ROS Reference Manual Sections (2) thru (7)

RIDGE

9010E volume 2 of 2

NAME

intro – introduction to system calls and error numbers (callable from C)

SYNTAX

#include <errno.h>

DESCRIPTION

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

All of the possible error numbers are not listed in each system call description because many errors are possible for most of the calls. The following is a complete list of the error numbers and their names as defined in **<errno.h>**.

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

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

3 ESRCH No such process
: No process can be found corresponding to that specified by *pid* in *kill* or *ptrace*.

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

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

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

7 E2IG Arg list too long
: An argument list longer than 5,120 bytes is presented to a member of the *exec* family.

8 ENOEXEC Exec format error
: A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number (see *a.out*(4)).

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

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

11 EAGAIN No more processes
: A *fork*, failed because the system's process table is full or the user is not allowed to create any more processes.

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

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

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

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

16 EBUSY 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). It will also occur if an attempt is made to enable accounting when it is already enabled.

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

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

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

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

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

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

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

24 EMFILE Too many open files
: No process may have more than 64 file descriptors open at a time.

25 ENOTTY Not a typewriter

26 ETXTBSY Text file busy
: An attempt to execute a pure-procedure program which 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 file size (1,082,201,088 bytes) or ULIMIT; see *ulimit*(2).

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

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

31 EMLINK Too many links
  An attempt to make more than the maximum number of links (1000) to a file. This condition normally generates a signal; the error is returned if the signal is ignored.

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

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

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

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

DEFINITIONS

Process ID

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

Parent Process ID

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

Process Group ID

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

Tty Group ID

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

Real User ID and Real Group ID

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

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

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

Effective User ID and Effective Group ID

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

Super-user

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

File Name.

Names consisting of 1 to 16 characters may be used to name an ordinary file, special file or directory.

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

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

Path Name and Path Prefix

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

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

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

where <file-name> is a string of 1 to 16 characters other than the ASCII slash and null, and <dirname> is a string of 1 to 16 characters (other than the ASCII slash and null) that names a directory.

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

A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

Directory.

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

Root Directory and Current Working Directory.

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

File Access Permissions.

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

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

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

Otherwise, the corresponding permissions are denied.

NAME

access – determine accessibility of a file

SYNTAX

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

DESCRIPTION

Path points to a path name naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed as follows:

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

Access to the file is denied if one or more of the following are true:

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

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

The named file does not exist. [ENOENT]

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

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

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

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

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

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits, members of the file's group other than the owner have permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

RETURN VALUE

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

SEE ALSO

chmod(2), stat(2).

NAME

alarm – set a process's alarm clock

SYNTAX

unsigned alarm (sec)
unsigned sec;

DESCRIPTION

Alarm instructs the calling process's alarm clock to send the signal SIGALRM to the calling process after the number of real time seconds specified by *sec* have elapsed; see *signal*(2).

Alarm requests are not stacked; successive calls reset the calling process's alarm clock.

If *sec* is 0, any previously made alarm request is canceled. *Sec* is a virtually unlimited integer.

RETURN VALUE

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

SEE ALSO

pause(2), signal(2).

NAME

brk, sbrk – change data segment space allocation

SYNTAX

int brk (endds)
char *endds;

char *sbrk (incr)
int incr;

DESCRIPTION

Brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment; see *exec*(2). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is set to zero.

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

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

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

> Such a change would result in more space being allocated than is allowed by a system-imposed maximum. [ENOMEM]
>
> Such a change would result in the break value being greater than or equal to the start address of any attached shared memory segment.

RETURN VALUE

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

SEE ALSO

exec(2).

NAME

chdir – change working directory

SYNTAX

int chdir (path)
char *path;

DESCRIPTION

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

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

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

The named directory does not exist. [ENOENT]

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

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

RETURN VALUE

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

SEE ALSO

chroot(2).

NAME

chmod – change mode of file

SYNTAX

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

DESCRIPTION

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

Access permission bits are interpreted as follows:

04000	Set user ID on execution.
02000	Set group ID on execution.
00400	Read by owner
00200	Write by owner
00100	Execute (or search if a directory) by owner
00070	Read, write, execute (search) by group
00007	Read, write, execute (search) by others

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

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

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

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

The named file does not exist. [ENOENT]

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

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

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

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

RETURN VALUE

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

SEE ALSO

chown(2), mknod(2).

NAME

chown – change owner and group of a file

SYNTAX

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

DESCRIPTION

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

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

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

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

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

The named file does not exist. [ENOENT]

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

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

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

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

RETURN VALUE

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

SEE ALSO

chmod(2).

NAME

close – close a file descriptor

SYNTAX

int close (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat, open, dup,* or *fcntl,* system call. *Close* closes the file descriptor indicated by *fildes.*

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

RETURN VALUE

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

SEE ALSO

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

NAME

creat – create a new file or rewrite an existing one

SYNTAX

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

DESCRIPTION

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

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

Upon successful completion, a non-negative integer, namely the file descriptor, is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls. See *fcntl*(2). No process may have more than 64 files open simultaneously. A new file may be created with a mode that forbids writing.

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

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

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

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

The path name is null. [ENOENT]

The file does not exist and the directory in which the file is to be created does not permit writing. [EACCES]

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

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

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

The named file is an existing directory. [EISDIR]

Sixty-four (64) file descriptors are currently open. [EMFILE]

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

RETURN VALUE

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

SEE ALSO

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

NAME

dup, dup2 – duplicate an open file descriptor

SYNTAX

int dup (fildes)
int fildes;

int dup2 (fildes newfildes)
int fildes
int newfildes

DESCRIPTION

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

Dup and **dup2** return a new file descriptor having the following in common with the original:

Same open file

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

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

The new file descriptor is set to remain open across *exec* system calls. See fcntl (2).

Dup returns the lowest available file descriptor. **Dup2** closes *newfildes* and returns the value *newfildes,* which now refers to the same object as *fildes.*

Dup and **Dup2** fail if one or more of the following are true:

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

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

Dup2 fails if:

[EMFILE] *newfildes* is not in the range 0 to 63.

RETURN VALUE

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

SEE ALSO

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

NAME

execl, execv, execle, execve, execdve, execlp, execvp, execdvp – execute a file

SYNTAX

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

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

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

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

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

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

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

int execdvp (file, argv)
char *file, *argv[ ];

DESCRIPTION

Exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header (see *a.out*(4)), a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful *exec* because the calling process is overlaid by the new process.

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

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

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

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

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

Arg0, arg1, ..., argn are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

Envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the calling process's environment in the global cell:

extern char **environ;

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

Execdve is like *execve,* and *execdvp* is like *execvp,* except the "d" forms place the new or overlaid process into the interactive debugger (debug(1)) before execution begins.

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

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

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

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

process ID, parent process ID, process group ID, tty group ID (see exit (2) and signal (2)), current working directory, root directory, file mode creation mask (see umask (2)), *utime, stime, cutime,* and *cstime* (see *times*(2))

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

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

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

Search permission is denied for a directory listed in the new process file's path prefix. [EACCES]

The new process file is not an ordinary file. [EACCES]

The new process file mode denies execution permission. [EACCES]

The exec is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but an invalid magic number in its header. [ENOEXEC]

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

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

The number of bytes in either the new process's argument list or the new environment is greater than the system-imposed limit of >4092 bytes. [E2IG]

The new process file length is less than size values in its header. [EFAULT]

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

RETURN VALUE

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

SEE ALSO

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

NAME

exit, _exit – terminate process

SYNTAX

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

DESCRIPTION

Exit terminates the calling process with the following consequences:

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

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (i.e., bits 0377) of *status* are made available to it; see *wait*(2).

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by *times*.

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

An accounting record is written on the accounting file if the system's accounting routine is enabled; see *acct*(2).

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

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

SEE ALSO

intro(2), signal(2), wait(2).

WARNING

See *WARNING* in *signal*(2).

NAME

fcntl – file control

SYNTAX

#include <fcntl.h>

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

DESCRIPTION

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

The *cmds* available are:

F_DUPFD Return a new file descriptor as follows:

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

Same open file as the original file.

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

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

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

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

F_GETFD Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0** the file will remain open across *exec*, otherwise the file will be closed upon execution of *exec*.

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

F_GETFL Get *file* status flags.

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

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

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

Cmd is F_DUPFD and 64 file descriptors are currently open. [EMFILE]

Cmd is F_DUPFD and *arg* is negative or greater than 64. [EINVAL]

RETURN VALUE

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

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

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

SEE ALSO

close(2), dup(2), dup2(2), exec(2), open(2), fcntl(5).

NAME

fork – create a new process

SYNTAX

int fork ( )

DESCRIPTION

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

environment
close-on-exec flag (see *exec*(2))
signal handling settings (i.e., SIG_DFL, SIG_IGN, function address)
set-user-ID mode bit
set-group-ID mode bit
process group ID
tty group ID (see *exit*(2) and *signal*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))

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

The child process has a unique process ID.

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

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

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

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

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

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

RETURN VALUE

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

SEE ALSO

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

NAME

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

SYNTAX

int getpid ( )

int getppid ( )

DESCRIPTION

Getpid returns the process ID of the calling process.

Getppid returns the parent process ID of the calling process.

SEE ALSO

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

NAME

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

SYNTAX

int getuid ( )

int geteuid ( )

int getgid ( )

int getegid ( )

DESCRIPTION

Getuid returns the real user ID of the calling process.

Geteuid returns the effective user ID of the calling process.

Getgid returns the real group ID of the calling process.

Getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

NAME

ioctl – control device

SYNTAX

ioctl (fildes, request, arg)

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The writeups of various devices in Section 7 discuss how *ioctl* applies to them.

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

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

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

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

RETURN VALUE

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

SEE ALSO

termio(7).

NAME

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

SYNTAX

int kill (pid, sig)
int pid, sig;

DESCRIPTION

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

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

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

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

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

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

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

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

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

Sig is not a valid signal number. [EINVAL]

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

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

RETURN VALUE

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

SEE ALSO

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

NAME

link – link to a file

SYNTAX

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

DESCRIPTION

Path1 points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file. *Link* fails and no link is created if one or more of the following are true:

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

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

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

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

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

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

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

Path2 points to a null path name. [ENOENT]

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

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

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

RETURN VALUE

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

SEE ALSO

unlink(2).

NAME

lseek – move read/write file pointer

SYNTAX

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

DESCRIPTION

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

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

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

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

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

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

Fildes is not an open file descriptor. [EBADF]

Fildes is associated with a pipe or fifo.

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

The resulting file pointer would be negative. [EINVAL]

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

RETURN VALUE

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

SEE ALSO

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

NAME
: mkdir – make a directory file

SYNTAX

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

DESCRIPTION

Mkdir creates a new directory file with name *path*. The mode of the new file is initialized from *mode*. (The protection part of the mode is modified by the process's mode mask; see *umask*(2)).

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to that of the parent directory in which it is created.

The low-order 9 bits of mode are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared. See *umask*(2).

RETURN VALUE

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

ERRORS

Mkdir will fail and no directory will be created if:

[EPERM]	The process's effective user ID is not super-user.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EROFS]	The named file resides on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	*Path* points outside the process's allocated address space.
[EIO]	An I/O error occured while writing to the file system.

SEE ALSO

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

NAME

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

SYNTAX

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

DESCRIPTION

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

```
0170000 file type; one of the following:
        0010000 fifo special
        0020000 character special
        0040000 directory
        0060000 block special
        0100000 or 0000000 ordinary file
0004000 set user ID on execution
0002000 set group ID on execution
0000777 access permissions; constructed from the following
        00000400 read by owner
        0000200 write by owner
        0000100 execute (search on directory) by owner
        0000070 read, write, execute (search) by group
        0000007 read, write, execute (search) by others
```

The file's owner ID is set to the process's effective user ID. The file's group ID is set to the process's effective group ID.

