghostname, shostname – get/set name of current host

SYNOPSIS

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

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

DESCRIPTION

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

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

RETURN VALUE

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

ERRORS

The following errors may be returned by these calls:

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

[EPERM] The caller was not the superuser.

RESTRICTIONS

Host names are limited to 32 characters.

These system calls are the same as the *gethostname* and *sethostname* system calls in ULTRIX-32. Due to the current constraint that function names must be significant in the first seven characters, the names were shortend to keep them from clashing with the (not yet implemented) *gethostid* and *sethostid* system calls.

For compatibility reasons, you can use *gethostname* and *sethostname* in your program, provided that either the top of the program contains the line

```
#define gethostname ghostname
```

or the module is compiled with

```
-Dgethostname=ghostname
```

as an argument to *cc*. This works because *cpp* recognizes eight characters of significance.

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

getpid, getppid – get process identification

SYNOPSIS

getpid( )
getppid( )

DESCRIPTION

Getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files. *Getppid* returns the process ID of the parent of the current process.

SEE ALSO

mktemp(3)

ASSEMBLER

(getpid = 20.)
sys getpid
(pid in r0)
(ppid in r1)

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

ioctl, stty, gtty – control device

SYNOPSIS

```
#include <sgtty.h>

ioctl(fildes, request, argp)
struct sgttyb *argp;

stty(fildes, argp)
struct sgttyb *argp;

gtty(fildes, argp)
struct sgttyb *argp;
```

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The writeups of various devices in section 4 discuss how *ioctl* applies to them.

For certain status setting and status inquiries about terminal devices, the functions *stty* and *gtty* are equivalent to

```
ioctl(fildes, TIOCSETP, argp)
ioctl(fildes, TIOCGETP, argp)
```

respectively; see *tty*(4).

The following two calls, however, apply to any open file:

```
ioctl(fildes, FIOCLEX, NULL);
ioctl(fildes, FIONCLEX, NULL);
```

The first causes the file to be closed automatically during a successful *exec* operation; the second reverses the effect of the first.

RETURN VALUE

Zero is returned if the call was successful; −1 if the file descriptor does not refer to the kind of file for which it was intended.

DIAGNOSTICS

Ioctl will fail if one or more of the following is true:

[EBADF]	*Fildes* is not a valid descriptor.
[EBADF]	Carrier was lost on the terminal line.
[EFAULT]	*Argp* points to an illegal address.
[ENOTTY]	The specified request does not apply to the kind of object that the descriptor *fildes* references.
[ENXIO]	An attempt was made to set an invalid line discipline.
[EPERM]	The specified ioctl() function is only valid for the superuser (TIOCLOCAL, TIOCSMLB, TIOCCMLB).

RESTRICTIONS

Strictly speaking, since *ioctl* may be extended in different ways to devices with different properties, *argp* should have an open-ended declaration like

```
union { struct sgttyb ...; ... } *argp;
```

The important thing is that the size is fixed by 'struct sgttyb'.

SEE ALSO

stty(1), tty(4), exec(2)

ASSEMBLER

(ioctl = 54.)
sys ioctl; fildes; request; argp

(stty = 31.)
(file descriptor in r0)
stty; argp

(gtty = 32.)
(file descriptor in r0)
sys gtty; argp

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 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 superuser.

If the process number is 0, the signal is sent to all other processes in the sender's process group; see *tty*(4).

If the process number is −1, and the user is the superuser, 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.

RETURN VALUE

Zero is returned if the process is killed; −1 is returned otherwise, with *errno* containing the error code.

DIAGNOSTICS

Kill will fail and no signal will be sent if any of the following occur:

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] Attempt was made to send to an unassigned process group.

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

[ESRCH] No processes killed.

SEE ALSO

signal(2), kill(1)

ASSEMBLER

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

NAME

killpg – send signal to a process or a process group

SYNOPSIS

killpg(pgrp, sig)

cc ... –ljobs

DESCRIPTION

Killpg sends the signal *sig* to the specified process group. See *sigsys*(2) for a list of signals; see *intro*(3J) for an explanation of process groups.

The sending process and members of the process group must have the same effective user ID, otherwise this call is restricted to the superuser. As a single special case the continue signal SIGCONT may be sent to any process which is a descendant of the current process. This allows a command interpreter such as *csh*(1) to restart set-user-id processes stopped from the keyboard with a stop signal.

The calls

killpg(0, sig)

and

kill(0, sig)

have identical effects, sending the signal to all members of the invoker's process group (including the process itself). It is preferable to use the call involving *kill* in this case, as it is portable to other UNIX systems.

RETURN VALUE

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

DIAGNOSTICS

Killpg will fail and no signal will be sent if any of the following occur:

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] Attempt to send to an unassigned process group.

[EPERM] The caller is not the superuser.

[ESRCH] No process can be found corresponding to the process group specified by *pgrp*.

[ESRCH] No processes killed.

RESTRICTIONS

The job control facilities are not available in standard Version 7 UNIX. These facilities are still under development and may change in future releases of the system as better interprocess communication facilities and support for virtual terminals become available. The options and specifications of this system call and even the call itself are thus subject to change.

SEE ALSO

csh(1), kill(1), kill(2), signal(2), sigsys(2J), intro(3J)

ASSEMBLER (PDP-11)

```
(kill = 37.)
(process number in r0)
 sys kill;        – sig    (negative signal number means killpg)
```

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.

DIAGNOSTICS

Link will fail and no link will be created if:

[EACCES]	A component of either path prefix denies search permission.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EEXIST]	A file by the same name already exists.
[EFAULT]	One of the path names specified is outside the process's allocated address space.
[ENFILE]	Insufficient system space to contain i-node.
[ENOTDIR]	A path prefix component is not a directory.
[ENOENT]	A path prefix component does not exist.
[ENOENT]	The file named by *name1* does not exist.
[EPERM]	The file named by *name1* is a directory and the effective user ID is not the superuser.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EXDEV]	The link named by *name2* and the file named by *name1* are on different file systems.
[E2BIG]	The argument count exceeded "ncargs".

SEE ALSO

ln(1), unlink(2)

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 un*lock*ed. This call may only be executed by the superuser.

DIAGNOSTICS

Lock will fail if:

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

RESTRICTIONS

*Lock*ed 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.

RETURN VALUE

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

DIAGNOSTICS

Lseek will fail and the file pointer will remain unchanged if:

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

[ESPIPE] *Fildes* is associated with a pipe.

RESTRICTIONS

Lseek is a no-op on character special files.

SEE ALSO

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

ASSEMBLER

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

Offset1 and *offset2* are the high and low words of *offset*; r0 and 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 superuser.

RETURN VALUE

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

DIAGNOSTICS

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

[EACCES]	Access to a directory file denied.
[EACCES]	A component of the path prefix denies search permission.
[EEXIST]	The named file exists.
[EFAULT]	*Name* points outside the process's allocated address space.
[ENOENT]	A component of the path prefix does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOSPC]	No i-nodes are available on the device.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the superuser.
[EROFS]	The named file resides on a read-only file system.

SEE ALSO

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

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.

DIAGNOSTICS

Mount will fail if:

[EACCES]	A component of the path prefix denies search permission.
[EBUSY]	*Name* is not a directory or another process currently holds a reference to it.
[EBUSY]	No space remains in the mount table.
[ENODEV]	*Special* does not exist.
[ENOENT]	An element within *name* does not exist.
[ENOTBLK]	*Special* is not for a block oriented device.
[ENOTDIR]	A component of the path prefix in *special* or *name* is not a directory.
[ENXIO]	The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).
[EPERM]	The process's effective user ID is not the superuser.
[EROFS]	*Name* resides on a read-only file system.

Umount may fail with one of the following errors:

[EBUSY]	A process is holding a reference to a file located on the file system.
[EINVAL]	The requested device is not in the mount table.
[ENODEV]	*Special* does not exist.
[ENOTBLK]	*Special* is not a block device.
[ENXIO]	The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).
[EPERM]	The process's effective user ID is not the superuser.

SEE ALSO

mount(1)

ASSEMBLER

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

(umount = 22.)
sys umount; special

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 superuser. 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).

DIAGNOSTICS

Nice will fail and the process's scheduling priority remain the same if:

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

SEE ALSO

nice(1)

ASSEMBLER

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

NAME

nostk – allow process to manage its own stack

SYNOPSIS

nostk()

DESCRIPTION

Nostk informs the system that the process wishes to manage its own stack. The system releases the stack segment(s) it has reserved, making them available for allocation (via *brk(2))* by the user.

C program should use *nostk* only with great caution and understanding of the C language calling and stack conventions. It is most useful for assembler programs that want to use the entier available address space.

ASSEMBLER

(nostk = 84.)
sys nostk

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.

DIAGNOSTICS

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

[EACCES]	A component of the path prefix denies search permission.
[EBUSY]	The device specified is already open for exclusive use and the caller is not the superuser.
[EFAULT]	*Name* points outside the process's allocated address space.
[EISDIR]	The named file is a directory and the arguments specify it is to be opened for writing.
[EMFILE]	The maximum number of file descriptors allowed are already open.
[ENFILE]	No more system file descriptors are available.
[ENFILE]	Insufficient system space to contain i-node.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENXIO]	Device special file is not for the current system (major device number is greater than "nchrdev", or greater than "nblkdev").
[ENXIO]	An attempt was made to open /dev/tty without a controlling terminal.
[ENXIO]	An illegal device was specified, or the device is already open.
[EROFS]	The named file resides on a read-only file system and the file is to be modified.
[ETO]	The specified tape drive is already open.
[ETOL]	The specified tape drive is not on-line.
[ETPL]	Tape position was lost for the specified tape drive.
[ETWL]	The specified tape drive is physically write-locked.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and the *open* call has requested write access.

SEE ALSO

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

ASSEMBLER

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

phys – allow a process to access physical addresses

SYNOPSIS

phys(segreg, size, physadr)

DESCRIPTION

The argument *segreg* specifies a process virtual (data-space) address range of 8K bytes starting at virtual address *segreg*× 8K bytes. This address range is mapped into physical address *physadr*× 64 bytes. Only the first *size*× 64 bytes of this mapping is addressable. If *size* is zero, any previous mapping of this virtual address range is nullified. For example, the call

```
phys(6, 1, 0177775);
```

will map virtual addresses 0160000-0160077 into physical addresses 017777500-017777577. In particular, virtual address 0160060 is the PDP-11 console located at physical address 017777560.

This call may only be executed by the superuser.

DIAGNOSTICS

Phys will fail and the segmentation registers be unaffected if:

[EINVAL] *Segreg* is less than 0 or greater than 7.

[EINVAL] *Size* is less than 0 or greater than 128.

[EINVAL] An attempt was made to map a segment which was already mapped by some method other than by *phys*.

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

RESTRICTIONS

This system call is obviously very machine dependent and very dangerous. This system call is not considered a permanent part of the system.

SEE ALSO

PDP-11 segmentation hardware

ASSEMBLER

(phys = 52.)

sys phys; segreg; size; physadr

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. A signal is generated if a write on a pipe with only one end is attempted.

DIAGNOSTICS

The *pipe* call will fail if:

[EFAULT] The *fildes* buffer is in an invalid area of the process's allocated address space.

[EMFILE] Too many file descriptors are active.

[ENFILE] No more system file descriptors are available.

[ENFILE] Insufficient system space to contain i-node.

[ENOSPC] No more i-nodes are available on the device.

RESTRICTIONS

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

SEE ALSO

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

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: 0177777(8) gives a 1-1 mapping of pc's to words in *buff*; 077777(8) maps each pair of instruction words together. 02(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling 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. (On the PDP-11 the T-bit is used and just one instruction is executed; on the Interdata the stop does not take place until a store instruction is executed.) This is part of the mechanism for implementing breakpoints.

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.

On the Interdata 8/32, 'word' means a 32-bit word and 'even' means 0 mod 4.

DIAGNOSTICS

[EFAULT] The specified address is out of bounds.

[EPERM] The specified process cannot be traced.

[ESRCH] The specified process does not exist.

RESTRICTIONS

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.

SEE ALSO

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

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;
unsigned nbytes;

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.

DIAGNOSTICS

Read will fail if one or more of the following is true:

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

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

[EFAULT] The specified address is either odd or out of range when dealing with character special files.

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

[ENXIO] The /dev/mem address specified is illegal.

[ETPL] The magtape tape position was lost or the tape unit went offline during a read.

[EWOULDBLOCK] The process would hang waiting for input (when mode is set to interrupt on each character input).

SEE ALSO

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

ASSEMBLER

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

NAME

renice – set program priority

SYNOPSIS

renice(pid,nice)

DESCRIPTION

Renice sets the scheduling priority of the process with id *pid* to *nice*. The superuser may *renice* any process to any value from -127 to 127. Other users may only increase the nice value of one of their own processes.

DIAGNOSTICS

Renice will fail and the process's scheduling priority will be unaltered if:

[EPERM] The named process's effective user ID is not the same as the calling process and the process's effective user ID is not the superuser.

[ESRCH] The named process does not exist.

SEE ALSO

nice(1), renice(1), nice(2)

ASSEMBLER

(renice = 87.)
sys renice; pid; nice
(old nice value in r0)

NAME

setpgrp, getpgrp – set/get process group

SYNOPSIS

int getpgrp(pid)

setpgrp(pid, pgrp)

cc ... –ljobs

DESCRIPTION

The process group of the specified process is returned by *getpgrp*. *Setpgrp* sets the process group of the specified process *pid* to the specified *pgrp*. If *pid* is zero, then the call applies to the current process.

If the invoker is not the superuser, then the affected process must have the same effective user-id as the invoker or be a descendant of the invoking process.

This call is used by *csh*(1) to create process groups in implementing job control. The TIOCGPGRP and TIOCSPGRP calls described in *tty*(4) are used to get/set the process group of the control terminal.

See *intro*(3J) for a general discussion of job control.

DIAGNOSTICS

Setpgrp will fail and the process group will not be altered if one of the following is true:

[EPERM] The effective user ID of the requested process is different from that of the caller and the process is not a descendent of the calling process.

[ESRCH] The requested process does not exist.

RESTRICTIONS

The job control facilities are not available in standard version 7 UNIX. These facilities are still under development and may change in future releases of the system as better interprocess communication facilities and support for virtual terminals become available. The options and specifications of these system calls and even the calls themselves are thus subject to change.

A system call *setpgrp* has been implemented in other versions of UNIX which are not widely used outside of Bell Laboratories; these implementations have, in general, slightly different semantics.

SEE ALSO

csh(1), getuid(2), intro(3J), tty(4)

ASSEMBLER

(setpgrp = 39.)
(process id in r0)
sys setpgrp; newgrp

(getpgrp is implemented as setpgrp(pid,-1))
sys setgrp; -1
(process group 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 superuser or if the argument is the real ID.

DIAGNOSTICS

Setuid will fail and the user ID will be unchanged if:

[EPERM] A non-superuser attempt is made to change the effective user ID to anything other than the real user ID.

Setgid will fail and the group ID will be unchanged if:

[EPERM] A non-superuser attempt is made to change the effective group ID to anything other than the real group ID.

SEE ALSO

getuid(2)

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.

DIAGNOSTICS

Signal will fail if:

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

RESTRICTIONS

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.

SEE ALSO

kill(1), kill(2), ptrace(2), sigsys(2J), setjmp(3), sigset(3J)

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

sigsys – catch or ignore signals

SYNOPSIS

```
#include <signal.h>

(*sigsys(sig, func))()
void (*func)();
int (*func)();

cc ... -ljobs
```

DESCRIPTION

N.B.: The system currently supports two signal implementations. The one described in *signal*(2) is standard in version 7 UNIX systems, and retained for backward compatibility as it is different in a number of ways. The one described here (with the interface in *sigset*(3J)) provides for the needs of the job control mechanisms (see *intro*(3J)) used by *csh*(1), and corrects the bugs in the standard implementation of signals, allowing programs which process interrupts to be written reliably.

The routine *sigsys* is not normally called directly; rather the routines of *sigset*(3J) should be used. These routines are kept in the 'jobs' library, accessible by giving the loader option **-ljobs**. The features described here are less portable then those of *signal*(2) and should not be used in programs which are to be moved to other versions of UNIX.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see *tty*(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals which cannot be blocked, the *sigsys* call allows signals either to be ignored, held until a later time (protecting critical sections in the process), or to cause an interrupt to a specified location. Here is the list of all 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, held or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
	16	unassigned
SIGSTOP	17	stop (cannot be caught, held or ignored)
SIGTSTP	18	stop signal generated from keyboard
SIGCONT	19	continue after stop
SIGCHLD	20	child status has changed

SIGTTIN	21	background read attempted from control terminal
SIGTTOU	22	background write attempted to control terminal
SIGTINT	23	input record is available at control terminal
SIGXCPU	24	cpu time limit exceeded (see *vlimit*(2)) (VAX-11 only)
SIGXFSZ	25	file size limit exceeded (see *vlimit*(2)) (VAX-11 only)

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

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with or . Signals marked with are discarded if the action is SIG_DFL; signals marked with cause the process to stop. If *func* is SIG_HOLD the signal is remembered if it occurs, but not presented to the process; it may be presented later if the process changes the action for the signal. If *func* is SIG_IGN the signal is subsequently ignored, and pending instances of the signal are discarded (i.e. if the action was previously SIG_HOLD.) Otherwise when the signal occurs *func* will be called.

A return from the function will continue the process at the point it was interrupted. Except as indicated, a signal, set with *sigsys*, is reset to SIG_DFL after being caught. However by specifying DEFERSIG(func) as the last argument to *sigsys*, one causes the action to be set to SIG_HOLD before the interrupt is taken, so that recursive instances of the signal cannot occur during handling of the signal.

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 terminal; but not a file) and during a *pause* or *wait*(2). When a signal occurs during one of these calls, 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. *Read* and *write* calls which have done no I/O, *ioctl*s blocked with SIGTTOU, and *wait2* or *wait3* calls are restarted.

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

The system provides two other functions by oring bits into the signal number: SIGDOPAUSE causes the process to *pause* after changing the signal action. It can be used to atomically re-enable a held signal which was being processed and wait for another instance of the signal. SIGDORTI causes the system to simulate an *rei* instruction clearing the mark the system placed on the stack at the point of interrupt before checking for further signals to be presented due to the specified change in signal actions. This allows a signal package such as *sigset*(3J) to dismiss from interrupts cleanly removing the old state from the stack before another instance of the interrupt is presented.

After a *fork*(2) or *vfork*(2) the child inherits all signals. *Exec*(2) resets all caught signals to default action; held signals remain held and ignored signals remain ignored.

RETURN VALUE

The value BADSIG is returned if the given signal is out of range.

DIAGNOSTICS

Sigsys will fail if:

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

RESTRICTIONS

The job control facilities are not available in standard version 7 UNIX. These facilities are still under development and may change in future releases of the system as better interprocess communication facilities and support for virtual terminals become available. The

options and specifications of this facility and the system calls supporting it are thus subject to change.

Since only one signal action can be changed at a time, it is not possible to get the effect of SIGDOPAUSE for more than one signal at a time.

The traps (listed below) should be distinguishable by extra arguments to the signal handler, and all hardware supplied parameters should be made available to the signal routine.

SEE ALSO

kill(1), kill(2), ptrace(2), intro(3J), sigset(3J), setjmp(3), tty(4)

ASSEMBLER (PDP-11)

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

If *label* is 0, default action is reinstated. If *label* is 1, the signal is ignored. If *label* is 3, the signal is held. Any other even *label* specifies an address in the process where an interrupt is simulated. If label is otherwise odd, the signal is sent to the function whose address is the label with the low bit cleared with the action set to SIG_HOLD. (Thus DEFERSIG is indicated by the low bit of a signal catch address. 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.

```
#ifndef STAT_H
#define STAT_H

struct  stat
{
        dev_t   st_dev;
        ino_t   st_ino;
        unsigned short st_mode;
        short   st_nlink;
        id_t    st_uid;
        id_t    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 */
```

#endif STAT_H

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

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 by writing and changing the i-node.

DIAGNOSTICS

Stat will fail if one or more of the following is true:

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

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

[ENFILE] Insufficient system space to contain i-node.

[ENOENT] The named file, or an element within the named file, does not exist.

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

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

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

[EFAULT] *Buf* points to an address outside of the process's allocated address space.

SEE ALSO

ls(1), filsys(5)

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 superuser may use this call.

DIAGNOSTICS

Stime will fail and the system's current time will be unchanged if:

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

SEE ALSO

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

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)

RESTRICTIONS

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(tloc)
long *tloc;

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

```
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.

DIAGNOSTICS

Ftime will fail if:

[EFAULT] *Tp* points to an address outside the process's allocated address space.

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

times – get process times

SYNOPSIS

times(buffer)
struct tbuffer *buffer;

DESCRIPTION

Times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/HZ seconds, where HZ = 60 in North America.

After the call, the buffer will appear as follows:

```
struct tbuffer {
        long    proc_user_time;
        long    proc_system_time;
        long    child_user_time;
        long    child_system_time;
};
```

The children times are the sum of the children's process times and their children's times.

SEE ALSO

time(1), time(2)

ASSEMBLER

(times = 43.)
sys times; buffer

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.

DIAGNOSTICS

The *unlink* will succeed unless:

[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[EFAULT]	*Name* points to an address outside the process's allocated address space.
[ENFILE]	Insufficient system space to contain i-node.
[ENOENT]	The named file, or an element within the named file, does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The named file is a directory and the process's effective user ID is not the superuser.
[EROFS]	The named file resides on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is currently open for reading or writing by some process.

SEE ALSO

rm(1), link(2)

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 superuser. The 'inode-changed' time of the file is set to the current time.

DIAGNOSTICS

Utime will fail if one or more of the following is true:

[EACCES]	A component of the path prefix denies search permission.
[EFAULT]	*Timep* points to an address outside the process's allocated address space.
[ENFILE]	Insufficient system space to contain i-node.
[ENOENT]	The named file, or an element within the named file, does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The process's effective user ID is not the superuser and not the same as the owner of the file.
[EROFS]	The named file is located on a read-only file system.

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.

RETURN VALUE

If *wait* returns due to the receipt of a signal, a value of −1 is returned to the calling process and the *errno* is set to EINTR. If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

DIAGNOSTICS

Wait will fail and return immediately if the following is true:

[ECHILD] The calling process has no unwaited-for child processes.

[EFAULT] The *status* argument points to an address outside of the process's allocated address space.

SEE ALSO

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

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

wait2 – wait for process to terminate

SYNOPSIS