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

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

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

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

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

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

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

The named file exists. [EEXIST]

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

RETURN VALUE

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

SEE ALSO

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

NAME

mount – mount a file system

SYNTAX

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

DESCRIPTION

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

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

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

Mount may be invoked only by the super-user.

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

> The effective user ID is not super-user. [EPERM]
>
> Any of the named files does not exist. [ENOENT]
>
> A component of a path prefix is not a directory. [ENOTDIR]
>
> *Spec* is not a block special device. [ENOTBLK]
>
> The device associated with *spec* does not exist. [ENXIO]
>
> *Dir* is not a directory. [ENOTDIR]
>
> *Spec* or *dir* points outside the process's allocated address space. [EFAULT]
>
> *Dir* is currently mounted on, is someone's current working directory or is otherwise busy. [EBUSY]
>
> The device associated with *spec* is currently mounted. [EBUSY]

RETURN VALUE

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

SEE ALSO

umount(2).

BUGS

Currently, a tape cannot be mounted, and physical write-protect cannot be set.

NAME

nice – change priority of a process

SYNTAX

int nice (incr)
int incr;

DESCRIPTION

Nice adds the *incr* value to the "nice number" of the calling process. A low "nice number" means high process priority. A high nice number means a lower priority.

Nice values range from 0 to 79, inclusive. A request to set the value beyond a limit results in the value being set to the corresponding limit.

Only the super-user can add a negative value and thereby increase priority. [EPERM]

RETURN VALUE

Upon successful completion, *nice* returns the new nice process priority. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

nice(1), exec(2).

NAME

open – open for reading or writing

SYNTAX

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

DESCRIPTION

Path points to a path name naming a file. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes. See *read*(2) and *write*(2).

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

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

If O_NDELAY is clear:

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

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

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

O_CREAT If the file exists, this flag has no effect. Otherwise, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

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

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

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

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

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

No process may have more than 64 file descriptors open simultaneously.

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

> A component of the path prefix is not a directory. [ENOTDIR]
>
> O_CREAT is not set and the named file does not exist. [ENOENT]
>
> A component of the path prefix denies search permission. [EACCES]
>
> *Oflag* permission is denied for the named file. [EACCES]
>
> The named file is a directory and *oflag* is write or read/write. [EISDIR]
>
> The named file resides on a read-only file system and *oflag* is write or read/write. [EROFS]
>
> Sixty-four (64) file descriptors are currently open. [EMFILE]
>
> The named file is a character special or block special file, and the device associated with this special file does not exist. [ENXIO]
>
> The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write. [ETXTBSY]
>
> *Path* points outside the process's allocated address space. [EFAULT]
>
> O_CREAT and O_EXCL are set, and the named file exists. [EEXIST]
>
> O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading. [ENXIO]

RETURN VALUE

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

SEE ALSO

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

NAME

pause – suspend process until signal

SYNTAX

pause ( )

DESCRIPTION

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

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

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

SEE ALSO

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

NAME

pipe – create an interprocess channel

SYNTAX

int pipe (fildes)
int fildes[2];

DESCRIPTION

Pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading, and *fildes*[1] is opened for writing.

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

No process may have more than 64 file descriptors open simultaneously.

Pipe will fail if 63 or more file descriptors are currently open [EMFILE].

RETURN VALUE

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

SEE ALSO

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

NAME

ptrace – process trace

SYNTAX

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

DESCRIPTION

Ptrace allows a parent process to control the execution of its child process. Its primary use is the implementation of breakpoint debugging.

The child process behaves normally until it encounters a signal documented in signal(2), at which time it enters a stopped state and its parent is notified via wait(2). While the child is in the stopped state, its parent can examine and modify it, and/or terminate or continue, by means of *ptrace*.

request determines the precise action of *ptrace*:

0 must be issued by the child process, to set its own trace flag, that indicates it should be left in a stopped state upon receipt of a signal rather than the state specified by *func*, so that it may be traced by its parent. see *signal(2)* for *func* info. The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

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

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

3 With this request, the word at location *addr* in the child's process control block in the system's address space (see **<sys/user.h>**) is returned to the parent process. ~~Addresses in this area range from 0 to 1024 on the PDP-11s and 0 to 2048 on the 3B20S and VAX~~. The *data* argument is ignored. This request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of – 1 is returned to the parent process and the parent's *errno* is set to EIO.

4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated, request 4 writes a word into I space, and request 5 writes a word into D space. If I and D space are not separated, either request 4 or request 5 may be used with equal results. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests will fail if *addr* is a location in a pure procedure space and another process is executing in that space, or *addr* is not the start address of a word. Upon failure a value of – 1 is returned to the parent process and the parent's *errno* is set to EIO.

6 With this request, a few entries in the child's process control block can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are the general registers, the program counter, and the trapsword.

7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the parent. This request will fail if *data* is not 0 or a valid signal number, in which case a value of − 1 is returned to the parent process and the parent's *errno* is set to EIO.

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

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.

GENERAL ERRORS

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

Request is an illegal number. [EIO]

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

SEE ALSO

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

NAME

read – read from file

SYNTAX

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

Buf points outside the allocated address space. [EFAULT]

RETURN VALUE

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

SEE ALSO

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

NAME

rename – change the name of a file

SYNTAX

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

DESCRIPTION

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

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

RETURN VALUE

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

ERRORS

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

[ENOTDIR]	A component of either path prefix is not a directory.
[ENOENT]	A component of either path prefix does not exist.
[EACCES]	A component of either path prefix denies search permission.
[ENOENT]	The file named by *from* does not exist.
[EPERM]	The file named by *from* is a directory and the effective user ID is not super-user.
[EXDEV]	The link named by *to* and the file named by *from* are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.
[EINVAL]	*From* is a parent directory of *to*.

SEE ALSO

open(2)

NAME

rmdir – remove a directory file

SYNTAX

rmdir(path)
char *path;

DESCRIPTION

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

RETURN VALUE

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

ERRORS

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

[ENOTEMPTY]	The named directory contains files other than "." and ".." in it.
[ENOENT]	The pathname was too long.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EROFS]	The directory entry to be removed resides on a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.

SEE ALSO

mkdir(2), unlink(2)

NAME

setuid, setgid – set user and group IDs

SYNTAX

int setuid (uid)
int uid;

int setgid (gid)
int gid;

DESCRIPTION

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

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

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

Setuid (*setgid*) will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user. [EPERM]

RETURN VALUE

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

SEE ALSO

getuid(2), setregid(2), setreuid(2), intro(2).

NAME

setregid – set real and effective group ID

SYNTAX

setregid(rgid, egid)
int rgid, egid;

DESCRIPTION

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

Supplying a value of – 1 for either the real or effective group ID forces the system to substitute the current ID in place of the – 1 parameter.

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.

ERRORS

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

SEE ALSO

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

NAME

setreuid – set real and effective user ID's

SYNTAX

setreuid(ruid, euid)
int ruid, euid;

DESCRIPTION

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

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.

ERRORS

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

SEE ALSO

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

NAME

signal – specify what to do upon receipt of a signal

SYNTAX

#include <sys/signal.h>

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

DESCRIPTION

Signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

Sig can be assigned any one of the following except SIGKILL:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03	quit
SIGILL	04	illegal instruction (not reset when caught)
SIGTRAP	05	trap 21 (trace trap) (not reset when caught)
SIGIOT	06	unused
SIGEMT	07	trap 3 (emulator trap)
SIGFPE	08	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18	child died
SIGPWROFF	19	power went off
SIGIO	59	i/o is possible on a descriptor (see the c_lflag **lintrup** in termio(7))
SIGRBUG	100	trap 0, enter system debugger (privilege req'd)
SIGDEBUG	101	trap 1, enter program debugger
SIGTRAP4	104	trap 4, reserved
SIGTRAP5	105	trap 5, reserved
SIGTRAP6	106	trap 6, reserved
SIGTRAP7	107	trap 7, reserved
SIGTRAP8	108	trap 8, reserved
SIGTRAP9	109	trap 9, reserved
SIGTRAP10	110	trap 10,reserved
SIGTRAP11	111	trap 11,reserved
SIGTRAP12	112	trap 12,reserved
SIGTRAP13	113	trap 13,reserved
SIGTRAP14	114	trap 14, user-defined trap1
SIGTRAP15	115	trap 15, user-defined trap2
SIGINTOVER	116	trap 16, integer overflow
SIGINTDIVZERO	117	trap 17, divide by zero
SIGREALOVER	118	trap 18, real overflow
SIGREALUNDER	119	trap 19, real underflow
SIGREALDIVZERO	120	trap 20, real div by 0

SIGTRAP21	121 trap 21, reserved
SIGTRAP22	122 trap 22, reserved
SIGTRAP23	123 trap 23, reserved
SIGTRAP24	124 trap 24, reserved
SIGTRAP25	125 trap 25, reserved
SIGTRAP26	126 trap 26, reserved
SIGTRAP27	127 trap 27, reserved
SIGTRAP28	128 trap 28, reserved
SIGTRAP29	129 trap 29, reserved
SIGTRAP30	130 trap 30, reserved
SIGPWRON	131 power went back on
SIGBOUNDS	132 array bounds exceeded

Func is assigned one of four values: SIG_DBG, SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values of are as follows:

SIG_DBG – suspend process and enter debugger (**debug_**).

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2).

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signal SIGKILL cannot be ignored.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Before entering the signal-catching function, the value of *func* for the caught signal will be set to its default value. The default signal for **SIGQUIT, SIGILL, SIGBUS,** and **SIGSEGV** is **SIG_DBG**. All other signals have a default setting of **SIG_DFL**.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

When a signal that is to be caught occurs during a *read*, a *write*, an *open*, or an *ioctl* system call on a slow device (like a terminal; but not a file), during a *pause* system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call will return a – 1 to the calling process with *errno* set to EINTR.

SIGFPE

SIGREALOVER (118), SIGREALUNDER (119), and SIGREALDIVZERO (120) work in conjunction with SIGFPE (08). When SIGFPE is set, 118, 119, and 120 are set to the same value. When a hardware floating-point exception occurs, one of 118, 119, or 120 is generated. A process never receives SIGFPE as a result of a floating-point exception.

A call to *signal* cancels a pending signal *sig* except for a pending SIGKILL signal.

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

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

Func points to an illegal address. [EFAULT]

RETURN VALUE

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

SEE ALSO

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

NAME

spawnl, spawnv, spawnle, spawnve, spawnlp, spawnvp – spawn a process

SYNTAX

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

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

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

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

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

int spawnvp (file, argv)
char *file, *argv[ ];

DESCRIPTION

spawn creates a new process. The new process is constructed from an ordinary, executable file called the *"new process file"* consisting of a header (see *a.out (4)*), a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss).

Spawn is the exact equivalent of a combination of *fork(2)* and *exec(2)*, but executes faster.