```
#include <wait.h>

wait2(&w.w_status, options)
union wait w;
int options;

cc ... -ljobs
```

DESCRIPTION

Wait2 is similar to the standard *wait*(2) system call, but allow additional options useful with job control. They return the process ID of a terminated or stopped child process. The *w.w_status* and *option* words are described by definitions and macros in the file <wait.h>; the union and its bitfield definitions and associated macros given there provide convenient and mnemonic access to the word of status returned by a *wait2* call. If the call returns a process ID, several macros are available to interpret the status word returned. If the process is stopped, WIFSTOPPED(w) is true, and the signal that caused it to stop is w.w_stopsig. If the process is not stopped (has terminated), WIFEXITED(w) determines whether it terminated by calling *exit*(2); if so, the exit code is w.w_retcode. WIFSIGNALED(w) is true if the process was terminated by a signal (see *signal*(2)); the signal causing termination was w.w_termsig, and w.w_coredump indicates whether a core dump was produced.

There are two *options*, which may be combined by *or*ing them together. The first is WNOHANG which causes the *wait2* to not hang if there are no processes which wish to report status, rather returning a pid of 0 in this case as the result of the *wait2*. The second option is WUNTRACED which causes *wait2* to return information when children of the current process which are stopped but not traced (with *ptrace*(2)) because they received a SIGTTIN, SIGTTOU, SIGTSTP or SIGSTOP signal. See *sigsys*(2J) for a description of these signals.

RETURN VALUE

Returns –1 if there are no children not previously waited for, or 0 if the WNOHANG option is given and there are no stopped or exited children.

DIAGNOSTICS

Wait2 will fail and return immediately if one or more of the following is true:

[ECHILD] The calling process has no unwaited-for child processes.

[EFAULT] The w.w_status argument points to an address outside the process's allocated address space.

RESTRICTIONS

This call is peculiar to this version of UNIX. The options and specifications of this system call and even the call itself are subject to change. It may be replaced by other facilities in future versions of the system.

SEE ALSO

exit(2), fork(2), sigsys(2J), wait(2)

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.

DIAGNOSTICS

Write will fail if one or more of the following is true:

[EBADF]	Fildes is not a valid descriptor open for writing.
[EFAULT]	The specified address is odd or the count is odd when dealing with disk character special files.
[EFAULT]	The buffer points to an address outside the process's allocated address space.
[EFBIG]	An attempt was made to write a file that exceeds the maximum file size.
[ENXIO]	The specified /dev/mem address is illegal.
[EPIPE]	An attempt is made to write to a pipe that is not open for reading by any process.
[EQUOT]	An attempt was made to write a file that exceeds a quota governing the file.
[ETPL]	The magtape tape position was lost or the tape went offline during a write.

SEE ALSO

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

ASSEMBLER

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

NAME

zaptty – zap the controlling tty

SYNOPSIS

zaptty( )

DESCRIPTION

Zaptty will disassociates a process from it's controlling tty. The next terminal type device that the process opens will become the new controlling tty. *Zaptty* is useful for daemons, since they usually do not want to be associated with any terminal.

For obvious reasons, this call is restricted to the superuser.

ASSEMBLER

(zaptty = 69.)
sys zaptty

Chapter 3

Subroutines

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 librarys *libc* and *libcov*, 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' or '–lcov' 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 librarys *libc* and *libcov* 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/libcov.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

SYNOPSIS

abort()

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*

RESTRICTIONS

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)

RESTRICTIONS

There are no provisions for overflow.

NAME

crypt, encrypt – a one way hashing encryption algorithm

SYNOPSIS

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

encrypt(block)
char *block;

DESCRIPTION

Crypt is the password encryption routine. It is based on a one way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search

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

There is a character array of length 64 containing only the characters with numerical value 0 and 1. When 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 by **crypt.**

The *encrypt* entry provides (rather primitive) access to the actual hashing algorithm. The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value of 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *crypt.*

SEE ALSO

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

RESTRICTIONS

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 {
        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)

RESTRICTIONS

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

curses – screen functions with "optimal" cursor motion

SYNOPSIS

cc [flags] files –lcurses –ltermlib [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh()* tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin()* should be called before exiting.

SEE ALSO

Screen Updating and Cursor Movement Optimization: A Library Package, Ken Arnold, stty(2), setenv(3), termcap(5)

AUTHOR

Ken Arnold

FUNCTIONS

addch(ch)	add a character to stdscr
addstr(str)	add a string to stdscr
box(win,vert,hor)	draw a box around a window
crmode()	set cbreak mode
clear()	clear stdscr
clearok(scr,boolf)	set clear flag for scr
clrtobot()	clear to bottom on stdscr
clrtoeol()	clear to end of line on stdscr
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete win
echo()	set echo mode
endwin()	end window modes
erase()	erase stdscr
getch()	get a char through stdscr
getcap(name)	get terminal capability name
getstr(str)	get a string through stdscr
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) co-ordinates
initscr()	initialize screens
insch(c)	insert a char
insertln()	insert a line
leaveok(win,boolf)	set leave flag for win
longname(termbuf,name)	get long name from termbuf
move(y,x)	move to (y,x) on stdscr
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocrmode()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping
noraw()	unset raw mode

overlay(win1,win2)	overlay win1 on win2
overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on stdscr
raw()	set raw mode
refresh()	make current screen look like stdscr
resetty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through stdscr
scroll(win)	scroll win one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
standend()	end standout mode
standout()	start standout mode
subwin(win,lines,cols,begin_y,begin_x)	create a subwindow
touchwin(win)	"change" all of win
unctrl(ch)	printable version of ch
waddch(win,ch)	add char to win
waddstr(win,str)	add string to win
wclear(win)	clear win
wclrtobot(win)	clear to bottom of win
wclrtoeol(win)	clear to end of line on win
wdelch(win,c)	delete char from win
wdeleteln(win)	delete line from win
werase(win)	erase win
wgetch(win)	get a char through win
wgetstr(win,str)	get a string through win
winch(win)	get char at current (y,x) in win
winsch(win,c)	insert char into win
winsertln(win)	insert line into win
wmove(win,y,x)	set current (y,x) co-ordinates on win
wprintw(win,fmt,arg1,arg2,...)	printf on win
wrefresh(win)	make screen look like win
wscanw(win,fmt,arg1,arg2,...)	scanf through win
wstandend(win)	end standout mode on win
wstandout(win)	start standout mode on win

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**.

*Key*s and *content*s 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*.

RESTRICTIONS

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

opend, fopend, closed, fclosd – generic device open/close routines

SYNOPSIS

```
#include <stdio.h>

opend(name, mode)
char *name;
int mode;

closed(fildes)
int mode;

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

fclosd(stream)
FILE *stream;
```

DESCRIPTION

These functions extract and use characteristics from the devices file *devices*(5).

opend acts like a normal open but extracts the entry for the actual *device name* from the data file and does settings as specified in the data file.

fopend acts like a normal fopen but returns a *file pointer* in stead of a *file descriptor* (see also opend).

closed closes the device opened by opend.

fclosd closes the device opened by fopend.

FILES

/etc/devices

SEE ALSO

devices(5), open(2), close(2), fopen(3S), fclose(3S)

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 *ec* , except that the correct digit has been rounded for Fortran F-format output of the number f digits specified by *ndigits.*

Gcvt converts the *vali* 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)

RESTRICTIONS

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 exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

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

double sqrt(x)
double x;
```

DESCRIPTION

Exp returns the exponential function of *x*.

Log returns the natural logarithm of *x*; *log10* returns the base 10 logarithm.

Pow returns x^y.

Sqrt returns the square root of *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.

RESTRICTIONS

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* = *x*2***n* indirectly through *eptr*.

Ldexp returns the quantity *value**2***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.*

RESTRICTIONS

The end-of-file return from *getchar* is incompatible with that in UNIX editions 1-6.

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

getdate – convert time and date from ASCII

SYNOPSIS

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

time_t getdate(buf, now)
char *buf;
struct timeb *now;
```

DESCRIPTION

Getdate is a routine that converts most common time specifications to standard UNIX format. The first argument is the character string containing the time and date; the second is the assumed current time (used for relative specifications); if **NULL** is passed, *ftime*(3C) is used to obtain the current time and timezone.

The character string consists of 0 or more specifications of the following form:

tod — A *tod* is a time of day, which is of the form *hh*[:*mm*[:*ss*]] (or *hhmm*) [*meridian*] [*zone*]. If no meridian – **am** or **pm** – is specified, a 24-hour clock is used. A *tod* may be specified as just *hh* followed by a *meridian*.

date — A *date* is a specific month and day, and possibly a year. Acceptable formats are *mm*/*dd*[/*yy*] and *monthname dd*[, *yy*] If omitted, the year defaults to the current year; if a year is specified as a number less than 100, 1900 is added. If a number not followed by a day or relative time unit occurs, it will be interpreted as a year if a *tod*, *monthname*, and *dd* have already been specified; otherwise, it will be treated as a *tod*. This rule allows the output from *date*(1) or *ctime*(3) to be passed as input to *getdate*.

day — A *day* of the week may be specified; the current day will be used if appropriate. A *day* may be preceded by a *number*, indicating which instance of that day is desired; the default is **1**. Negative *numbers* indicate times past. Some symbolic *numbers* are accepted: **last**, **next**, and the ordinals **first** through **twelfth** (**second** is ambiguous, and is not accepted as an ordinal number). The symbolic number **next** is equivalent to **2**; thus, **next monday** refers not to the immediately coming Monday, but to the one a week later.

relative time — Specifications relative to the current time are also accepted. The format is [*number*] *unit*; acceptable units are **year**, **month**, **fortnight**, **week**, **day**, **hour**, **minute**, and **second**.

The actual date is formed as follows: first, any absolute date and/or time is processed and converted. Using that time as the base, day-of-week specifications are added; last, relative specifications are used. If a date or day is specified, and no absolute or relative time is given, midnight is used. Finally, a correction is applied so that the correct hour of the day is produced after allowing for daylight savings time differences.

Getdate accepts most common abbreviations for days, months, etc.; in particular, it will recognize them with upper or lower case first letter, and will recognize three-letter abbreviations for any of them, with or without a trailing period. Units, such as **weeks**, may be specified in the singular or plural. Timezone and meridian values may be in upper or lower case, and with or without periods.

SEE ALSO

ctime(3), time(2)

AUTHOR

Steven M. Bellovin (unc!smb)
Dept. of Computer Science
University of North Carolina at Chapel Hill

BUGS

Because *yacc*(1) is used to parse the date, *getdate* cannot be used a subroutine to any program that also needs *yacc*.

The grammar and scanner are rather primitive; certain desirable and unambiguous constructions are not accepted. Worse yet, the meaning of some legal phrases is not what is expected; **next week** is identical to **2 weeks**.

The daylight savings time correction is not perfect, and can get confused if handed times between midnight and 2:00 am on the days that the reckoning changes.

Because *localtime*(2) accepts an old-style time format without zone information, attempting to pass *getdate* a current time containing a different zone will probably fail.

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 broken-out fields of a line in the group file.

```
struct  group {
        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.

RESTRICTIONS

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.

RESTRICTIONS

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)

RESTRICTIONS

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 {
        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 returned on EOF or error.

RESTRICTIONS

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.

RESTRICTIONS

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*x + y*y),
```

taking precautions against unwarranted overflows.

SEE ALSO

exp(3) for *sqrt*

NAME

intro – summary of job control facilities

SYNOPSIS

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

int fildes, signo;
short pid, pgrp;
union wait status;

ioctl(fildes, TIOCSPGRP, &pgrp)
ioctl(fildes, TIOCGPGRP, &pgrp)

sigset(signo, action)
sighold(signo)
sigrelse(signo)
sigpause(signo)
sigsys(signo, action)

wait2(&status, options)

cc ... -ljobs
```

DESCRIPTION

The facilities described here are used to support the job control implemented in *csh*(1), and may be used in other programs to provide similar facilities. Because these facilities are not standard in UNIX and because the signal mechanisms are also slightly different, the associated routines are not in the standard C library, but rather in the **-ljobs** library.

For descriptions of the individual routines see the various sections listed in SEE ALSO below. This section attempt only to place these facilities in context, not to explain the semantics of the individual calls.

Terminal arbitration mechanisms.

The job control mechanism works by associating with each process a number called a *process group ;* related processes (e.g. in a pipeline) are given the same process group. The system assigns a single process group number to each terminal. Processes running on a terminal are given read access to that terminal only if they are in the same process group as that terminal.

Thus a command interpreter may start several jobs running in different process groups and arbitrate access to the terminal by controlling which, if any, of these processes is in the same process group as the terminal. When a process which is not in the process group of the terminal tries to read from the terminal, all members of the process group of the process receive a SIGTTIN signal, which normally then causes them to stop until they are continued with a SIGCONT signal. (See *sigsys*(2J) for a description of these signals; *tty*(4) for a description of process groups.)

If a process which is not in the process group of the terminal attempts to change the terminals mode, the process group of that process is sent a SIGTTOU signal, causing the process group to stop. A similar mechanism is (optionally) available for output, causing processes to block with SIGTTOU when they attempt to write to the terminal while not in its process group; this is controlled by the LTOSTOP bit in the tty mode word, enabled by "stty tostop" and disabled (the default) by "stty -tostop." (The LTOSTOP bit is described in *tty*(4)).