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

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

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

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

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

Arg0, arg1, ..., argn are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

Envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a null pointer. For *spawnl* and *spawnv*, the C run-time start-off routine places a pointer to the calling process's environment in the global cell:

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

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

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

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

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

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

process ID
parent process ID
process group ID
tty group ID (see *exit*(2) and *signal*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
utime, *stime*, *cutime*, and *cstime* (see *times*(2))

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

The system limit on the number of executing processes, or the limit on the number of executing processes for one user, is exceeded. [EAGAIN]

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

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

Search permission is denied for a directory listed in the new process file's path prefix. [EACCES]

The new process file is not an ordinary file. [EACCES]

The new process file mode denies execution permission. [EACCES]

The spawn is not an *spawnlp* or *spawnvp*, and the new process file has the appropriate access permission but an invalid magic number in its header. [ENOEXEC]

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

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

The number of bytes in either the new process's argument list or the new environment is greater than the system-imposed limit of >4092 bytes. [E2IG]

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

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

RETURN VALUE

If an error occurs, *spawn* returns − 1 to the calling process and the *errno* will be set to indicate the error. If successful, *spawn* returns the process ID of the new process, and *errno* is set to zero.

SEE ALSO

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

NAME

stat, fstat – get file status

SYNTAX

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

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

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

DESCRIPTION

Path points to a path name naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *Stat* obtains information about the named file.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, or *fcntl*. system call.

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

While the contents of the structure pointed to by *buf* include the following members, they are not necessarily in this order, nor are they necessarily the only entries:

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

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

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

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

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

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

The named file does not exist. [ENOENT]

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

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

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

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

Buf points to an invalid address. [EFAULT]

RETURN VALUE

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

SEE ALSO

chmod(2), chown(2), creat(2), link(2), time(2), unlink(2).

NAME

time – get time

SYNTAX

long time ((long *) 0)

long time (tloc)
long *tloc;

DESCRIPTION

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

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

Time will fail if *tloc* points to an illegal address. [EFAULT]

RETURN VALUE

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

NAME

times – get process and child process times

SYNTAX

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

long times (buffer)
struct tms *buffer;
```

DESCRIPTION

Times fills the structure pointed to by *buffer* with time-accounting information. The following is this contents of the structure:

```
struct  tms {
        time_t  tms_utime;
        time_t  tms_stime;
        time_t  tms_cutime;
        time_t  tms_cstime;
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. All times are milliseconds (thousandths of a second).

Tms_utime is the CPU time used while executing instructions in the user space of the calling process.

Tms_stime is the CPU time used by the associated User Monitor process on behalf of the calling process, not whole operating system time.

Tms_cutime is the sum of the *tms_utimes* and *tms_cutimes* of the child processes.

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

If *buffer* points to an illegal address, *times* fails and returns error 14 (EFAULT - bad address). [DEFAULT]

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in milliseconds, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a – 1 is returned and *errno* is set to indicate the error.

NAME

truncate, ftruncate – truncate a file to a specified length

SYNTAX

```
truncate(path, length)
char *path;
int length;

ftruncate(fd, length)
int fd, length;
```

DESCRIPTION

Truncate causes the file named by *path* or referenced by *fd* to be truncated to at most *length* bytes in size. If the file previously was larger than this size, the extra data is lost. With *ftruncate*, the file must be open for writing.

RETURN VALUES

A value of 0 is returned if the call succeeds. If the call fails a – 1 is returned, and the global variable *errno* specifies the error.

ERRORS

Truncate succeeds unless:

[EPERM]	The pathname contains a character with the high-order bit set.
[ENOENT]	The pathname was too long.
[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	A component of the *path* prefix denies search permission.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EFAULT]	*Name* points outside the process's allocated address space.

Ftruncate succeeds unless:

[EBADF]	The *fd* is not a valid descriptor.
[EINVAL]	The *fd* references a socket, not a file.

SEE ALSO

open(2)

BUGS

Partial blocks discarded as the result of truncation are not zero filled; this can result in holes in files which do not read as zero.

These calls should be generalized to allow ranges of bytes in a file to be discarded.

NAME

umask – set and get file creation mask

SYNTAX

int umask (cmask)
int cmask;

DESCRIPTION

Umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

RETURN VALUE

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

SEE ALSO

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

NAME

umount – unmount a file system

SYNTAX

int umount (spec)
char *spec;

DESCRIPTION

Umount requests that a previously mounted file system contained on the block special device identified by *spec* be unmounted. *Spec* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

Umount may be invoked only by the super-user.

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

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

Spec does not exist. [ENXIO]

Spec is not a block special device. [ENOTBLK]

Spec is not mounted. [EINVAL]

A file on *spec* is busy. [EBUSY]

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

RETURN VALUE

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

SEE ALSO

mount(2).

NAME

unlink – remove directory entry

SYNTAX

int unlink (path)
char *path;

DESCRIPTION

Unlink removes the directory entry named by the path name pointed to by *path*. An error can prevent removal:

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

The named file does not exist. [ENOENT]

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

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

The named file is a directory and the effective user ID of the process is not super-user. [EPERM]

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

The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed. [ETXTBSY]

The directory entry to be unlinked is part of a read-only file system. [EROFS]

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

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

RETURN VALUE

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

SEE ALSO

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

NAME

utime – set file access and modification times

SYNTAX

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

DESCRIPTION

Path points to a path name naming a file. *Utime* sets the access and modification times of the named file.

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

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use *utime* this way.

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

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

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

The named file does not exist. [ENOENT]

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

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

The effective user ID is not super-user and not the owner of the file and *times* is not NULL. [EPERM]

The effective user ID is not super-user and not the owner of the file and *times* is NULL and write access is denied. [EACCES]

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

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

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

RETURN VALUE

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

SEE ALSO

stat(2).

NAME

wait – wait for child process to stop or terminate

SYNTAX

int wait (stat_loc) int *stat_loc;
int wait ((int *)0)

DESCRIPTION

Wait suspends the calling process until it receives a signal that is to be caught (see *signal*(2)), or until one of the calling process's child processes stops in a trace mode (see *ptrace*(2)), or terminates. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) is non-zero, 16 bits of information called "status" are stored in the low-order bits of the location pointed to by *stat_loc*. *Status* can be used to differentiate between stopped and terminated child processes and, if the child process terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished as follows:

> If the child process stopped, the high-order eight bits of status will contain the number of the signal that caused the process to stop and the low-order eight bits of the argument that the child process passed to *exit* (see *exit*(2))
>
> If the child process terminated due to an *exit* call, the low-order eight bits of status will be zero and the high-order eight bits will contain the low-order eight bits of the argument that the child process passed to *exit* (see *exit*(2)).
>
> If the child process terminted due to a signal, the high-order eight bits of status will be zero and the low-order eight bits will contain the number of the signal that caused the termination. In addition, if the low-order seventh bit (i.e., bit 200) is set, a "core image" will have been produced (see *signal*(2)).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes (see *intro*(2)).

Wait will fail and return immediately if either or both of the following are true:

> The calling process has no existing unwaited-for child processes. [ECHILD]
>
> *Stat_loc* points to an illegal address. [EFAULT]

RETURN VALUE

If *wait* returns due to the receipt of a signal, a value of -1 is returned to the calling process and errno ~~*erno*~~ 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.

SEE ALSO

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

WARNING

See *WARNING* in *signal*(2).

NAME

write – write on a file

SYNTAX

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

DESCRIPTION

Fildes is a file descriptor obtained from a *creat, open, dup,* or *fcntl* system call. *Write* attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written. On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write. *Write* fails and the file pointer remains unchanged if any of the following are true:

> *Fildes* is not a valid file descriptor open for writing.
>
> An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. [EFBIG]
>
> *Buf* points outside the process's allocated address space. [EFAULT]

If a *write* requests that more bytes be written than there is room for only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

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

NAME

intro – introduction to subroutines and libraries

SYNTAX

#include <stdio.h>

#include <math.h>

DESCRIPTION

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

(3A)	C-callable Subroutines
(3B)	Pascal-callable Subroutines
(3C)	These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the **– lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.
(3F)	FORTRAN function library automatically accessed by F77 compiler.
(3M)	These functions constitute the Math Library, *libm*. They are automatically loaded as needed by the FORTRAN compiler. They are not automatically loaded by the C compiler, *cc*(1); however, the link editor searches this library under the **– lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**.
(3S)	These functions constitute the "standard I/O package" (see *stdio*(3S)). These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.
(3X)	Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

DEFINITIONS

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

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

FILES

/lib/libc.a
/lib/libm.a

SEE ALSO

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

DIAGNOSTICS

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

NAME

AbortCommand – abort a command process

SYNTAX

FUNCTION AbortCommand (pID : ProcessID) : Error;

DESCRIPTION

AbortCommand is used to kill a command process. Any files left open by the command process will be closed, and other system resources will be freed.

The command process **pID** must have been created by the LoadCommand routine, and can be either active or suspended. A user process that calls AbortCommand must be managed by the same User Monitor that manages the process **pID**; in other words, it is not possible with AbortCommand to kill processes that belong to another user.

An exit code is returned to the invoking process that started the command process via the StartCommand routine. The exit code value is currently undefined when a process is killed by AbortCommand.

SEE ALSO

LoadCommand (3B), StartCommand (3B)

NOTES

ErBadPID is returned if **pID** is not a valid command process ID.

NAME

Access – check accesibility of a file.

SYNTAX

```
FUNCTION Access (name : String;
                 mode : Integer): Error;
```

DESCRIPTION

Access checks the file or directory **name** for accessibility according to the **mode**, which is any additive combination of the values 4, 2, and 1, for read, write, and execute access, respectively. **Mode** value 0 checks whether the directories leading to the file can be searched, and that the file exists.

SEE ALSO

ChangeMode (3B)

NOTES

ErBadFileName is returned if the name is too long or contains invalid characters.

ErFileNotFound is returned if the file or directory cannot be found.

ErAccess is returned if any of the desired access modes would not be granted.

NAME

ChangeDir – change current working directory

SYNTAX

FUNCTION ChangeDir (name : String) : Error;

DESCRIPTION

ChangeDir changes the current working directory to the pathname **name**. The current working directory is the default prefix for pathnames not beginning with /.

SEE ALSO

GetCurrentDir (3B)

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters, while ErNotDirectory is returned if the name is not a directory.

NAME

ChangeFileSize – change the size of a file

SYNTAX

```
FUNCTION ChangeFileSize (handle : Link;
                         desiredSize : Integer) : Error;
```

DESCRIPTION

ChangeFileSize extends or truncates the size of the open file specified by **handle** to the number of bytes specified by **desiredSize**. The amount of secondary storage space allocated to the file is changed if necessary.

The file must have been opened for writing.

SEE ALSO

Create (3B), Open (3B)

NOTES

ErBadLink is returned if **handle** is not a valid open file, while ErNotWritable indicates that the file cannot be written.

NAME

ChangeMode – change access mode of a file

SYNTAX

```
FUNCTION ChangeMode (name : String;
                     accessMode: Halfword) : Error;
```

DESCRIPTION

ChangeMode changes the access mode (permission bits) of the file or directory **name** to the value **accessMode**. The bits are set to 1 to grant the access; to 0 to deny the access. The access mode can assume combinations of the following:

100000000000	set user ID on execution
010000000000	set group ID on execution
001000000000	(not used)
000100000000	read permission for owner
000010000000	write permission for owner
000001000000	execute (or search directory) permission for owner
000000111000	read, write, and execute permission for group
000000000111	read, write, and execute permission for others

Only the owner of a file or directory, or the Super-User, can change its access mode.

SEE ALSO

Access (3B)

NOTES

ErBadFileName is returned of the name is too long, or contains invalid characters.

ErFileNotFound is returned if the file or directory cannot be found.

ErNotOwner is returned if the caller is neither the owner not the Super-User.

NAME

Close – close a file

SYNTAX

FUNCTION Close (handle : Link) : Error;

DESCRIPTION

Given a link **handle** to a file as returned by a Create or Open call, Close will close the associated file. Programs which use large numbers of files should use Close when no more access to a file is required, since there is a limit on the number of open files per process. All open files for a process are closed automatically when the process terminates.