How the shell manipulates process groups.

A shell which is interactive first establishes its own process group and a process group for the terminal; this prevents other processes from being inadvertently stopped while the terminal is under its control. The shell then assigns each job it creates a distinct process group. When a job is to be run in the foreground, the shell gives the terminal to the process group of the job using the TIOCSPGRP ioctl (See *ioctl*(2) and *tty*(4)). When a job stops or completes, the shell reclaims the terminal by resetting the terminals process group to that of the shell using TIOCSPGRP again.

Shells which are running shell scripts or running non-interactively do not manipulate process groups of jobs they create. Instead, they leave the process group of sub-processes and the terminal unchanged. This assures that if any sub-process they create blocks for terminal i/o, the shell and all its sub-processes will be blocked (since they are a single process group). The first interactive parent of the non-interactive shell can then be used to deal with the stoppage.

Processes which are orphans (whose parents have exited), and descendants of these processes are protected by the system from stopping, since there can be no interactive parent. Rather than blocking, reads from the control terminal return end-of-file and writes to the control terminal are permitted (i.e. LTOSTOP has no effect for these processes.) Similarly processes which ignore or hold the SIGTTIN or SIGTTOU signal are not sent these signals when accessing their control terminal; if they are not in the process group of the control terminal reads simply return end-of-file. Output and mode setting are also allowed.

Before a shell *suspends* itself, it places itself back in the process group in which it was created, and then sends this original group a stopping signal, stopping the shell and any other intermediate processes back to an interactive parent. The shell also restores the process group of the terminal when it finishes, as the process which then resumes would not necessarily be in control of the terminal otherwise.

Naive processes.

A process which does not alter the state of the terminal, and which does no job control can invoke subprocesses normally without worry. If such a process issues a *system*(3S) call and this command is then stopped, both of the processes will stop together. Thus simple processes need not worry about job control, even if they have "shell escapes" or invoke other processes.

Processes which modify the terminal state.

When first setting the terminal into an unusual mode, the process should check, with the stopping signals held, that it is in the foreground. It should then change the state of the terminal, and set the catches for SIGTTIN, SIGTTOU and SIGTSTP. The following is a sample of the code that will be needed, assuming that unit 2 is known to be a terminal.

```
        short   tpgrp;
        ...

retry:
        sigset(SIGTSTP, SIG_HOLD);
        sigset(SIGTTIN, SIG_HOLD);
        sigset(SIGTTOU, SIG_HOLD);
        if (ioctl(2, TIOCGPGRP, &tpgrp) != 0)
                goto nottty;
        if (tpgrp != getpgrp(0)) { /* not in foreground */
                sigset(SIGTTOU, SIG_DFL);
```

```
                kill(0, SIGTTOU);
                /* job stops here waiting for SIGCONT */
                goto retry;
        }
        ...save old terminal modes and set new modes...
        sigset(SIGTTIN, onstop);
        sigset(SIGTTOU, onstop);
        sigset(SIGTSTP, onstop);
```

It is necessary to ignore SIGTSTP in this code because otherwise our process could be moved from the foreground to the background in the middle of checking if it is in the foreground. The process holds all the stopping signals in this critical section so no other process in our process group can mess us up by blocking us on one of these signals in the middle of our check. (This code assumes that the command interpreter will not move a process from foreground to background without stopping it; if it did we would have no way of making the check correctly.)

The routine which handles the signal should clear the catch for the stop signal and *kill*(2) the processes in its process group with the same signal. The statement after this *kill* will be executed when the process is later continued with SIGCONT.

Thus the code for the catch routine might look like:

```
        ...
        sigset(SIGTSTP, onstop);
        sigset(SIGTTIN, onstop);
        sigset(SIGTTOU, onstop);
        ...

onstop(signo)
        int signo;
{
        ... restore old terminal state ...
        sigset(signo, SIG_DFL);
        kill(0, signo);
        /* stop here until continued */
        sigset(signo, onstop);
        ... restore our special terminal state ...
}
```

This routine can also be used to simulate a stop signal.

If a process does not need to save and restore state when it is stopped, but wishes to be notified when it is continued after a stop it can catch the SIGCONT signal; the SIGCONT handler will be run when the process is continued.

Processes which lock data bases such as the password file should ignore SIGTTIN, SIGTTOU, and SIGTSTP signals while the data bases are being manipulated. While a process is ignoring SIGTTIN signals, reads which would normally have hung will return end-of-file; writes which would normally have caused SIGTTOU signals are instead permitted while SIGTTOU is ignored.

Interrupt-level process handling.

Using the mechanisms of *sigset*(3J) it is possible to handle process state changes as they occur by providing an interrupt-handling routine for the SIGCHLD signal which occurs whenever the status of a child process changes. A signal handler for this signal is established by:

```
sigset(SIGCHLD, onchild);
```

The shell or other process would then await a change in child status with code of the form:

```
recheck:
        sighold(SIGCHLD);           /* start critical section */
        if (no children to process) {
                sigpause(SIGCHLD); /* release SIGCHLD and pause */
                goto recheck;
        }
        sigrelse(SIGCHLD);          /* end critical region */
        /* now have a child to process */
```

Here we are using *sighold* to temporarily block the SIGCHLD signal during the checking of the data structures telling us whether we have a child to process. If we didn't block the signal we would have a race condition since the signal might corrupt our decision by arriving shortly after we had finished checking the condition but before we paused.

If we need to wait for something to happen, we call *sigpause* which automatically releases the hold on the SIGCHLD signal and waits for a signal to occur by starting a *pause*(2). Otherwise we simply release the SIGCHLD signal and process the child. *Sigpause* is similar to the PDP-11 *wait* instruction, which returns the priority of the processor to the base level and idles waiting for an interrupt.

It is important to note that the long-standing bug in the signal mechanism which would have lost a SIGCHLD signal which occurred while the signal was blocked has been fixed. This is because *sighold* uses the SIG_HOLD signal set of *sigsys*(2J) to prevent the signal action from being taken without losing the signal if it occurs. Similarly, a signal action set with *sigset* has the signal held while the action routine is running, much as a the interrupt priority of the processor is raised when a device interrupt is taken.

In this interrupt driven style of termination processing it is necessary that the *wait* calls used to retrieve status in the SIGCHLD signal handler not block. This is because a single invocation of the SIGCHLD handler may indicate an arbitrary number of process status changes: signals are not queued. This is similar to the case in a disk driver where several drives on a single controller may report status at once, while there is only one interrupt taken. It is even possible for no children to be ready to report status when the SIGCHLD handler is invoked, if the signal was posted while the SIGCHLD handler was active, and the child was noticed due to a SIGCHLD initially sent for another process. This causes no problem, since the handler will be called whenever there is work to do; the handler just has to collect all information by calling *wait3* until it says no more information is available. Further status changes are guaranteed to be reflected in another SIGCHLD handler call.

Restarting system calls.

In older versions of UNIX "slow" system calls were interrupted when signals occurred, returning EINTR. The new signal mechanism *sigset*(3J) normally restarts such calls rather than interrupting them. To summarize: *pause* and *wait* return error EINTR (as before), *ioctl* and *wait3* restart, and *read* and *write* restart unless some data was read or written in which case they return indicating how much data was read or written. In programs which use the older *signal*(2) mechanisms, all of these calls return EINTR if a signal occurs during the call.

SEE ALSO

csh(1), ioctl(2), killpg(2), setpgrp(2), sigsys(2J), wait2(2J), signal(2), tty(4)

RESTRICTIONS

The job control facilities are not available in standard version 7 UNIX. These facilities are

still under development and may change in future releases of the system as better interprocess communication facilities and support for virtual terminals become available. The options and specifications of these system calls and even the calls themselves are thus subject to change.

NAME

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

SYNOPSIS

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

double jn(n, x);
double x;

double y0(x)
double x;

double y1(x)
double x;

double yn(n, x)
double 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.

RESTRICTIONS

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 new-line. *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

pkopen, pkclose, pkread, pkwrite, pkfail – packet driver simulator

SYNOPSIS

char *pkopen(fd)

pkclose(ptr)
char *ptr;

pkread(ptr, buffer, count)
char *ptr, *buffer;

pkwrite(ptr, buffer, count)
char *ptr, *buffer;

pkfail()

DESCRIPTION

These routines are a user-level implementation of the full-duplex end-to-end communication protocol described in *pk*(4). If *fd* is a file descriptor open for reading and writing, *pkopen* carries out the initial synchronization and returns an identifying pointer. The pointer is used as the first parameter to *pkread, pkwrite,* and *pkclose*.

Pkread, pkwrite and *pkclose* behave analogously to *read, write* and *close*(2). However, a write of zero bytes is meaningful and will produce a corresponding read of zero bytes.

SEE ALSO

pk(4), pkon(2)

DIAGNOSTICS

Pkfail is called upon persistent breakdown of communication. *Pkfail* must be supplied by the user.
Pkopen returns a null (0) pointer if packet protocol can not be established.
Pkread returns −1 on end of file, 0 in correspondence with a 0-length write.

RESTRICTIONS

The Packet Driver is not supported by the ULTRIX-11 operating system.

This simulation of *pk*(4) leaves something to be desired in needing special read and write routines, and in not being inheritable across calls of *exec*(2). Its prime use is on systems that lack *pk*.
These functions use *alarm*(2); simultaneous use of *alarm* for other puposes may cause trouble.

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 graphics stream on standard output for *plot*(1) filters
–lt300 GSI 300 terminal
–lt300s
GSI 300S terminal
–lt450 DASI 450 terminal
–lt4014
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.

RESTRICTIONS

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 contains 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 π to 5 decimals:

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

SEE ALSO

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

RESTRICTIONS

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.

RESTRICTIONS

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*

RESTRICTIONS

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

regex, regcmp − regular expression compile/execute

SYNOPSIS

```
char *regcmp(string1[,string2, ...],0);
char *string1, *string2, ...;

char *regex(re,subject[,ret0, ...]);
char *re, *subject, *ret0, ...;
```

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. *Malloc*(3C) is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A zero return from *regcmp* indicates an incorrect argument. *Regcmp*(1) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns zero on failure or a pointer to the next unmatched character on success. A global character pointer *_locl* points to where the match began. *Regcmp* and *regex* were mostly borrowed from the editor, *ed*(1) however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[]*.^ These symbols retain their current meaning.

$ Matches the end of the string, **\n** matches the new-line.

− Within brackets the minus means *through*. For example, **[a−z]** is equivalent to **[abcd...xyz]**. The − can appear as itself only if used as the last or first character. For example, the character class expression **[]−**] matches the characters] and −.

+ A regular expression followed by + means *one or more times*. For example, **[0−9]+** is equivalent to **[0−9][0−9]***.