Unused secondary storage space may be deallocated, although Close does not change the logical size of a file.

SEE ALSO

Create (3B), Open (3B)

NOTES

ErBadLink is returned if **handle** is not a valid open file.

NAME

CloseFile – close a stream file

SYNTAX

PROCEDURE CloseFile (var f : Text);

DESCRIPTION

CloseFile closes the stream file associated with **f**, which must have been opened by the OpenFile routine. Closing a stream file causes any buffers to be flushed if necessary, and the file becomes inactive.

All open stream files are closed automatically when a program terminates normally, or when the SysExit routine is called.

SEE ALSO

Close (3B), OpenFile (3B), SysExit (3B)

NAME

ConcatString – concatenate two strings

SYNTAX

```
FUNCTION ConcatString (s1 : String;
                       s2 : String) : String;
```

DESCRIPTION

ConcatString creates a new string that is the concatenation of the strings **s1** and **s2**. The new string is returned, and neither **s1** nor **s2** are modified.

SEE ALSO

NewString (3B)

NAME

CopyOfString – make a copy of a string

SYNTAX

FUNCTION CopyOfString (s : String) : String;

DESCRIPTION

CopyOfString creates a new string that is a copy of string **s**. The new string is returned, with the same length and same contents as **s**. String **s** is not modified.

SEE ALSO

NewString (3B), OverlayString (3B)

NAME

CopySubString – copy a substring into another string

SYNTAX

```
PROCEDURE CopySubString (dest : String;
                         dFirst : Integer;
                         dLast : Integer;
                         source : String;
                         sFirst : Integer);
```

DESCRIPTION

CopySubString copies the substring from **source** to the substring in **dest**. The characters starting at position **sFirst** in the string **source** are copied to the characters in positions **dFirst** to **dLast** in the string **dest**.

SEE ALSO

NewString (3b), SubString (3b)

NOTES

No bounds check is made to insure that either of the substrings are completely contained within the strings. No check is made for overlapping substrings when the source and destination are the same string.

NAME

Create – create a file

SYNTAX

```
FUNCTION Create (name : String;
            accessMode : Halfword;
            allocSize : Integer;
        var handle : Link;
        var flag : Integer) : Error;
```

DESCRIPTION

Create tries to create a new file, or prepares to rewrite an existing file. The string **name** represents the pathname of the file.

If the file did not exist, it is given access mode **accessMode**, and a file size of zero. If the file did exist, its mode and owner remain unchanged, but the file size is truncated to zero length. In either case, the file will be allocated sufficient storage space to hold **allocSize** bytes.

The file is then opened for both reading and writing, and **handle** will contain a link which is used for subsequent input/output operations on the file.

A value is returned in **flag** that indicates whether the file should be accessed using block mode only (0), character mode only (1), or either mode (2).

SEE ALSO

Close (3B), Delete (3B), Open (3B)

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

If the file could not be created because of insufficient secondary storage space, ErVolumeFull is returned.

ErAccess is returned if either the file exists and its access mode is not writable, or the file does not exist but the directory in which it is to be created is not writable.

ErCantModifyDir is returned if a directory with the pathname **name** already exists.

NAME

CreateEquate – create a name equation

SYNTAX

```
FUNCTION CreateEquate (pID : ProcessID;
                       mapFrom : String;
                       mapTo : String) : Error;
```

DESCRIPTION

CreateEquate creates a name equation for the alias **mapFrom** to the string **mapTo**. The name equation is added to the list of name equations maintained by the User Monitor for each command process. When a name is specified in a request to the User Monitor, the name equation list is searched and if the name matches **mapFrom**, the string **mapTo** is substituted for the name. The most recent equation for **mapFrom** is used in the case of multiple equations for the same string.

The parameter **pID** specifies to which command process the name equation applies. If the value of **pID** is -1, then the name equation applies to all of the command processes managed by a single User Monitor.

When a command process terminates, its list of name equations is automatically deleted. The list associated with all command processes managed by a single User Monitor can only be shortened by DeleteEquate.

SEE ALSO

DeleteEquate (3B)

NOTES

ErBadFileName is returned if the strings are too long, while ErBadPID is returned if **pID** is not a valid command process ID.

NAME

CreateSpecial – create a directory or special file

SYNTAX

```
FUNCTION CreateSpecial (name : String;
                        accessMode : Halfword;
                        deviceFlag : Integer) : Error;
```

DESCRIPTION

CreateSpecial creates a directory whose pathname is given by **name**. The directory is given the access mode **accessMode**.

The parameter **deviceFlag** must be zero to create a directory. A nonzero value is used to create a device driver, which is not implemented. This interface is subject to change.

SEE ALSO

Create (3B), Delete (3B)

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErAccess is returned if the directory intended to contain **name** is not writable.

NAME

DecodeTime – convert a timestamp to a date and time

SYNTAX

```
PROCEDURE DecodeTime (time : TimeStamp;
                      var year : Integer;
                      var month : Integer;
                      var day : Integer;
                      var hour : Integer;
                      var minute : Integer;
                      var second : Integer;
                      var millisecond : Integer;
                      var nanosecond : Integer);
```

DESCRIPTION

DecodeTime converts a timestamp to a set of numbers that represent a date and time. The timestamp **time** is an encoding of the number of nanoseconds since the beginning of the year 1970. DecodeTime converts the timestamp as described below, handling leap years properly.

The **year** will contain a value greater than or equal to 1900, the **month** will contain a value from 1 to 12, and the **day** will contain a value from 1 to 31. The **hour** will contain a value from 0 to 23, and both the **minute** and the **second** will contain a value from 0 to 59. The **millisecond** will contain a value from 0 to 999, and the **nanosecond** will contain a value from 0 to 999999.

SEE ALSO

EncodeTime (3B)

NAME

Delete – delete a file

SYNTAX

FUNCTION Delete (name : String) : Error;

DESCRIPTION3

Delete removes **name** from the file system. The contents of a file are destroyed, and the secondary storage space is freed.

A directory may be removed only if it is empty.

SEE ALSO

Create (3B)

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErFileNotFound is returned if the name cannot be found.

ErAccess is returned if the name is not writable.

ErDirNotEmpty is returned if **name** is a directory, and it contains one or more entries.

NAME

DeleteEquate – delete a name equation

SYNTAX

```
FUNCTION DeleteEquate (pID : ProcessID;
                       mapFrom : String) : Error;
```

DESCRIPTION

DeleteEquate removes the name equation for the alias **mapFrom** from the list of name equations maintained by the User Monitor for the command process **pID**. If **pID** is the value -1, then the name equation is removed from the list of name equations that applies to all command processes managed by a single User Monitor. If multiple name equations exist for **mapFrom**, then only the most recent mapping will be deleted.

When a command process terminates, its list of name equations is automatically deleted. DeleteEquate is used to either delete a name equation before using CreateEquate to change a mapping, or to delete a name equation from the list for all command processes.

SEE ALSO

CreateEquate (3B)

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErBadPID is returned if **pID** is not a valid command process ID.

ErNoEquate is returned if **mapFrom** cannot be found.

NAME

Dispose – deallocate a string

SYNTAX

PROCEDURE Dispose (s : String);

DESCRIPTION

Dispose deallocates the data structure associated with string **s**, and makes its space on the heap available for reuse. The heap storage management allows strings to be allocated and deallocated in any order.

SEE ALSO

NewString (3B)

NOTES

Unpredictable results will occur if **s** had never been allocated, or if it had already been disposed.

NAME

EncodeTime – convert a date and time to a timestamp

SYNTAX

```
PROCEDURE EncodeTime (year : Integer;
                       month : Integer;
                       day : Integer;
                       hour : Integer;
                       minute : Integer;
                       second : Integer;
                       millisecond : Integer;
                       nanosecond : Integer;
                    var time : TimeStamp);
```

DESCRIPTION

EncodeTime converts a set of numbers that represent a date and time to a timestamp. The timestamp **time** is an encoding of the number of nanoseconds since the beginning of the year 1900. EncodeTime converts the set of numbers as described below, handling leap years properly.

The **year** should contain a value greater than or equal to 1900, the **month** should contain a value from 1 to 12, and the **day** should contain a value from 1 to 31. The **hour** should contain a value from 0 to 23, and both the **minute** and the **second** should contain a value from 0 to 59. The **millisecond** should contain a value from 0 to 999, and the **nanosecond** should contain a value from 0 to 999999.

SEE ALSO

DecodeTime (3B)

NOTES

An input parameter that is out of range results in a timestamp that is equal to zero.

NAME

EqualString
– compare two strings for equality

SYNTAX

```
FUNCTION EqualString (s1 : String;
                      s2 : String) : Boolean;
```

DESCRIPTION

EqualString compares the two strings **s1** and **s2** for equality. The value True is returned if both strings have exactly the same length and contents, otherwise the value False is returned. The contents are compared character by character, using the underlying character code values, thus upper and lower case characters are not identical.

SEE ALSO

NewString (3B)

NAME

FileStatus – check the status of a stream file

SYNTAX

FUNCTION FileStatus (var f : Text) : Error;

DESCRIPTION

FileStatus checks the status of the open stream file associated with **f**. The value returned is zero if no errors have occurred during any operations on the stream file.

The status may be tested immediately after an OpenFile call to check that the file was correctly opened, or it may be checked after an input/output operation to insure that the transfer was successfully completed.

SEE ALSO

OpenFile (3B)

NOTES

ErNotOpen is returned if the stream file is not open.

ErEOF is returned if there was an attempt to read past end of file.

ErFileStatus is returned if if any other error has occurred.

NAME

FillString – fill a string with a character

SYNTAX

```
PROCEDURE FillString (s : String;
                      first : Integer;
                      last : Integer;
                      ch : Char);
```

DESCRIPTION

FillString fills a substring within string **s** with the character **ch**. The characters from the position **first** to the position **last** are all given the value of **ch**.

SEE ALSO

NewString (3B)

NOTES

No bounds check is made to insure that the substring is completely contained within the string.

NAME

GetArgs – get command arguments

SYNTAX

FUNCTION GetArgs (var argc : Integer) : PStringVector;

DESCRIPTION

GetArgs is used by a command process to retrieve its command arguments from the User Monitor. The command arguments are typically file names to be operated on, or options to control the execution of the program. The invoking process, usually the Shell, accumulates the command arguments and passes them to the User Monitor via StartCommand when a command process is started.

GetArgs sets the argument count into **argc** and returns a pointer to an array of strings, as described by the following Pascal type definitions:

```
PStringVector = ^ StringVector;
StringVector  = Array [0..0] of String;
```

The array is indexed from zero, and actually contains **argc**+ 1 elements, where the last element is the value Nil.

By convention, the string at position zero is the name of the command process, and the other strings are the command arguments in sequence. Thus **argc** is always at least one.

SEE ALSO

StartCommand (3B)

NOTES

The strings returned by GetArgs should not be deallocated via Dispose since they are not allocated by NewString.

NAME

GetCurrentDir – get the name of the current working directory

SYNTAX

FUNCTION GetCurrentDir : String;

DESCRIPTION

GetCurrentDir returns a string which is the pathname of the current working directory.

SEE ALSO

ChangeDir (3B)

NAME

LoadCodeAndData – create a command process with existing data segment

SYNTAX

```
FUNCTION LoadCodeAndData (codeName : String;
                          dataName : String;
                          var pID : ProcessID) : Error;
```

DESCRIPTION

LoadCodeAndData creates a command process, using the datafile **codeName** as the executable code segment. The code file is found using the standard search order. If the name starts with /, then that exact pathname is used. Otherwise, a pathname is constructed by appending **codeName** first to the current working directory, then the directory **/bin**, and finally to the directory **/usr/bin**. The code file must have execute permission.

The existing file **dataName** is used as the data segment for the command process. If the name starts with /, then that exact pathname is used. Otherwise, the file is looked for in the current directory. The data file must have both read and write permission. The same file can be used for both the code and the data segment.

If both files are found, a queue segment is allocated, and an inactive process is created. A process ID is returned in **pID**, which is used for further management of the command process.

SEE ALSO

AbortCommand (3B), LoadCommand (3B), StartCommand (3B)

NOTES

ErBadFIleName is returned if a name is too long, or contains invalid characters.

ErFileNotFound is returned if a file cannot be found, while ErBadFileType is returned if it is not a regular file.

NAME

LoadCommand – create a command process

SYNTAX

```
FUNCTION LoadCommand (name : String;
                 var pID : ProcessID) : Error;
```

DESCRIPTION

LoadCommand creates a command process, using the file **name** as the executable code segment. The code file is found using the standard search order. If the name starts with /, then that exact pathname is used. Otherwise, a pathname is constructed by appending **name** first to the current working directory, then the directory **/bin**, and finally the directory **/usr/bin**. The code file must have execute permission.

If a code file is found, a data and a queue segment are allocated, and an inactive process is created. A process ID is returned in **pID**, which is used for further management of the command process.

SEE ALSO

AbortCommand (3B), LoadCodeAndData (3B), StartCommand (3B)

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErFileNotFound is returned if the file can not be found, while ErBadFileType is returned if it is not a regular file.

NAME

LookupName – lookup a file name

SYNTAX

```
FUNCTION LookupName (name : String;
               var fType : Integer;
               var fID : FileID) : Error;
```

DESCRIPTION

LookupName takes a pathname **name** and determines its mapping in the file system. If the name is a regular file, then **fType** will contain the value 1, and **fID** will contain the internal file identifier.

If the name is a directory, then **fType** will contain the value 0. The contents of **fID** for a directory are interpreted as four 16-bit values in sequence according to the following Pascal type definitions:

```
ownerID   = Halfword;
groupID   = Halfword;
protect   = Halfword;
linkCount = Halfword;
```

The **ownerID** and **groupID** fields indicate the owner of the directory. The **protect** field is the protection bits, or access mode, for the directory. The **linkCount** field represents the number of aliases, or links, to the directory from other directories in the file system.

SEE ALSO

ReadDirectory (3B), ReadLabel (3B)

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErFileNotFound is returned if the file or directory can not be found.

NAME

NewString – create a new string

SYNTAX

FUNCTION NewString (length : Integer) : String;

DESCRIPTION

NewString creates a new string, with enough space to hold **length** characters. A data structure is allocated on the heap, with the length field filled in, and undefined values for the sequence of characters. A pointer to this structure is returned by NewString, and the calling program can fill in the desired characters.

When the string is no longer required, its space should be deallocated by the Dispose routine.

SEE ALSO

Dispose (3B)

NAME

Open – open a file

SYNTAX

```
FUNCTION Open (name : String;
               mode : Integer;
           var handle : Link;
           var flag : Integer;
           var fileSize : Integer) : Error;
```

DESCRIPTION

Open tries to open an existing file, where the string **name** represents the pathname of the file. The file will be opened for reading (**mode** is 0), writing (**mode** is 1), or both reading and writing (**mode** is 2).

If the file is opened successfully, **handle** will contain a link which is used for subsequent input/output operations on the file. The file will be positioned at its beginning.

A value is returned in **flag** that indicates whether the file should be accessed using block mode only (0), character mode only (1), or either mode (2).

The value returned in **fileSize** is the number of bytes in the file. The size of a file that represents a device driver may not be determinable when the file is first accessed, and will be zero in this case.

SEE ALSO

Close (3B), Create (3B)

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErOpenMode is returned if **mode** is not 0, 1, or 2.

ErFileNotFound is returned if the file can not be found.

NAME

OpenFile – open a stream file

SYNTAX

```
PROCEDURE OpenFile (var f : Text;
                    name : String;
                    mode : Char);
```

DESCRIPTION

OpenFile is used to associate a stream file with the variable **f**, which can then be used in standard Pascal input/output operations or can be a parameter to other stream file routines. The string **name** specifies the pathname of the file. The character **mode** can be one of four different letters (upper or lower case) that indicate how the file should be opened, as described below.

The value **r** indicates the file should be opened for reading only, and therefore must exist.

The value **w** indicates the file should be opened for writing only, and is either created if it did not exist, or truncated to zero length if it did exist.

The value **a** indicates the file should be appended to, which is similar to writing only. The file is created if it did not exist. If the file exists, it is not truncated and the read/write cursor is positioned at the end of file.

The value **u** indicates that the file should be opened for update, which permits both reading and writing. The file is created if it did not exist, and the read/write cursor is positioned at the beginning of the file.

SEE ALSO

CloseFile (3B), Create (3B), FileStatus (3B), Open (3B)

NOTES

The FileStatus routine should be used to determine if a stream file was opened successfully. A stream file may not be opened correctly if the file name is not valid, does not exist or cannot be accessed according to the mode, or if too many open files already exist.

NAME

OverlayString – copy one string onto another

SYNTAX

```
PROCEDURE OverlayString (dest : String;
                         source : String);
```

DESCRIPTION

OverlayString copies the characters of the **source** string onto the previous contents of the **dest** string, and sets the length of **dest** to that of **source**. The **source** string is not modified.

SEE ALSO

CopyOfString (3B), NewString (3B)

NOTES

Unpredictable results may occur if the destination string was not at least as long as the source string.

NAME

PositionFile – move the read/write cursor of a stream file

SYNTAX

```
FUNCTION PositionFile (var f : Text;
                       offset : Integer;
                       origin : Integer ) : Integer;
```

DESCRIPTION

PositionFile moves the read/write cursor of the stream file associated with **f**. The next input or output operation on the stream file will occur at the new position, where position 0 is the first byte of the file.

The new position becomes the byte position determined from the signed value in **offset** and the value of **origin**. If **origin** is 0, the offset is from the beginning of the file; if **origin** is 1, the offset is relative to the current position; if **origin** is 2, the offset is relative to the end of the file. The return value of *PositionFile* is the resulting byte position in the file.

A successful *PositionFile* always clears the end-of-file status.

SEE ALSO

OpenFile(3B), SetFileSize(3B)

NOTES

An error is indicated by return value -1. An error may occur if **origin** is not 0, 1, or 2, if the stream file is not open or does not allow block mode or random access, or if the postion would be beyond the end of the file.

NAME

ReadBlock – read a block of a file

SYNTAX

```
FUNCTION ReadBlock (handle : Link;
                    bufAddr : PageAddress;
                    fileCursor : Integer;
                    length : Integer;
                var actual : Integer) : Error;
```

DESCRIPTION

ReadBlock is used to read a block of data from the file specified by **handle**. The **handle** is a file link returned from a successful Create or Open call, and it represents a file that must allow block mode access.

The block of data from the file at the byte position specified by **fileCursor** is transferred to the buffer address in the user process data segment specified by **bufAddr**, which must be page-aligned.

The amount of data to be read is specified in **length**, and the amount actually transferred is returned in **actual**, which can be from 0 to 4096 bytes.

SEE ALSO

Create (3B), Open (3B), ReadChar (3B), WriteBlock (3B), WriteChar (3B)

NOTES

ErBadLink is returned if **handle** is not a valid open file.

ErNotAligned is returned if **bufAddr** is not page-aligned, that is, an address that is not a multiple of 4096, while ErBadBlockLength is returned if **length** is greater than 4096.

ErNotReadable is returned if the file was not opened to allow reading.

ErEOF is returned if an attempt is made to read a block past the end of file.

NAME

ReadChar – read a character

SYNTAX

```
FUNCTION ReadChar (handle : Link;
          var ch : Char) : Error;
```

DESCRIPTION

ReadChar reads a single character from the file specified by **handle**. The **handle** is a file link returned from a successful Create or Open call, and it represents a file that must allow character mode access.

The character is returned in **ch**, and can be any 8-bit value.

SEE ALSO

Create (3B), Open (3B), ReadBlock (3B), WriteBlock (3B), WriteChar (3B)

NOTES

ErBadLink is returned if **handle** is not a valid open file.

ErNotReadable is returned if the file was not opened to allow reading.

ErEOF is returned if an attempt is made to read a character past the end of file.

NAME

ReadDirectory - read the contents of a directory

SYNTAX

```
FUNCTION ReadDirectory (name : String;
                        firstRequest : Boolean;
                        dirPage : PDirectoryPage;
                      var numEntries : Integer;
                      var anyMore : Boolean) : Error;
```

DESCRIPTION

ReadDirectory is used to read the contents of a directory. The directory contents are returned in a standard format, which contains entries that map a name to an internal file identifier, or indicate that an entry is the name of a subdirectory.

Each call to ReadDirectory returns only one 4096-byte page of information. The entries are returned in alphabetical order so that multiple requests can be made, each time specifying a different place in the alphabetical list to start returning more entries. The parameters "firstRequest" and "anyMore" are used as described below to make multiple requests, thus enabling a large directory to be read.

The first call to ReadDirectory for a directory whose pathname is "name" should have the parameter "firstRequest" set to True. The number of valid entries for the returned directory page is returned in "numEntries". If more information exists in the directory than can be returned in a single response, then the parameter "anyMore" will be set to True upon return, otherwise "anyMore" will be set to False.

If "anyMore" is True, then another call to ReadDirectory should be made with "firstRequest" set to False. The parameter "name" should be extended to include the directory name and the name of the last entry returned, separated by a "/". The returned information will start with the next alphabetical entry.

ReadDirectory places a block of directory information into the data area specified by "dirPage" in the following format:

```
PDirectoryPage = ^ DirectoryPage;
DirectoryPage  = Array [0..127] of DirectoryEntry;
DirectoryEntry = Record
                   name  : Array [1..16] of Char;
                   fType : Integer;
                   fID   : FileID;
                 end;
```

The page may contain up to 128 entries, starting with entry number zero. Each entry has a "name" field, which is 1 to 16 characters with blanks filled at the end. The "fType" field has the value 1 if the entry is a regular file, and the "fID" field contains an internal file identifier in this case.

The "fType" field has the value 0 if the entry is a directory. The contents of the "fID" field for a directory are interpreted as four 16-bit values in sequence according to the following Pascal type definitions:

```
ownerID   = Halfword;
groupID   = Halfword;
protect   = Halfword;
linkCount = Halfword;
```

The "ownerID" and "groupID" fields indicate the owner of the directory. The "protect" field is the protection bits, or access mode, for the directory. The "linkCount" field represents the number of aliases, or links, to the directory from other directories in the file system.

SEE ALSO

CreateSpecial (3B), LookupName (3B), ReadLabel (3B) ErBadFileName is returned if the name is too long, or contains invalid characters, while ErNotDirectory is returned if the name is not a directory.

NAME

ReadLabel – read a file label

SYNTAX

```
FUNCTION ReadLabel (fID : FileID;
                    lab : PFileLabel) : Error;
```

DESCRIPTION

ReadLabel is used to read the contents of the file label for the file specified by the internal file identifier **fID**. The file label contains information about the file that is maintained by the file system.