{m} {m,} {m,u}
Integer values enclosed in **{}** indicate the number of times the preceding regular expression is to be applied. *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

(...)$*n* The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+ 1)*th argument following the subject argument. At present, at most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

(...) Parentheses are used for grouping. An operator, e.g. *, +, **{}**, can work on a single character or a regular expression enclosed in parenthesis. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
        ...
newcursor = regex((ptr=regcmp("^\n",0)),cursor);
free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
char *newcursor, *name;
        ...
name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0",0);
newcursor = regex(name,"123Testing321",ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (cursor+11). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
        ...
newcursor = regex(name,string);
```

This example applies a precompiled regular expression in **file.i** (see *regcmp*(1)) against *string*.

This routine is kept in **/lib/libPW.a**.

SEE ALSO

ed(1), regcmp(1), free(3C), malloc(3C).

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required. The following user-supplied replacement for *malloc*(3C) reuses the same vector saving time and space:

```
/* user's program */
        ...
malloc(n) {
static int rebuf[256];
        return &rebuf;
}
```

NAME

scanf, fscanf, sscanf – 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.

RESTRICTIONS

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 completely 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

sigset, signal, sighold, sigignore, sigrelse, sigpause – manage signals

SYNOPSIS

#include <signal.h>
void action();
int action(); *(if void is not supported)*
int sig;

sigset(sig, action)
signal(sig, action)

sighold(sig)
sigignore(sig)
sigrelse(sig)

sigpause(sig)

cc ... –ljobs

DESCRIPTION

This is a package of signal management functions to manage the signals as described in *sigsys*(2J). These functions are not available in most versions of UNIX, and should not be used when the mechanisms of *signal*(2) would suffice, as they would then impair portability. These functions are contained in the *jobs* library, obtained by specifying the loader option **–ljobs**.

Sigset is used to provide a default signal handler for signal *sig*. This function is remembered across subsequent calls to the other functions, and need not be specified again. After *sigset* instances of *sig* will cause an interrupt to be taken at *func*, with the signal then held so that recursive trapping due to the signal will not occur. During normal return from *func*, the routines arrange for the signal action to be restored so that subsequent signals will also trap to *func*. If a non-local exit is to be taken, then *sigrelse* must be called to un-hold the signal action, restoring the original catch. *Func* may also be specified as SIG_DFL, SIG_IGN or SIG_HOLD, as described in *sigsys*(2J). The value specified on the previous call to *sigset* is returned; if *sigset* has never been called, then the default action inherited from the system is returned.

Signal is like *sigset*, but the signal will not be held when the action routine is called; rather it will have reverted to SIG_DFL. This is generally unsafe, but is included for backwards compatibility to the old signal mechanism. It should not be used.

Sighold and *sigrelse* may be used to block off *sig* in a piece of code where it cannot be tolerated. After *sigrelse* the catch initially set with *sigset* will be restored.

Sigignore can be used to temporarily set the action for *sig* to ignore the signal. If the signal had been held before the call to *sigignore*, any pending instance of the signal will be discarded.

Sigpause may be used by a routine which wishes to check for some condition produced at interrupt level by the *sig* signal, and then to pause waiting for the condition to arise with the catch of the signal enabled. In correct usage it must be preceded by an instance of *sighold* to block the signal. *Sigpause* is like *pause* in that it will return after *any* signal is processed. The usual thing to do then is to reenable the hold with *sighold*, check the condition again, and *sigpause* again if the condition has not arisen.

SEE ALSO

signal(2), sigsys(2J), intro(3J), tty(4)

RESTRICTIONS

Sighold and *sigrelse* do not nest; the first *sigrelse* restores the default catch.

These functions store information in data space. You thus must call *sigsys*(2J) rather than any of *sigset* or *signal* after a *vfork*(2) in the child which is to then *exec*.

NAME

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

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x, y)
double 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 pi.

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 pi.

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.

RESTRICTIONS

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

NAME

sinh, cosh, tanh − hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

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.

RESTRICTIONS

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

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – terminal independent operation routines

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base *termcap*(5). These are low level routines; see *curses*(3) for a higher level package.

Tgetent extracts the entry for terminal *name* into the buffer at *bp*. *Bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to *tgetnum*, *tgetflag*, and *tgetstr*. *Tgetent* returns −1 if it cannot open the *termcap* file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type **name** is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a path name rather than */etc/termcap*. This can speed up entry into programs that call *tgetent*, as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file */etc/termcap*.

Tgetnum gets the numeric value of capability *id*, returning −1 if is not given for the terminal. *Tgetflag* returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. *Tgetstr* gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in *termcap*(5), except for cursor addressing and padding information.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. (Programs which call tgoto should be sure to turn off the XTABS bit(s), since tgoto may now output a tab. Note that programs using termcap should in general turn off XTABS anyway since some terminals use control I for other functions, such as nondestructive space.) If a % sequence is given which is not understood, then *tgoto* returns "OOPS".

Tputs decodes the leading padding information of the string *cp; affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by *stty (2)*. The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libtermlib.a	−ltermlib library
/etc/termcap	data base

SEE ALSO

ex(1), curses(3), termcap(5)

AUTHOR

William Joy

RESTRICTIONS

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 nu 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.

RESTRICTIONS

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.

Chapter 4

Special files

NAME

newtty – summary of the 'new' tty driver

USAGE

stty new

stty new crt

DESCRIPTION

This is a summary of the new tty driver, described completely, with the old terminal driver, in *tty*(4). The new driver is largely compatible with the old but provides additional functionality for job control.

CRTs and printing terminals.

The new terminal driver is normally set differently on CRTs and on printing terminals. On CRTs at speeds of 1200 baud or greater it normally erases input characters physically with backspace-space-backspace when they are erased logically; at speed under 1200 baud this is often unreasonably slow, so the cursor is normally merely moved to the left. This is the behavior after a "stty new crt"; to have the tty driver always erase the characters use "stty new crt crterase crtkill"; to have the characters remain even at 1200 baud or greater use "stty new crt –crterase –crtkill".

On printing terminals the command "stty new prterase" should be given. Logically erased characters are then echoed printed backwards between a '\' and an '/' character.

Other terminal modes are possible, but less commonly used; see *tty*(4) and *stty*(1) for details.

Input editing and output control.

When preparing input the character # (normally changed to ^H using *stty*(1)) erases the last input character, ^W the last input word, and the character @ (often changed to ^U) erases the entire current input line. A ^R character causes the pending input to be retyped. Lines are terminated by a return or a newline; a ^D at the beginning of a line generates an end-of-file.

Control characters echo as ^x when typed, for some x; the delete character is represented as ^?.

The character ^V may be typed before *any* character so that it may be entered without its special effect. For backwards compatibility with the old tty driver the character '\' prevents the special meaning of the character and line erase characters, much as ^V does.

Output is suspended when a ^S character is typed and resumed when a ^Q character is typed. Output is discarded after a ^O character is typed until another ^O is type, more input arrives, or the condition is cleared by a program (such as the shell just before it prints a prompt.)

Signals.

A non-interactive program is interrupted by a ^? (delete); this character is often reset to ^C using *stty*(1). A quit ^\ character causes programs to terminate like ^? does, but also causes a *core* image file to be created which can then be examined with a debugger. This is often used to stop runaway processes. Interactive programs often catch interrupts and return to their command loop; only the most well debugged programs catch quits.

Programs may be stopped by hitting ^Z, which returns control to the shell. They may then be resumed using the job control mechanisms of the shell. The character ^Y is like ^Z but takes effect when read rather then when typed; it is much less frequently used.

See *tty*(4) for a more complete description of the new terminal driver.

SEE ALSO

csh(1), stty(1), tty(4)

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

NAME

tty – general terminal interface

DESCRIPTION

This section describes both a particular special file, and the general nature of the terminal interface.

The file */dev/tty* is, in each process, a synonym for the control terminal associated with that process. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

As for terminals in general: all of the low-speed asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of the interface.

When a terminal file is opened, it causes the process to wait until a connection is established. In practice user's programs seldom open these files; they are opened by *init* and become a user's input and output file. The very first terminal file open in a process becomes the *control terminal* for that process. The control terminal plays a special role in handling quit or interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork*, even if the control terminal is closed. The set of processes that thus share a control terminal is called a *process group*; all members of a process group receive certain signals together, see DEL below and *kill*(2).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently this limit is 256 characters. When the input limit is reached all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not however necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information. There are special modes, discussed below, that permit the program to read each character as typed without waiting for a full line.

During input, erase and kill processing is normally done. By default, the character '#' erases the last character typed, except that it will not erase beyond the beginning of a line or an EOT. By default, the character '@' kills the entire line up to the point where it was typed, but not beyond an EOT. Both these characters operate on a keystroke basis independently of any backspacing or tabbing that may have been done. Either '@' or '#' may be entered literally by preceding it by '\'; the erase or kill character remains, but the '\' disappears. These two characters may be changed to others.

When desired, all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by '\'. In addition, the following escape sequences can be generated on output and accepted on input:

for	use
`	\'
|	\!
~	\^
{	\(
}	\)

Certain ASCII control characters have special meaning. These characters are not passed to a reading program except in raw mode where they lose their special character. Also, it is possible to change these characters from the default; see below.

EOT (Control-D) may be used to generate an end of file from a terminal. When an EOT is received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOT is discarded. Thus if there are no characters waiting, which is to say the EOT occurred at the beginning of a line, zero characters will be passed back, and this is the standard end-of-file indication.

DEL (Rubout) is not passed to a program but generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location. See *signal*(2).

FS (Control-\ or control-shift-L) generates the *quit* signal. Its treatment is identical to the interrupt signal except that unless a receiving process has made other arrangements it will not only be terminated but a core image file will be generated.

DC3 (Control-S) delays all printing on the terminal until something is typed in.

DC1 (Control-Q) restarts printing after DC3 without generating any input to a program.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a *hangup* signal is sent to all processes with the terminal as control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file on their input can terminate appropriately when hung up on.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. Even parity is always generated on output. The EOT character is not transmitted (except in raw mode) to prevent terminals that respond to it from hanging up.

Several *ioctl*(2) calls apply to terminals. Most of them use the following structure, defined in <*sgtty.h*>:

```
struct sgttyb {
        char    sg_ispeed;
        char    sg_ospeed;
        char    sg_erase;
        char    sg_kill;
        int     sg_flags;
};
```

The *sg_ispeed* and *sg_ospeed* fields describe the input and output speeds of the device according to the following table, which corresponds to the DEC DH-11 interface. If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are as defined in <*sgtty.h*>.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud

B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A
EXTB	15	External B

In the current configuration, only 110, 150, 300 and 1200 baud are really supported on dial-up lines. Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The *sg_erase* and *sg_kill* fields of the argument structure specify the erase and kill characters respectively. (Defaults are # and @.)

The *sg_flags* field of the argument structure contains several bits that determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection
BSDELAY	0100000	Select backspace delays (not implemented):
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0100000	
CRDELAY	0030000	Select carriage-return delays:
CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	
TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0001000	
TAB2	0004000	
XTABS	0006000	
NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	
EVENP	0000200	Even parity allowed on input (most terminals)
ODDP	0000100	Odd parity allowed on input
RAW	0000040	Raw mode: wake up on all characters, 8-bit interface
CRMOD	0000020	Map CR into LF; echo LF or CR as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map upper case to lower on input
CBREAK	0000002	Return each character as soon as typed
TANDEM	0000001	Automatic flow control

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

Backspace delays are currently ignored but might be used for Terminet 300's.

If a form-feed/vertical tab delay is specified, it lasts for about 2 seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is unimplemented and is 0.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 is unimplemented and is 0.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Type 3, called XTABS, is not a delay at all but causes tabs to be replaced by the appropriate number of spaces on output.

Characters with the wrong parity, as determined by bits 200 and 100, are ignored.

In raw mode, every character is passed immediately to the program without waiting until a full line has been typed. No erase or kill processing is done; the end-of-file indicator (EOT), the interrupt character (DEL) and the quit character (FS) are not treated specially. There are no delays and no echoing, and no replacement of one character for another; characters are a full 8 bits for both input and output (parity is up to the program).

Mode 020 causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (for terminals with a new-line function).

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line, but quit and interrupt work, and output delays, case-translation, CRMOD, XTABS, ECHO, and parity work normally. On the other hand there is no erase or kill, and no special treatment of \ or EOT.

TANDEM mode causes the system to produce a stop character (default DC3) whenever the input queue is in danger of overflowing, and a start character (default DC1) when the input queue has drained sufficiently. It is useful for flow control when the 'terminal' is actually another machine that obeys the conventions.

Several *ioctl* calls have the form:

```
#include <sgtty.h>

ioctl(fildes, code, arg)
struct sgttyb *arg;
```

The applicable codes are:

TIOCGETP
: Fetch the parameters associated with the terminal, and store in the pointed-to structure.

TIOCSETP
: Set the parameters according to the pointed-to structure. The interface delays until output is quiescent, then throws away any unread characters, before changing the modes.

TIOCSETN
: Set the parameters but do not delay or flush input. Switching out of RAW or CBREAK mode may cause some garbage input.

With the following codes the *arg* is ignored.

TIOCEXCL
: Set "exclusive-use" mode: no further opens are permitted until the file has been closed.

TIOCNXCL
: Turn off "exclusive-use" mode.

TIOCHPCL
: When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

TIOCFLUSH
: All characters waiting in input or output queues are flushed.

The following codes affect characters that are special to the terminal interface. The argument is a pointer to the following structure, defined in *<sgtty.h>*:

```
struct tchars {
        char    t_intrc;        /* interrupt */
        char    t_quitc;                /* quit */
        char    t_startc;       /* start output */
        char    t_stopc;        /* stop output */
        char    t_eofc;         /* end-of-file */
        char    t_brkc;         /* input delimiter (like nl) */
};
```

The default values for these characters are DEL, FS, DC1, DC3, EOT, and −1. A character value of −1 eliminates the effect of that character. The *t_brkc* character, by default −1, acts like a new-line in that it terminates a 'line,' is echoed, and is passed to the program. The 'stop' and 'start' characters may be the same, to produce a toggle effect. It is probably counterproductive to make other special characters (including erase an kill) identical.

The calls are:

TIOCSETC
: Change the various special characters to those given in the structure.

TIOCSETP
: Set the special characters to those given in the structure.

FILES

/dev/tty
/dev/tty*
/dev/console

SEE ALSO

getty(8), stty (1), signal(2), ioctl(2)

BUGS

Half-duplex terminals are not supported.

The terminal handler has clearly entered the race for ever-greater complexity and generality. It's still not complex and general enough for TENEX fans.

NAME

tty – general terminal interface

DESCRIPTION

This section describes both a particular special file **/dev/tty** and the terminal drivers used for conversational computing.

Line disciplines.

The system provides different *line disciplines* for controlling communications lines. In this version of the system there are two disciplines available:

old The old (standard) terminal driver. This is used when using the standard shell *sh*(1) and for compatibility with other standard version 7 UNIX systems.

new A newer terminal driver, with features for job control; this must be used when using *csh*(1). See *newtty*(1) for a short user-level summary.

Line discipline switching is accomplished with the TIOCSETD *ioctl*:

int ldisc = LDISC; ioctl(filedes, TIOCSETD, &ldisc);

where LDISC is OTTYDISC for the standard tty driver and NTTYDISC for the new driver. The standard (currently old) tty driver is discipline 0 by convention. The current line discipline can be obtained with the TIOCGETD ioctl. Pending input is discarded when the line discipline is changed.

All of the low-speed asynchronous communications ports can use any of the available line disciplines, no matter what hardware is involved.

The control terminal.

When a terminal file is opened, it causes the process to wait until a connection is established. In practice, user programs seldom open these files; they are opened by *init*(8) and become a user's standard input and output file.

If a process which has no control terminal opens a terminal file, then that terminal file becomes the control terminal for that process. The control terminal is thereafter inherited by a child process during a *fork*(2), even if the control terminal is closed.

The file **/dev/tty** is, in each process, a synonym for a *control terminal* associated with that process. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

Process groups.

As described more completely in *jobs*(3), command processors such as *csh*(1) can arbitrate the terminal between different *jobs* by placing related jobs in a single process group and associating this process group with the terminal. A terminal's associated process group may be set using the TIOCSPGRP *ioctl*(2):

ioctl(fildes, TIOCSPGRP, &pgrp)

or examined using TIOCGPGRP rather than TIOCSPGRP, returning the current process group in *pgrp*. The new terminal driver aids in this arbitration by restricting access to the terminal by processes which are not in the current process group; see **Job access control** below.

Modes.

The terminal drivers have three major modes, characterized by the amount of processing on the input and output characters:

cooked The normal mode. In this mode lines of input are collected and input editing is done. The edited line is made available when it is completed by a newline or when an EOT (control-D, hereafter ^D) is entered. A carriage return is usually made synonymous with newline in this mode, and replaced with a newline whenever it is typed. All driver functions (input editing, interrupt generation, output processing such as delay generation and tab expansion, etc.) are available in this mode.

CBREAK
This mode eliminates the character, word, and line editing input facilities, making the input character available to the user program as it is typed. Flow control, literal-next and interrupt processing are still done in this mode. Output processing is done.

RAW This mode eliminates all input processing and makes all input characters available as they are typed; no output processing is done either. TANDEM mode is available for input flow control, however; see below. Characters are a full 8 bits (parity is up to the program).

The style of input processing can also be very different when, in the new terminal driver, a process asks for notification via a SIGTTIN *signal*(2) when input is ready to be read from the control terminal. In this case a *read*(2) from the control terminal will never block but rather return an error indication (EIO) if there is no input available.

Input editing.

A UNIX terminal ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently this limit is 256 characters. In the old terminal driver all the saved characters are thrown away when the limit is reached, without notice; the new driver simply refuses to accept any further input, and rings the terminal bell.

Input characters are normally accepted in either even or odd parity with the parity bit being stripped off before the character is given to the program. By clearing either the EVEN or ODD bit in the flags word it is possible to have input characters with that parity discarded (see the **Summary** below.)

In all of the line disciplines, it is possible to simulate terminal input using the TIOCSTI *ioctl*, which takes, as its third argument, the address of a character. The system pretends that this character was typed on the argument terminal, which must be the control terminal except when executed by the superuser (this call is not in standard version 7 UNIX).

Input characters are normally echoed by putting them in an output queue as they arrive. This may be disabled by clearing the ECHO bit in the flags word using the *stty*(2) call or the TIOCSETN or TIOCSETP *ioctl*s (see the **Summary** below).

In cooked mode, terminal input is processed in units of lines. A program attempting to read will normally be suspended until an entire line has been received (but see the description of SIGTTIN in **Modes** above and FIONREAD in **Summary** below.) No matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, line editing is normally done, with the character '#' logically erasing the last character typed and the character '@' logically erasing the entire current input line. These are often reset on crt's, with ^H replacing #, and ^U replacing @. These characters never erase beyond the beginning of the current input line or an ^D. These characters may be entered literally by preceding them with '\'; in the old teletype driver both the '\' and the character entered literally will appear on the screen; in the new driver the '\' will normally disappear.

The drivers normally treat either a carriage return or a newline character as terminating an input line, replacing the return with a newline and echoing a return and a line feed. If the CRMOD bit is cleared in the local mode word then the processing for carriage return is disabled, and it is simply echoed as a return, and does not terminate cooked mode input.

In the new driver there is a literal-next character ^V which can be typed in both cooked and CBREAK mode preceding any character to prevent its special meaning. This is to be preferred to the use of '\' escaping erase and kill characters, but '\' is (at least temporarily) retained with its old function in the new driver for historical reasons.

The new terminal driver also provides two other editing characters in normal mode. The word-erase character, normally ^W, erases the preceding word, but not any spaces before it. For the purposes of ^W, a word is defined as a sequence of non-blank characters, with tabs counted as blanks. Finally, the reprint character, normally ^R, retypes the pending input beginning on a new line. Retyping occurs automatically in cooked mode if characters which would normally be erased from the screen are fouled by program output.

Input echoing and redisplay

In the old terminal driver, nothing special occurs when an erase character is typed; the erase character is simply echoed. When a kill character is typed it is echoed followed by a new-line (even if the character is not killing the line, because it was preceded by a '\'!).

The new terminal driver has several modes for handling the echoing of terminal input, controlled by bits in a local mode word.

Hardcopy terminals. When a hardcopy terminal is in use, the LPRTERA bit is normally set in the local mode word. Characters which are logically erased are then printed out backwards preceded by '\' and followed by '/' in this mode.

Crt terminals. When a crt terminal is in use, the LCRTBS bit is normally set in the local mode word. The terminal driver then echoes the proper number of erase characters when input is erased; in the normal case where the erase character is a ^H this causes the cursor of the terminal to back up to where it was before the logically erased character was typed. If the input has become fouled due to interspersed asynchronous output, the input is automatically retyped.

Erasing characters from a crt. When a crt terminal is in use, the LCRTERA bit may be set to cause input to be erased from the screen with a "backspace-space-backspace" sequence when character or word deleting sequences are used. A LCRTKIL bit may be set as well, causing the input to be erased in this manner on line kill sequences as well.

Echoing of control characters. If the LCTLECH bit is set in the local state word, then non-printing (control) characters are normally echoed as ^X (for some X) rather than being echoed unmodified; delete is echoed as ^?.

The normal modes for using the new terminal driver on crt terminals are speed dependent. At speeds less than 1200 baud, the LCRTERA and LCRTKILL processing is painfully slow, so *stty*(1) normally just sets LCRTBS and LCTLECH; at speeds of 1200 baud or greater all of these bits are normally set. *Stty*(1) summarizes these option settings and the use of the new terminal driver as "newcrt."

Output processing.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. (As noted above, input characters are normally echoed by putting them in the output queue as they arrive.) When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. Even parity is normally generated on output. The EOT character is not transmitted in cooked mode to prevent terminals that respond to it from hanging up; programs using raw or cbreak mode should be careful.

The terminal drivers provide necessary processing for cooked and CBREAK mode output including delay generation for certain special characters and parity generation. Delays are available after backspaces ^H, form feeds ^L, carriage returns ^M, tabs ^I and newlines ^J. The driver will also optionally expand tabs into spaces, where the tab stops are assumed to be set every eight columns. These functions are controlled by bits in the tty flags word; see **Summary** below.

The terminal drivers provide for mapping between upper and lower case on terminals lacking lower case, and for other special processing on deficient terminals.

Finally, in the new terminal driver, there is a output flush character, normally ^O, which sets the LFLUSHO bit in the local mode word, causing subsequent output to be flushed until it is cleared by a program or more input is typed. This character has effect in both cooked and CBREAK modes and causes pending input to be retyped if there is any pending input. Ioctls to flush the characters in the input and output queues, TIOCFLUSH, and to return the number of characters still in the output queue, TIOCOUTQ are also available.

Upper case terminals and Hazeltines

If the LCASE bit is set in the tty flags, then all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by '\'. In addition, the following escape sequences can be generated on output and accepted on input:

for	`	|	~	{	}
use	\'	\!	\^	\(	\)

To deal with Hazeltine terminals, which do not understand that ~ has been made into an ASCII character, the LTILDE bit may be set in the local mode word when using the new terminal driver; in this case the character ~ will be replaced with the character ` on output.

Flow control.

There are two characters (the stop character, normally ^S, and the start character, normally ^Q) which cause output to be suspended and resumed respectively. Extra stop characters typed when output is already stopped have no effect, unless the start and stop characters are made the same, in which case output resumes.

A bit in the flags word may be set to put the terminal into TANDEM mode. In this mode the system produces a stop character (default ^S) when the input queue is in danger of overflowing, and a start character (default ^Q) when the input has drained sufficiently. This mode is useful when the terminal is actually another machine that obeys the conventions.

Line control and breaks.

There are several *ioctl* calls available to control the state of the terminal line. The TIOCSBRK *ioctl* will set the break bit in the hardware interface causing a break condition to exist; this can be cleared (usually after a delay with *sleep*(3)) by TIOCCBRK. Break conditions in the input are reflected as a null character in RAW mode or as the interrupt

character in cooked or CBREAK mode. The TIOCCDTR *ioctl* will clear the data terminal ready condition; it can be set again by TIOCSDTR. TIOCSBRK, TIOCBRK, TIOCSDTR and TIOCCDTR are available only where the hardware and device driver are able to support them.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a SIGHUP hangup signal is sent to the processes in the distinguished process group of the terminal; this usually causes them to terminate (the SIGHUP can be suppressed by setting the LNOHANG bit in the local state word of the driver.) Access to the terminal by other processes is then normally revoked, so any further reads will fail, and programs that read a terminal and test for end-of-file on their input will terminate appropriately.

It is possible to ask that the phone line be hung up on the last close with the TIOCHPCL *ioctl*; this is normally done on the outgoing line.

Interrupt characters.

There are several characters that generate interrupts in cooked and CBREAK mode; all are sent the processes in the control group of the terminal, as if a TIOCGPGRP *ioctl* were done to get the process group and then a *killpg*(2) system call were done, except that these characters also flush pending input and output when typed at a terminal (*à la* TIOCFLUSH). The characters shown here are the defaults; the field names in the structures (given below) are also shown. The characters may be changed, although this is not often done.

^? **t_intrc** (Delete) generates a SIGINTR signal. This is the normal way to stop a process which is no longer interesting, or to regain control in an interactive program.

^\ **t_quitc** (FS) generates a SIGQUIT signal. This is used to cause a program to terminate and produce a core image, if possible, in the file **core** in the current directory.

^Z **t_suspc** (EM) generates a SIGTSTP signal, which is used to suspend the current process group.

^Y **t_dstopc** (SUB) generates a SIGTSTP signal as ^Z does, but the signal is sent when a program attempts to read the ^Y, rather than when it is typed.

Job access control.

When using the new terminal driver, if a process which is not in the distinguished process group of its control terminal attempts to read from that terminal its process group is sent a SIGTTIN signal, which normally causes the members of that process group to stop. If, however, the process is ignoring or holding SIGTTIN signal, is an orphan its parent has exited and it has been inherited by the *init*(8) process, or if it is a process in the middle of process creation using *vfork*(2)), it is instead returned an end-of-file. Under older UNIX systems these processes would typically have had their input files reset to **/dev/null**, so this is a compatible change.

When using the new terminal driver with the LTOSTOP bit set in the local modes, a process is prohibited from writing on its control terminal if it is not in the distinguished process group for that terminal. Processes which are holding or ignoring SIGTTOU signals, which are orphans, or which are in the middle of a *vfork*(2) are excepted and allowed to produce output.

Summary of modes.

Unfortunately, due to the evolution of the terminal driver, there are 4 different structures which contain various portions of the driver data. The first of these (**sgttyb**) contains that part of the information largely common between version 6 and version 7 UNIX systems.

The second contains additional control characters added in version 7. The third is a word of local state peculiar to the new terminal driver, and the fourth is another structure of special characters added for the new driver. In the future a single structure may be made available to programs which need to access all this information; most programs need not concern themselves with all this state.

The basic *ioctl*s use the structure defined in < *sgtty.h*> :