ReadLabel places a block of file label information into the data area specified by **lab** in the following format:

```
PFileLabel = ^ FileLabel;
FileLabel  = Record
          internalCreate : TimeStamp;
          createTime     : TimeStamp;
          refTime        : TimeStamp;
          modTime        : TimeStamp;
          ownerID        : Halfword;
          groupID        : Halfword;
          protect        : Halfword;
          linkCount      : Halfword;
          fileSize       : Integer;
          uType          : Integer;
        end;
```

The **internalCreate** field contains the time that the file was actually created. The **createTime** field contains the time that the file was logically created, which may be different than the **internalCreate** time if the file is a copy of another file, for instance. The **refTime** field contains the last time that the file was referenced, that is, closed after being opened for reading or writing. The **modTime** field contains the last time that the file was modified, that is, closed after being opened for writing.

The **ownerID** and **groupID** fields indicate the owner of the directory. The **protect** field is the protection bits, or access mode, for the file. The **linkCount** field represents the number of name mappings, or links, to the file from directories in the file system.

The **fileSize** field maintains the size of the file in bytes. The **uType** field contains a file type value maintained by the system.

SEE ALSO

LookupName (3B), ReadDirectory (3B)

NOTES

ErBadFileID is returned if **fID** is not a valid internal file identifier.

NAME

SearchString – search for a character in a string

SYNTAX

```
FUNCTION SearchString (s : String;
                       ch : Char) : Integer;
```

DESCRIPTION

SearchString searches the string **s** for the first occurrence of the character value **ch**. If **ch** is found, then its position in the sequence of characters is returned. Zero is returned if **ch** can not be found.

SEE ALSO

NewString (3B)

NAME

SetFileSize – change the size of a stream file

SYNTAX

```
FUNCTION SetFileSize (var f : Text;
                      desiredSize : Integer) : Error;
```

DESCRIPTION

SetFileSize extends or truncates the size of the stream file associated with **f** to the number of bytes specified by **desiredSize**. The amount of secondary storage space allocated to the file is changed if necessary.

The stream file must have been opened with a mode that allows writing.

If the stream file is truncated in front of the current read/write cursor, then the read/write cursor is positioned to the new end of file.

SEE ALSO

OpenFile (3B), ChangeFileSize (3B), PositionFile (3B)

NOTES

ErNotWritable is returned if the file is not writable.

NAME

SetDataBounds – set the maximum stack and heap sizes.

SYNTAX

```
PROCEDURE SetDataBounds (desiredStackMax : Integer;
                         desiredHeapMax : Integer;
                     var resultingStackMax : Integer;
                     var resultingHeapMax : Integer);
```

DESCRIPTION

SetDataBounds sets the maximum sizes of the stack and heap areas within the data segment associated with the calling command process.

When a command process is created via the LoadCommand routine, its data segment is created with a certain amount of secondary storage allocated to the stack and the heap areas. The stack area grows upwards from location zero (increasing addresses), while the heap area grows downwards from the end of the address space (decreasing addresses).

The default maximum size of both the stack and heap areas is 32M bytes. Data segment references within the stack or heap area bounds will automatically be allocated secondary storage to satisfy the references as needed. Data segment references outside the stack or heap area bounds cause the command process to generate an illegal memory reference trap.

The default maximum sizes may be changed via SetDataBounds, where **desiredStackMax** and **desiredHeapMax** specify the maximum sizes in bytes for the stack and heap, respectively. These values should be positive numbers, and will be rounded up to a multiple of 4096 if necessary. The values **resultingStackMax** and **resultingHeapMax** are returned and indicate the new maximum stack and heap sizes, respectively.

The amount of secondary storage space allocated to the data segment is not changed by SetDataBounds.

SEE ALSO

LoadCommand (3B)

NOTES

If the desired size is less than the currently allocated size, then the maximum size is not changed, and the current maximum size is returned.

NAME

StartCommand – start a command process executing

SYNTAX

```
FUNCTION StartCommand (pID : ProcessID;
                       args : PArgPage;
                       wait : Boolean;
                       debugFlag: Boolean) : Error;
```

DESCRIPTION

StartCommand is used to start execution of a command process. The command process **pID** must have been created by the LoadCommand or LoadCodeAndData routine; StartCommand activates **pID**, supplying its command arguments.

The command arguments are passed in a 4096-byte page to the User Monitor, which transmits them to the command process when it calls GetArgs. The parameter **args** points to a page which contains the arguments in the following format:

```
PArgPage = ^ ArgPage;
ArgPage  = Record
              argc : Integer;
              strings : Array [0..0] of StringBody;
           end;
```

The argument page contains an argument count and zero or more strings packed sequentially. The strings consist of a length field followed by the sequence of characters, with the length fields aligned on word (4-byte) boundaries. By convention, the argument count is at least one, with the first string being the command name the process was invoked with.

If **wait** is True, then the invoking process is suspended until the command process terminates. In this case, the exit code of the command process is returned as the value of the StartCommand routine. If **wait** is False, then the invoking process is not suspended and no indication is given when the command process terminates.

If **debugFlag** is true, then the command process is not activated, and the Debug process is notified that **pID** is a suspended process. The command process can be activated by Debug, usually after breakpoints have been set. if **debugFlag** is false, the command process is activated immediately.

SEE ALSO

AbortCommand (3B), GetArgs (3B), LoadCodeAndData (3B), LoadCommand (3B), SysExit (3B)

NOTES

ErBadPID is returned if **pID** is not a valid command process ID.

ErCantStart is returned if the command process cannot be properly activated.

NAME

SubString – make a copy of a substring

SYNTAX

```
FUNCTION SubString (s : String;
                    first : Integer;
                    last : Integer) : String;
```

DESCRIPTION

SubString creates a new string that is a substring of string **s**. The characters from position **first** through position **last** in **s** are copied into the new string, which is then returned. String **s** is not modified.

SEE ALSO

CopySubString (3B), NewString (3B)

NOTES

No bounds check is made to insure that the substring is completely contained within the string.

NAME

SysExit – exit back to the system

SYNTAX

PROCEDURE SysExit (errorCode : Error);

DESCRIPTION

SysExit exits back to the system, thus terminating the calling command process. Any open stream files are closed before the process is terminated.

The exit code **errorCode** is returned to the invoking process which started the command process via StartCommand. By convention, the value zero indicates successful completion, while nonzero **errorCode** values indicate different errors defined by the command process.

SEE ALSO

CloseFile (3B), StartCommand (3B)

NOTES

This routine never returns to its caller.

NAME

WriteBlock – write a block of a file

SYNTAX

```
FUNCTION WriteBlock (handle : Link;
                     bufAddr : PageAddress;
                     fileCursor : Integer;
                     length : Integer;
                 var actual : Integer) : Error;
```

DESCRIPTION

WriteBlock is used to write a block of data to the file specified by **handle**. The **handle** is a file link returned from a successful Create or Open call, and it represents a file that must allow block mode access.

The block of data from the buffer address in the user process data segment specified by **bufAddr**, which must be page-aligned, is transferred to the file at the byte position specified by **fileCursor**.

The amount of data to be written is specified in **length**, and the amount actually transferred is returned in **actual**, which can be from 0 to 4096 bytes.

The file size is increased by any bytes which extend past the current end of file.

SEE ALSO

Create (3B), Open (3B), ReadBlock (3B), ReadChar (3B), WriteChar (3B)

NOTES

ErBadLink is returned if **handle** is not a valid open file.

ErNotAligned is returned if **bufAddr** is not page-aligned, that is, an address that is not a multiple of 4096, while ErBadBlockLength is returned if **length** is greater than 4096.

ErNotWritable is returned if the file was not opened to allow writing.

ErBadFileCursor is returned if an attempt is made to write a block beyond the current end of file, which would leave a gap in the file.

NAME

WriteChar – write a character

SYNTAX

```
FUNCTION WriteChar (handle : Link;
                    ch : Char) : Error;
```

DESCRIPTION

WriteChar writes a single character to the file specified by **handle**. The **handle** is a file link returned from a successful Create or Open call, and it represents a file that must allow character mode access.

The character **ch** that is written can be any 8-bit value.

SEE ALSO

Create (3B), Open (3B), ReadBlock (3B), ReadChar (3B), WriteBlock (3B)

NOTES

ErBadLink is returned if **handle** is not a valid open file.

ErNotWritable is returned if the file was not opened to allow writing.

NAME

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

SYNTAX

long a64l (s)
char *s;

char *l64a (l)
long l;

DESCRIPTION

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

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

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

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

BUGS

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

NAME

abort – generate an IOT fault

SYNTAX

int abort ( )

DESCRIPTION

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

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

SEE ALSO

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

DIAGNOSTICS

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

NAME

abs – return integer absolute value

SYNTAX

int abs (i)
int i;

DESCRIPTION

Abs returns the absolute value of its integer operand.

BUGS

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

SEE ALSO

NAME

atof – convert ASCII string to floating-point number

SYNTAX

double atof (nptr)
char *nptr;

DESCRIPTION

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

DIAGNOSTICS

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

SEE ALSO

scanf(3S).

NAME

bsearch – binary search

SYNTAX

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

DESCRIPTION

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

DIAGNOSTICS

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

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

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

NAME

clock – report CPU time used

SYNTAX

long clock ( )

DESCRIPTION

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

The resolution of the clock is 1 millisecond on the Ridge 32, 10 milliseconds on Western Electric 3B processors, 16.667 milliseconds on Digital Equipment Corporation processors.

SEE ALSO

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

BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

NAME

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

SYNTAX

#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;

DESCRIPTION

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

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

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

SEE ALSO

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

NAME

crypt, setkey, encrypt – generate DES encryption

SYNTAX

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

void setkey (key)
char *key;

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

DESCRIPTION

Crypt is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for 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 DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

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

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

SEE ALSO

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

BUGS

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

NAME

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

SYNTAX

```
#include <time.h>

char *ctime (clock)
long *clock;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ( )
```

DESCRIPTION

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

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

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

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

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

```
struct tm {
      int tm_sec; /* seconds (0 - 59) */
      int tm_min; /* minutes (0 - 59) */
      int tm_hour; /* hours (0 - 23) */
      int tm_mday; /* day of month (1 - 31) */
      int tm_mon; /* month of year (0 - 11) */
      int tm_year; /* year - 1900 */
      int tm_wday; /* day of week (Sunday = 0) */
      int tm_yday; /* day of year (0 - 365) */
      int tm_isdst;
};
```

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

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

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

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

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

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

SEE ALSO

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

BUGS

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

NAME

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

SYNTAX

#include <ctype.h>

int isalpha (c)
int c;

. . .

DESCRIPTION

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

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

DIAGNOSTICS

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

SEE ALSO

ascii(5).

NAME

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

SYNTAX

double drand48 ( )

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

long lrand48 ( )

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

long mrand48 ( )

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

void srand48 (seedval)
long seedval;

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

NOTES

The versions of these routines for the Ridge 32 are coded in portable C.

SEE ALSO

rand(3C).

NAME

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

SYNTAX

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

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

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

DESCRIPTION

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

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

Gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf.* It attempts to produce *ndigit* significant digits in Fortran F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

SEE ALSO

printf(3S).

BUGS

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

NAME

end, etext, edata – last locations in program

SYNTAX

extern end;
extern etext;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with *end*, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output (*stdio*(3S)), the profile (**– p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk(0)** (see *brk*(2)).

SEE ALSO

brk(2), malloc(3C).

NAME

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

SYNTAX

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

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

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

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \leq |x| < 1.0$, and the "exponent" n is an integer. *Frexp* returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*.

Ldexp returns the quantity $value * 2^{exp}$.