```
struct sgttyb {
        char    sg_ispeed;
        char    sg_ospeed;
        char    sg_erase;
        char    sg_kill;
        short   sg_flags;
};
```

The *sg_ispeed* and *sg_ospeed* fields describe the input and output speeds of the device according to the following table, which corresponds to the DEC DH-11 interface. If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are as defined in < *sgtty.h*> .

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A
EXTB	15	External B

In the current configuration, only 110, 150, 300 and 1200 baud are really supported on dial-up lines. Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The *sg_erase* and *sg_kill* fields of the argument structure specify the erase and kill characters respectively. (Defaults are # and @.)

The *sg_flags* field of the argument structure contains several bits that determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection
BSDELAY	0100000	Select backspace delays (not implemented):
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0100000	
CRDELAY	0030000	Select carriage-return delays:

CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	
TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0001000	
TAB2	0004000	
XTABS	0006000	
NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	
EVENP	0000200	Even parity allowed on input (most terminals)
ODDP	0000100	Odd parity allowed on input
RAW	0000040	Raw mode: wake up on all characters, 8-bit interface
CRMOD	0000020	Map CR into LF; echo LF or CR as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map upper case to lower on input
CBREAK	0000002	Return each character as soon as typed
TANDEM	0000001	Automatic flow control

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

Backspace delays are currently ignored but might be used for Terminet 300's.

If a form-feed/vertical tab delay is specified, it lasts for about 2 seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is suitable for the concept-100 and pads lines to be at least 9 characters at 9600 baud.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 is unimplemented and is 0.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Type 3, called XTABS, is not a delay at all but causes tabs to be replaced by the appropriate number of spaces on output.

Input characters with the wrong parity, as determined by bits 0200 and 0100, are ignored in cooked and CBREAK mode.

RAW disables all processing save output flushing with LFLUSHO; full 8 bits of input are given as soon as it is available; all 8 bits are passed on output. A break condition in the input is reported as a null character. If the input queue overflows in raw mode it is discarded; this applies to both new and old drivers.

CRMOD causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (for terminals with a new-line function).

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line; all processing is done except the input editing:

character and word erase and line kill, input reprint, and the special treatment of \ or EOT are disabled.

TANDEM mode causes the system to produce a stop character (default ^S) whenever the input queue is in danger of overflowing, and a start character (default ^Q) when the input queue has drained sufficiently. It is useful for flow control when the 'terminal' is really another computer which understands the conventions.

In addition to the TIOCSETD and TIOCGETD disciplines discussed in **Line disciplines** above, a large number of other *ioctl*(2) calls apply to terminals, and have the general form:

#include <sgtty.h>

ioctl(fildes, code, arg)
struct sgttyb *arg;

The applicable codes are:

TIOCGETP Fetch the basic parameters associated with the terminal, and store in the pointed-to *sgttyb* structure.

TIOCSETP Set the parameters according to the pointed-to *sgttyb* structure. The interface delays until output is quiescent, then throws away any unread characters, before changing the modes.

TIOCSETN Set the parameters like TIOCSETP but do not delay or flush input. Input is not preserved, however, when changing to or from RAW.

With the following codes the *arg* is ignored.

TIOCEXCL Set "exclusive-use" mode: no further opens are permitted until the file has been closed.

TIOCNXCL Turn off "exclusive-use" mode.

TIOCHPCL When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

The remaining calls are not available in vanilla version 7 UNIX. In cases where arguments are required, they are described; *arg* should otherwise be given as 0.

TIOCFLUSH If *arg* is 0, all characters waiting in input or output queues are flushed. If *arg* is FREAD (defined in <sys/file.h>), all characters in the input queues are flushed. If *arg* is FWRITE (defined in <sys/file.h>), all characters in the output queues are flushed.

TIOCSTI The argument is the address of a character which the system pretends was typed on the terminal.

TIOCSBRK The break bit is set in the terminal.

TIOCCBRK The break bit is cleared.

TIOCSDTR Data terminal ready is set.

TIOCCDTR Data terminal ready is cleared.

TIOCGPGRP *Arg* is the address of a word into which is placed the process group number of the control terminal.

TIOCSPGRP *Arg* is a word (typically a process id) which becomes the process group for the control terminal.

FIONREAD Returns in the long integer whose address is *arg* the number of immediately readable characters from the argument unit. This works for files, pipes, and terminals, but not (yet) for multiplexed channels.

The second structure associated with each terminal specifies characters that are special in both the old and new terminal interfaces: The following structure is defined in *< sys/ioctl.h>*, which is automatically included in *< sgtty.h>*:

```
struct tchars {
        char    t_intrc;        /* interrupt */
        char    t_quitc;        /* quit */
        char    t_startc;       /* start output */
        char    t_stopc;        /* stop output */
        char    t_eofc;         /* end-of-file */
        char    t_brkc;         /* input delimiter (like nl) */
};
```

The default values for these characters are ^?, ^\, ^Q, ^S, ^D, and −1. A character value of −1 eliminates the effect of that character. The *t_brkc* character, by default −1, acts like a new-line in that it terminates a 'line,' is echoed, and is passed to the program. The 'stop' and 'start' characters may be the same, to produce a toggle effect. It is probably counter-productive to make other special characters (including erase and kill) identical. The applicable *ioctl* calls are:

TIOCGETC
: Get the special characters and put them in the specified structure.

TIOCSETC Set the special characters to those given in the structure.

The third structure associated with each terminal is a local mode word; except for the LNOHANG bit, this word is interpreted only when the new driver is in use. The bits of the local mode word are:

LCRTBS	000001	Backspace on erase rather than echoing erase
LPRTERA	000002	Printing terminal erase mode
LCRTERA	000004	Erase character echos as backspace-space-backspace
LTILDE	000010	Convert ~ to ` on output (for Hazeltine terminals)
LMDMBUF	000020	Stop/start output when carrier drops
LLITOUT	000040	Suppress output translations
LTOSTOP	000100	Send SIGTTOU for background output
LFLUSHO	000200	Output is being flushed
LNOHANG	000400	Don't send hangup when carrier drops
LETXACK	001000	Diablo style buffer hacking (unimplemented)
LCRTKIL	002000	BS-space-BS erase entire line on line kill
LINTRUP	004000	Generate interrupt SIGTINT when input ready to read
LCTLECH	010000	Echo input control chars as ^X, delete as ^?
LPENDIN	020000	Retype pending input at next read or input character
LDECCTQ	040000	Only ^Q restarts output after ^S, like DEC systems

The applicable *ioctl* functions are:

TIOCLBIS
: *Arg* is the address of a mask which is the bits to be set in the local mode word.

TIOCLBIC
: *Arg* is the address of a mask of bits to be cleared in the local mode word.

TIOCLSET *Arg* is the address of a mask to be placed in the local mode word.

TIOCLGET *Arg* is the address of a word into which the current mask is placed.

The final structure associated with each terminal is the *ltchars* structure which defines interrupt characters for the new terminal driver. Its structure is:

```
struct ltchars {
        char    t_suspc;        /* stop process signal */
        char    t_dstopc;       /* delayed stop process signal */
        char    t_rprntc;       /* reprint line */
        char    t_flushc;       /* flush output (toggles) */
        char    t_werasec;      /* word erase */
        char    t_lnextc;       /* literal next character */
};
```

The default values for these characters are ^Z, ^Y, ^R, ^O, ^W, and ^V. A value of −1 disables the character.

The applicable *ioctl* functions are:

TIOCSLTC *Arg* is the address of an *ltchars* structure which defines the new local special characters.

TIOCGLTC
Arg is the address of an *ltchars* structure into which is placed the current set of local special characters.

FILES

/dev/tty
/dev/tty??
/dev/console

SEE ALSO

csh(1), stty(1), ioctl(2), signal(2), sigsys(2j), stty(2), newtty(4), getty(8), init(8)

RESTRICTIONS

Half-duplex terminals are not supported.

Chapter 5

File formats and conventions

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 NOVL   7

struct    ovlhdr {
          int                   max_ovl;/* maximum ovl size */
          unsigned ov_siz[NOVL];/* size of i'th overlay */
};

#define A_MAGIC1          0407     /* normal (I space only) */
#define A_MAGIC2          0410     /* read-only/shared text */
#define A_MAGIC3          0411     /* separated I&D */
#define A_MAGIC4          0405     /* overlay (NOT USED) */
#define A_MAGIC5          0430     /* overlay text kernel */
#define A_MAGIC6          0430     /* user overlay (shared text) */
#define A_MAGIC7          0431     /* user overlay (separated I&D) */

struct    nlist {    /* symbol table entry */
          char       n_name[8]; /* symbol name */
          int        n_type;    /* type flag */
          unsigned n_value;     /* value */
};
/*
 * Macros that take exec structures as arguments and tell
 * whether the file has a reasonable magic number or
 * offset to text.
 */
#define N_BADMAG(x) \
  (((x).a_magic)!=A_MAGIC1 && ((x).a_magic)!=A_MAGIC2 && \
   ((x).a_magic)!=A_MAGIC3 && ((x).a_magic)!=A_MAGIC4 && \
   ((x).a_magic)!=A_MAGIC5 && ((x).a_magic)!=A_MAGIC6 && \
   ((x).a_magic)!=A_MAGIC7)

#define N_OVMAG(x) \
```

```
(((x).a_magic)==A_MAGIC5 || ((x).a_magic)==A_MAGIC6 || \
 ((x).a_magic)==A_MAGIC7)

#define N_TXTOFF(x) \
  (N_OVMAG(x) ? sizeof (struct ovlhdr) + sizeof (struct exec) \
     : sizeof (struct exec))

                    /* 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 data), 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, 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, 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+St (the size of the text) the start of the relocation information is 020+St+Sd; the start of the symbol table is 020+2(St+Sd) if the relocation information is present, 020+St+Sd 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 (e.g., 'clr x'); if 0, that the reference is to the actual symbol (e.g., 'clr *$x').

The remainder of the relocation word (bits 15-4) contains a symbo number in the case of external references, and is unused otherwise. The first symbol is r mbered 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)

RESTRICTIONS

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

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=:/usr/ucb:/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, IFS.

SEE ALSO

exec(2), sh(1), term(7), login(1), termcap(5)

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

tar – tape archive format

DESCRIPTION

The *tar* command dumps files to and extracts files from magtape in tape archive format. Unlike *tp* tapes, *tar* tapes have no provision for a bootstrap program. The record size for *tar* tapes is always a multiple of 512 bytes. The most common blocking factors are one and 20.

Tape archive format consists of a header record followed by the contents of the file. The header record has the following structure:

```
#define TBLOCK      512
#define NBLOCK      20
#define NAMSIZ      100
union hblock {
        char dummy[TBLOCK];
        struct header {
                char name[NAMSIZ];
                char mode[8];
                char uid[8];
                char gid[8];
                char size[12];
                char mtime[12];
                char chksum[8];
                char linkflag;
                char linkname[NAMSIZ];
                char rdev[6];   /* file type */
        } dbuf;
} dblock, tbuf[NBLOCK];
```

SEE ALSO

tar(1)

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, *e.g.*, by *vi*(1) and *curses*(3). Terminals are described in *termcap* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *termcap*.

Entries in *termcap* consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

CAPABILITIES

(P) indicates padding may be specified
(P*) indicates that padding may be based on no. lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command char in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horiz motion only, line stays same
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode

ei	str		End insert mode; give ":ei=:" if ic
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no cm)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing is
im	bool		Insert mode (enter); give ":im=:" if ic
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by "other" function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of "keypad transmit" mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of "other" keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in "keypad transmit" mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on "other" function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no cm)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
ms	bool		Ok to move while in standout & underline mode
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correct working carriage rtn (DM2500,H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with is)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use cm
ti	str		String to begin programs that use cm
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue

ul	bool	Term. underlines even though doesn't overstrike
up	str	Upline (cursor up)
us	str	Start underscore mode
vb	str	Visible bell (may not move cursor)
ve	str	Sequence to end open/visual mode
vs	str	Sequence to start open/visual mode
xb	bool	Beehive (f1=escape, f2=ctrl C)
xn	bool	A newline is ignored after a wrap (Concept)
xr	bool	Return acts like ce \r \n (Delta Data)
xs	bool	Standout not erased by writing over it (HP 264?)
xt	bool	Tabs destructive, magic so char (Teleray 1061)

A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the *termcap* file as of this writing. (This particular concept entry is outdated, and is used as an example only.)

```
c1|c100|concept100:is=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200:\
:al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :\
:co#80:\
:dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:\
:li#24:mi:nd=\E=:\
:se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and that empty fields may be included for readability (here between the last field on a line and the first field on the next). Capabilities in *termcap* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have two letter codes. For instance, the fact that the Concept has "automatic margins" (i.e. an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '**#**' and then the value. Thus **co** which indicates the number of columns the terminal has gives the value '80' for the Concept.

Finally, string valued capabilities, such as **ce** (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, e.g. '20', or an integer followed by an '*', i.e. '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A **\E** maps to an ESCAPE character, **^x** maps to a control-x for any appropriate **x**, and the sequences **\n \r \t \b \f** give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a ****, and the characters **^** and **** may be given as **\^** and ****. If it is necessary to place a **:** in a capability it must be escaped in octal as **\072**. If it is necessary to place a null character in a string capability it

must be encoded as **\200**. The routines which deal with *termcap* use C strings, and strip the high bits of the output very late so that a **\200** comes out as a **\000** would.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *termcap* and to build up a description gradually, using partial descriptions with *ex* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *termcap* file to describe it or bugs in *ex*. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in */etc/termcap*. TERMCAP can also be set to the termcap entry itself to avoid reading the file when starting up the editor. (This only works on version 7 systems.)

Basic capabilities

The number of columns on each line for the terminal is given by the **co** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **li** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, then this is given by the **cl** string capability. If the terminal can backspace, then it should have the **bs** capability, unless a backspace is accomplished by a character other than ^**H** (ugh) in which case you should give this character as the **bc** string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability.

A very important point here is that the local cursor motions encoded in *termcap* are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the **am** capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch selectable automatic margins, the *termcap* file usually assumes that this is on, i.e. **am**.

These capabilities suffice to describe hardcopy and "glass-tty" terminals. Thus the model 33 teletype is described as

```
t3|33|tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as

```
cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80
```

Cursor addressing

Cursor addressing in the terminal is described by a **cm** string capability, with *printf*(3s) like escapes **%x** in it. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the **cm** string is thought of as being a function, then its arguments are the line and then the column to which motion is desired, and the **%** encodings have the following meanings:

%d	as in *printf*, 0 origin
%2	like %2d
%3	like %3d
%.	like %c
%+x	adds *x* to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %

%n exclusive or row and column with 0140 (DM2500)
%B BCD (16*(x/10)) + (x%10), no output.
%D Rev coding (x-2*(x%16)), no output (Delta Data)

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cm** capability is "cm=6\E&%r%2c%2Y". The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, "cm=^T%.%.". Terminals which use "%." need to be able to backspace the cursor (**bs** or **bc**), and to move the cursor up one line on the screen (**up** introduced below). This is necessary because it is not always safe to transmit \t, \n ^D and \r, as the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "cm=\E=%+ %+ ".

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, then this sequence should be given as **nd** (non-destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as **up**. If the terminal has no cursor addressing capability, but can home the cursor (to very upper left corner of screen) then this can be given as **ho**; similarly a fast way of getting to the lower left hand corner can be given as **ll**; this may involve going up with **up** from the home position, but the editor will never do this itself (unless **ll** does) because it makes no assumption about the effect of moving up from the home position.

Area clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **ce**. If the terminal can clear from the current position to the end of the display, then this should be given as **cd**. The editor only uses **cd** from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **al**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl**; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, then this can be given as **sb**, but just **al** suffices. If the terminal can retain display memory above then the **da** capability should be given; if display memory can be retained below then **db** should be given. These let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with **sb** may bring down non-blank lines.

Insert/delete character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *termcap*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "abc def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to

shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode (give this, with an empty value also if you gave **im** so). Now give as **ic** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ic**, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g. if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify delete mode by giving **dm** and **ed** to enter and exit delete mode, and **dc** to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as **so** and **se** respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining – half bright is not usually an acceptable "standout" mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **ug** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **us** and **ue** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

Many terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **vb**; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of *ex*, this can be given as **vs** and **ve**, sent at the start and end of these modes respectively. These can be used to change, e.g., from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as **ti** and **te**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor

addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kl, kr, ku, kd,** and **kh** respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0, k1, ..., k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0, l1, ..., l9**. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the *termcap* 2 letter codes can be given in the **ko** capability, for example, ":ko=cl,ll,sf,sb:", which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The **ma** entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with **kl, kr, ku, kd,** and **kh**. It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and **H** for **kh**. For example, the mime would be **:ma=^Kj^Zk^Xl:** indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pc**.

If tabs on the terminal require padding, or if the terminal uses a character other than **^I** to tab, then this can be given as **ta**.

Hazeltine terminals, which don't allow '~' characters to be printed should indicate **hz**. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **xx**.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, **is** will be printed before **if**. This is useful where **if** is */usr/lib/tabset/std* but **is** clears the tabs first.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since *termlib* routines search the entry from left to right, and since the tc

capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be canceled with **xx@** where xx is the capability. For example, the entry

hn|2621nl:ks@:ke@:tc=2621:

defines a 2621nl that does not have the **ks** or **ke** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), curses(3), termcap(3), tset(1), vi(1), ul(1), more(1)

AUTHOR

William Joy
Mark Horton added underlining and keypad support

RESTRICTIONS

Ex allows only 256 characters for string capabilities, and the routines in *termcap*(3) do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024.

The **ma**, **vs**, and **ve** entries are specific to the *vi* program.

Not all programs support all entries. There are entries that are not supported by any program.

Chapter 6

Games

NAME

banner – print large banner on printer

SYNOPSIS

/usr/games/banner [**–w***n*] message ...

DESCRIPTION

Banner prints a large, high quality banner on the standard output. If the message is omitted, it prompts for and reads one line of its standard input. If **–w** is given, the output is scrunched down from a width of 132 to *n* , suitable for a narrow terminal. If *n* is omitted, it defaults to 80.

The output should be printed on a hard-copy device, up to 132 columns wide, with no breaks between the pages. The volume is enough that you want a printer or a fast hardcopy terminal, but if you are patient, a decwriter or other 300 baud terminal will do.

BUGS

Several ASCII characters are not defined, notably <, >, [,], \, ^, _, {, }, |, and ~. Also, the characters ", ', and & are funny looking (but in a useful way.)

The **–w** option is implemented by skipping some rows and columns. The smaller it gets, the grainier the output. Sometimes it runs letters together.

AUTHOR

Mark Horton

NAME

banner – make long posters

SYNOPSIS

/usr/games/banner

DESCRIPTION

Banner reads the standard input and prints it sideways in huge built-up letters on the standard output.

NAME

ching, fortune – the book of changes and other cookies

SYNOPSIS

/usr/games/ching [hexagram]

/usr/games/fortune

DESCRIPTION

The *I Ching* or *Book of Changes* is an ancient Chinese oracle that has been in use for centuries as a source of wisdom and advice.

The text of the *oracle* (as it is sometimes known) consists of sixty-four *hexagrams*, each symbolized by a particular arrangement of six straight (– – –) and broken (– –) lines. These lines have values ranging from six through nine, with the even values indicating the broken lines.

Each hexagram consists of two major sections. The **Judgement** relates specifically to the matter at hand (E.g., "It furthers one to have somewhere to go.") while the **Image** describes the general attributes of the hexagram and how they apply to one's own life ("Thus the superior man makes himself strong and untiring.").

When any of the lines have the values six or nine, they are moving lines; for each there is an appended judgement which becomes significant. Furthermore, the moving lines are inherently unstable and change into their opposites; a second hexagram (and thus an additional judgement) is formed.

Normally, one consults the oracle by fixing the desired question firmly in mind and then casting a set of changes (lines) using yarrow–stalks or tossed coins. The resulting hexagram will be the answer to the question.

Using an algorithm suggested by S. C. Johnson, the Unix *oracle* simply reads a question from the standard input (up to an EOF) and hashes the individual characters in combination with the time of day, process id and any other magic numbers which happen to be lying around the system. The resulting value is used as the seed of a random number generator which drives a simulated coin–toss divination. The answer is then piped through **nroff** for formatting and will appear on the standard output.

For those who wish to remain steadfast in the old traditions, the oracle will also accept the results of a personal divination using, for example, coins. To do this, cast the change and then type the resulting line values as an argument.

The impatient modern may prefer to settle for Chinese cookies; try *fortune*.

SEE ALSO

It furthers one to see the great man.

DIAGNOSTICS

The great prince issues commands,
Founds states, vests families with fiefs.
Inferior people should not be employed.

RESTRICTIONS

Waiting in the mud
Brings about the arrival of the enemy.

If one is not extremely careful,
Somebody may come up from behind and strike him.
Misfortune.

Chapter 7

Miscellaneous

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	@	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*	∝
< wig	≲	*< ->*	↔	*empty*	∅
= wig	≅	*< = >*	⇔	*member*	∈
star	∗	|	∤	*nomem*	∉
bigstar	✳	|>	⊁	*cup*	∪
= dot	≐	*ang*	∠	*cap*	∩
andsign	∧	*3dot*	⋮	*subset*	⊂
= del	≙	*thf*	∴	*supset*	⊃
oppA	∀	*quarter*		*!subset*	⊆
oppE	∃	*3quarter*		*!supset*	⊇
angstrom	Å	*degree*	°		

FILES

/usr/pub/eqnchar

SEE ALSO

troff(1), eqn(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)

```
        messages
                hardware 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.lock
                lock 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)

RESTRICTIONS

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 pages of this manual. 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**:

*R *nroff*.

*S Change to default type size.

FILES

/usr/lib/tmac/tmac.an
/usr/man/man0/xx

SEE ALSO

troff(1), man(1)

RESTRICTIONS

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