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

DIAGNOSTICS

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

NAME

ftw – walk a file tree

SYNTAX

#include <ftw.h>

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

DESCRIPTION

Ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure (see *stat*(2)) containg information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

Ftw visits a directory before visiting any of its descendants.

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

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

SEE ALSO

stat(2), malloc(3C).

BUGS

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

NAME

getcwd – get path-name of current working directory

SYNTAX

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

DESCRIPTION

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

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

EXAMPLE

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

SEE ALSO

pwd(1), malloc(3C).

DIAGNOSTICS

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

NAME

getenv – return value for environment name

SYNTAX

char *getenv (name) char *name;

DESCRIPTION

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

SEE ALSO

environ(5).

NAME

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

SYNTAX

#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )

DESCRIPTION

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

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

When first called, *Getgrent* returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *Getgrgid* searches from the beginning of the file until a numerical group id matching *gid* is found and returns a pointer to the particular structure in which it was found. *Getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

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

FILES

/etc/group

SEE ALSO

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

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

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

BUGS

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

NAME

getlogin – get login name

SYNTAX

char *getlogin ( );

DESCRIPTION

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

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

FILES

/etc/utmp

SEE ALSO

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

DIAGNOSTICS

Returns the NULL pointer if *name* not found.

BUGS

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

NAME

getopt – get option letter from argument vector

SYNTAX

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

extern char *optarg;
extern int optind;

DESCRIPTION

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

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

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

DIAGNOSTICS

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

WARNING

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

EXAMPLE

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

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern int optind;
        extern char *optarg;
        .
        .
        .
        while ((c = getopt (argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
```

```
                                bproc( );
                        break;
                case 'f':
                        ifile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        bufsiza = 512;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                fprintf (stderr, "usage: . . . ");
                exit  2);
        }
        for ( ; optind < argc; optind++) {
                if (access (argv[optind], 4)) {
        •
        •
        •
}
```

SEE ALSO

getopt(1).

NAME

getpass – read a password

SYNTAX

char *getpass (prompt)
char *prompt;

DESCRIPTION

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

FILES

/dev/tty

SEE ALSO

crypt(3C).

WARNING

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

BUGS

The return value points to static data

NAME

getpw – get name from UID

SYNTAX

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

DESCRIPTION

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

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

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

Getpw returns non-zero on error.

WARNING

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

NAME

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

SYNTAX

#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )

DESCRIPTION

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

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

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

This structure is declared in <*pwd.h*> so it is not necessary to redeclare it.

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

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

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

FILES

/etc/passwd

SEE ALSO

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

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

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

BUGS

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

q f

f

NAME

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

SYNTAX

```
#include <utmp.h>

struct utmp *getutent ( )

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

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

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

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

DESCRIPTION

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

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

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

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

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

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

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

Endutent closes the currently open file.

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

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

COMMENTS

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* if it finds that it isn't already at the correct place in the file will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

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

eq f

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNTAX

```
#include <search.h>

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

int hcreate (nel)
unsigned nel;

void hdestroy ( )
```

DESCRIPTION

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

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

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

NOTES

Hsearch uses *open addressing* with a *multiplicative* hash function. However, its source code has many other options available which the user may select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV	Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.
USCR	Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a mannner similar to *strcmp* (see *string*(3C)).
CHAINED	Use a linked list to resolve collisions. If this option is selected, the following other options become available.

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

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

SEE ALSO

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

DIAGNOSTICS

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

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

BUGS

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

eq f

eq f

NAME

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

SYNTAX

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

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

DESCRIPTION

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

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

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

SEE ALSO

fs(4).

BUGS

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

NAME

lsearch, lfind – linear search and update

SYNTAX

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

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );
```

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. **Key** points to the datum to be sought in the table. **Base** points to the first element in the table. **Nelp** points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. **Compar** is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

Lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

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

EXAMPLE

This fragment will read in ≤ TABSIZE strings of length ≤ ELSIZE and store them in a table, eliminating duplicates.

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

#define TABSIZE 50
#define ELSIZE 120

        char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
        unsigned nel = 0;
        int strcmp( );
        . . .
        while (fgets(line, ELSIZE, stdin) != NULL &&
           nel < TABSIZE)
               (void) lsearch(line, (char *)tab, &nel,
                         ELSIZE, strcmp);
        . . .
```

SEE ALSO

bsearch(3C), hsearch(3C), tsearch(3C).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

malloc, free, realloc, calloc – main memory allocator

SYNTAX

char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use. On the Ridge 32, blocks obtained from *malloc* are aligned on even 8-byte boundaries.

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

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

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

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, then *realloc* will ask *malloc* to enlarge the arena by *size* bytes and will then move the data to the new space.

Realloc also works if *ptr* points to a block freed since the last call of *malloc, realloc,* or *calloc*; thus sequences of *free, malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS

Malloc, realloc and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

NOTE

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer.

NAME

memccpy, memchr, memcmp, memcpy, memset – memory operations

SYNTAX

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memccpy copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. It returns a pointer to the character after the copy of *c* in *s1*, or a NULL pointer if *c* was not found in the first *n* characters of *s2*.

Memchr returns a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s*, or a NULL pointer if *c* does not occur.

Memcmp compares its arguments, looking at the first *n* characters only, and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*.

Memcpy copies *n* characters from memory area *s2* to *s1*. It returns *s1*.

Memset sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s* .

NOTE

For user convenience, all these functions are declared in the optional <*memory.h*> header file.

BUGS

Memcmp uses native character comparison, which is signed on PDP-11s, unsigned on other machines.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

mktemp – make a unique file name

SYNTAX

char *mktemp (template)
char *template;

DESCRIPTION

Mktemp replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing **X**s; *mktemp* will replace the **X**s with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

BUGS

It is possible to run out of letters.

xeq f

NAME

nlist – get entries from name list

SYNTAX

#include <nlist.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*(4) for the structure declaration.

SEE ALSO

a.out(4)

DIAGNOSTICS

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

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNTAX

void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;

DESCRIPTION

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2).

NAME

putpwent – write password file entry

SYNTAX

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

Putpwent is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwuid* writes a line on the stream *f* which matches the format of **/etc/passwd**.

DIAGNOSTICS

Putpwent returns non-zero if an error was detected during its operation, otherwise zero.

WARNING

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

NAME

qsort – quicker sort

SYNTAX

void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned int nel;
int (*compar)( );

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Base points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

NOTES

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

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), string(3C).

NAME

rand, srand – simple random-number generator

SYNTAX

int rand ( )

void srand (seed)
unsigned seed;

DESCRIPTION

Rand uses a multiplicative congruential random-number generator with period 2**32 that returns successive pseudo-random numbers in the range 0 to 2**15-1.

Srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTES

The spectral properties of *rand* leave a great deal to be desired. *Drand48* (3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48 (3C)

NAME

setjmp, longjmp – non-local goto

SYNTAX

#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* (whose type, *jmp_buf,* is defined in the *<setjmp.h>* header file), for later use by *longjmp*. It returns the value 0.

Longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. All accessible data have values as of the time *longjmp* was called.

SEE ALSO

signal(2B).

WARNING

If *longjmp* is called when *env* was never primed by a call to *setjmp,* or when the last such call is in a function which has since returned, absolute chaos is guaranteed.

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep (seconds)
unsigned seconds;

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*; if the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred, and the caller's alarm catch routine is executed just before the *sleep* routine returns, but if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2B), pause(2B), signal(2B).

NAME

ssignal, gsignal – software signals

SYNTAX

```
#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;
```

DESCRIPTION

Ssignal and *gsignal* implement a software facility similar to *signal*(2B). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions, and is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. A call to *ssignal* associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user defined) *action function* or one of the manifest constants SIG_DFL (default) or SIG_IGN (ignore). *Ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns SIG_DFL.

Gsignal raises the signal identified by its argument, *sig*:

> If an action function has been established for *sig*, then that action is reset to SIG_DFL and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.
>
> If the action for *sig* is SIG_IGN, *gsignal* returns the value 1 and takes no other action.
>
> If the action for *sig* is SIG_DFL, *gsignal* returns the value 0 and takes no other action.
>
> If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

NOTES

There are some additional signals with numbers outside the range 1 through 15 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

eq f

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok – string operation

SYNTAX

#include <string.h>

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

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

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

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

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

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

int strlen (s)
char *s;

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

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

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

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

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

DESCRIPTION

The arguments *s1, s2* and *s* point to strings (arrays of characters terminated by a null character). The functions *strcat, strncat, strcpy* and *strncpy* all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

Strcat appends a copy of string *s2* to the end of string *s1*. *Strncat* appends at most *n* characters. Each returns a pointer to the null-terminated result.

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

Strcpy copies string *s2* to *s1*, stopping after the null character has been copied. *Strncpy* copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

Strlen returns the number of characters in *s*, not including the terminating null character.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a NULL pointer if no character from *s2* exists in *s1*.

Strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

Strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that on subsequent calls (which must be made with the first argument a NULL pointer) will work through the string *s1* immediately following that token. In this way subsequent calls will work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

NOTE

For user convenience, all these functions are declared in the optional <*string.h*> header file.

BUGS

Strcmp and *strncmp* use native character comparison, which is signed on PDP-11s, unsigned on other machines.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

strtol, atol, atoi – convert string to integer

SYNTAX

long strtol (str, ptr, base)
char *str;
char **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;

DESCRIPTION

Strtol returns as a long integer the value represented by the character string *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in **ptr*. If no integer can be formed, **ptr* is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thus: After an optional leading sign, a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment, or by an explicit cast.

Atol(str) is equivalent to *strtol(str, (char **)NULL, 10)*.

Atoi(str) is equivalent to *(int) strtol(str, (char **)NULL, 10)*.

SEE ALSO

atof(3C), scanf(3S).

BUGS

Overflow conditions are ignored.

NAME

swab – swap bytes

SYNTAX

void swab (from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP-11s and other machines. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*– 1 instead. If *nbytes* is negative *swab* does nothing.

NAME

tsearch, tdelete, twalk – manage binary search trees

SYNTAX

#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );

DESCRIPTION

Tsearch is a binary tree search routine generalized from Knuth (6.2.2) Algorithm T. It returns a pointer into a tree indicating where a datum may be found. If the datum does not occur, it is added at an appropriate point in the tree. *Key* points to the datum to be sought in the tree. *Rootp* points to a variable that points to the root of the tree. A NULL pointer value for the variable denotes an empty tree; in this case, the variable will be set to point to the datum at the root of the new tree. *Compar* is the name of the comparison function. It is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

Tdelete deletes a node from a binary search tree. It is generalized from Knuth (6.2.2) algorithm D. The arguments are the same as for *tsearch*. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. *Tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

Twalk traverses a binary search tree. *Root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* (defined in the <search.h> header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

NOTES

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.
Warning: the *root* argument to *twalk* is one level of indirection less than the *rootp* arguments to *tsearch* and *tdelete*.

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.
A NULL pointer is returned by *tsearch* and *tdelete* if *rootp* is NULL on entry.

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

BUGS

Awful things can happen if the calling function alters the pointer to the root.

NAME

ttyname, isatty – find name of a terminal

SYNTAX

char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;

DESCRIPTION

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

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

FILES

/dev/*

DIAGNOSTICS

Ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory /dev.

BUGS

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

NAME

ttyslot – find the slot in the utmp file of the current user

SYNTAX

int ttyslot ( )

DESCRIPTION

Ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/inittab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

FILES

/etc/inittab
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

BLANK

NAME

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

SYNTAX

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

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

double y0 (x)
double x;

double y1 (x)
double x;

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

DESCRIPTION

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

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

DIAGNOSTICS

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

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

SEE ALSO

matherr(3M).

NAME

erf, erfc – error function and complementary error function

SYNTAX

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

DESCRIPTION

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

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

SEE ALSO

exp(3M).

NAME

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

SYNTAX

#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

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

double sqrt (x)
double x;

DESCRIPTION

Exp returns e^x.

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

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

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

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

DIAGNOSTICS

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

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

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

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

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

SEE ALSO

hypot(3M), matherr(3M),

NAME

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

SYNTAX

#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

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

double fabs (x)
double x;

DESCRIPTION

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

Ceil returns the smallest integer not less than x.

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

Fabs returns $|x|$.

SEE ALSO

abs(3C).

NAME

gamma – log gamma function

SYNTAX

```
#include <math.h>

extern int signgam;

double gamma (x)
double x;
```

DESCRIPTION

delim $$ *Gamma* returns $ln (|GAMMA (^ x) |)$, where $GAMMA (^ x)$ is defined as $int from 0 to inf e sup { - t } t sup { x - 1 } dt$. The sign of GAMMA (^ x) is returned in the external integer *signgam*. The argument *x* may not be a non-positive integer.

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

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

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

DIAGNOSTICS

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

If the correct value would overflow, *gamma* returns HUGE and sets *errno* to ERANGE.

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

SEE ALSO

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

NAME

hypot – Euclidean distance function

SYNTAX

#include <math.h>

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

DESCRIPTION

Hypot returns

sqrt(x * x + y * y),

taking precautions against unwarranted overflows.

DIAGNOSTICS

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

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

SEE ALSO

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

NAME

matherr – error-handling function

SYNTAX

#include <math.h>

int matherr (x)
struct exception *x;

DESCRIPTION

Matherr is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors by including a function named *matherr* in their programs. *Matherr* must be of the form described above. A pointer to the exception structure *x* will be passed to the user-supplied *matherr* function when an error occurs. This structure, which is defined in the *<math.h>* header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	domain error
SING	singularity
OVERFLOW	overflow
UNDERFLOW	underflow
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element *name* points to a string containing the name of the function that had the error. The variables *arg1* and *arg2* are the arguments to the function that had the error. *Retval* is a double that is returned by the function having the error. If it supplies a return value, the user's *matherr* must return non-zero. If the default error value is to be returned, the user's *matherr* must return 0.

If *matherr* is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to non-zero and the program continues.

EXAMPLE

```
matherr(x)
register struct exception *x;
{
        switch (x– >type) {
        case DOMAIN:
        case SING: /* print message and abort */
                fprintf(stderr, "domain error in %s\n", x– >name);
                abort( );
        case OVERFLOW:
                if (!strcmp("exp", x– >name)) {
                        /* if exp, print message, return the argument */
                        fprintf(stderr, "exp of %f\n", x– >arg1);
                        x– >retval = x– >arg1;
                } else if (!strcmp("sinh", x– >name)) {
                        /* if sinh, set errno, return 0 */
                        errno = ERANGE;
                        x- >retval = 0;
                } else
                        /* otherwise, return HUGE */
                        x- >retval = HUGE;
                break;
        case UNDERFLOW:
                return (0); /* execute default procedure */
        case TLOSS:
        case PLOSS:
                /* print message and return 0 */
                fprintf(stderr, "loss of significance in %s\n", x- >name);
                x- >retval = 0;
                break;
        }
        return (1);
}
```

DEFAULT ERROR HANDLING PROCEDURES						
	Types of Errors					
	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
BESSEL:	–	–	H	0	–	*
y0, y1, yn (neg. no.)	M, – H	–	–	–	–	–
EXP:	–	–	H	0	–	
POW:	–	–	H	0	–	–
(neg.)**(non-int.), 0**0	M, 0	–	–	–	–	–
LOG:						
log(0):	–	M, – H	–	–	–	–
log(neg.):	M, – H	–	–	–	–	–
SQRT:	M, 0	–	–	–	–	–
GAMMA:	–	M, H	–	–	–	–
HYPOT:	–	–	H	–	–	–
SINH, COSH:	–	–	H	–	–	–
SIN, COS:	–	–	–	–	M, 0	M, *
TAN:	–	–	H	–	0	*
ACOS, ASIN:	M, 0	–	–	–	–	–

ABBREVIATIONS	
*	As much as possible of the value is returned.
M	Message is printed.
H	HUGE is returned.
– H	– HUGE is returned.
0	0 is returned.

NAME

sinh, cosh, tanh – hyperbolic functions

SYNTAX

```
#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

DESCRIPTION

Sinh, *cosh* and *tanh* return respectively the hyberbolic sine, cosine and tangent of their argument.

DIAGNOSTICS

Sinh and *cosh* return HUGE when the correct value would overflow, and set *errno* to ERANGE.

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

SEE ALSO

matherr(3M).

NAME

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

SYNTAX

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double tan (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double x, y;
```

DESCRIPTION

Sin, cos and *tan* return respectively the sine, cosine and tangent of their argument, which is in radians.

Asin returns the arcsine of x, in the range $-\pi/2$ to $\pi/2$.

Acos returns the arccosine of x, in the range 0 to π.

Atan returns the arctangent of x, in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arctangent of y/x, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

DIAGNOSTICS

Sin, cos and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0 when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to ERANGE.

Tan returns HUGE for an argument which is near an odd multiple of $\pi/2$ when the correct value would overflow, and sets *errno* to ERANGE.

Arguments of magnitude greater than 1.0 cause *asin* and *acos* to return 0 and to set *errno* to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

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

SEE ALSO

matherr(3M).

NAME

ctermid – generate file name for terminal

SYNTAX

#include <stdio.h>

char *ctermid(s)
char *s;

DESCRIPTION

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

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

NOTES

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

SEE ALSO

ttyname(3C).

NAME

cuserid – get character login name of the user

SYNTAX

#include <stdio.h>

char *cuserid (s)
char *s;

DESCRIPTION

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

DIAGNOSTICS

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

SEE ALSO

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

NAME

fclose, fflush – close or flush a stream

SYNTAX

#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;

DESCRIPTION

Fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

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

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

DIAGNOSTICS

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

SEE ALSO

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

q f

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNTAX

```
#include <stdio.h>

int feof (stream)
FILE
*stream;

int ferror (stream)
FILE
*stream;

void clearerr (stream)
FILE
*stream;

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

DESCRIPTION

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

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

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

Fileno returns the integer file descriptor associated with the named *stream*; see *open*(2).

NOTE

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

SEE ALSO

open(2), fopen(3S).

NAME

fopen, freopen, fdopen – open a stream

SYNTAX

#include <stdio.h>

FILE *fopen (file-name, type)
char *file-name, *type;

FILE *freopen (file-name, type, stream)
char *file-name, *type;
FILE *stream;

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

DESCRIPTION

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

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

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

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

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

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

Fdopen associates a *stream* with a file descriptor obtained from *open*, *dup*, or *creat*(2), which will open files but not return pointers to a FILE structure *stream* which are necessary input for many of the section 3S library routines. The *type* of *stream* must agree with the mode of the open file.

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

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

SEE ALSO

open(2), fclose(3S).

DIAGNOSTICS

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

NAME

fread, fwrite – binary input/output

SYNTAX

#include <stdio.h>

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

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

DESCRIPTION

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

NAME

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

SYNTAX

```
#include <stdio.h>

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

void rewind (stream)
FILE *stream;

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

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according as *ptrname* has the value 0, 1, or 2.

Rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0), except that no value is returned.

Fseek and *rewind* undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

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

SEE ALSO

lseek(2), fopen(3S).

DIAGNOSTICS

Fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal.

WARNING

Although on the UNIX System an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX Systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such a offset, which is not necessarily measured in bytes.

NAME

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

SYNTAX

#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ( )

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;

DESCRIPTION

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

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

BUGS

Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. In particular, **getc(*f++)** doesn't work sensibly. *Fgetc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

gets, fgets – get a string from a stream

SYNTAX

#include <stdio.h>

char *gets (s)
char *s;

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

DESCRIPTION

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

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

SEE ALSO

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

DIAGNOSTICS

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

NAME

popen, pclose – initiate pipe to/from a process

SYNTAX

#include <stdio.h>

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

int pclose (stream)
FILE *stream;

DESCRIPTION

The arguments to *popen* are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either **r** for reading or **w** for writing. *Popen* creates a pipe between the calling program and the command to be executed. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is **w**, by writing to the file *stream*; and one can read from the standard output of the command, if the I/O mode is **r**, by reading from the file *stream*.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter and a type **w** as an output filter.

SEE ALSO

wait(2B), fclose(3S), fopen(3S), system(3S).

DIAGNOSTICS

Popen returns a NULL pointer if files or processes cannot be created, or if the shell cannot be accessed.

Pclose returns – 1 if *stream* is not associated with a "*popen*ed" command.

BUGS

If the original and "*popen*ed" processes concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Problems with an output filter may be forestalled by careful buffer flushing, e.g. with *fflush*; see *fclose*(3S).

NAME

printf, fprintf, sprintf – print formatted output

SYNTAX

#include <stdio.h>

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

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

int sprintf (s, format [**,** arg] ... **)**
char *s, format;

DESCRIPTION

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output", followed by the null character (**\0**) in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **\0** in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *arg*s under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

Each conversion specification is introduced by the character **%** After the **%** the following appear in sequence:

> Zero or more *flags*, which modify the meaning of the conversion specification.
>
> An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag (see below) has been given) to the field width;
>
> A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (**.**) followed by a decimal digit string: a null digit string is treated as zero.
>
> An optional **l** specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*.
>
> A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*****) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

–	The result of the conversion will be left-justified within the field.
+	The result of a signed conversion will always begin with a sign (+ or –).
blank	If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

#	This flag specifies that the value is to be converted to an "alternate form."For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** (**X**) conversion, a non-zero result will have **0x** (**0X**) prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X	The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters ABCDEF for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
f	The float or double *arg* is converted to decimal notation in the style "[−]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are output; if the precision is explicitly 0, no decimal point appears.
e,E	The float or double *arg* is converted in the style "[−]d.ddde± dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.
g,G	The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than − 4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.
c	The character *arg* is printed.
s	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. If the string pointer *arg* has the value zero, the result is undefined. A *null* arg will yield undefined results.
%	Print a **%** no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %.2d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

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

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S).

NAME

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

SYNTAX

```
#include <stdio.h>

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

int putchar (c)
char c;

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

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

DESCRIPTION

Putc writes the character *c* onto the output *stream* (at the position where the file pointer, if defined, is pointing). *Putchar*(*c*) is defined as *putc*(*c, stdout*). *Putc* and *putchar* are macros.

Fputc behaves like *putc*, but is a function rather than a macro. *Fputc* runs more slowly than *putc*, but takes less space per invocation.

Putw writes the word (i.e. integer) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. *Putw* neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen*(see *fopen*(3S)) will cause it to become buffered or line-buffered. When an output stream is unbuffered information is queued for writing on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block; when it is line-buffered each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). *Setbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant EOF. This will occur if the file *stream* is not open for writing, or if the output file cannot be grown. Because EOF is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

BUGS

Because it is implemented as a macro, *putc* treats incorrectly a *stream* argument with side effects. In particular, **putc(c, *f++)**; doesn't work sensibly. *Fputc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor. For this reason the use of *putw* should be avoided.

NAME

puts, fputs – put a string on a stream

SYNTAX

#include <stdio.h>

int puts (s)
char *s;

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

DESCRIPTION

Puts writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream *stdout.*

Fputs writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

DIAGNOSTICS

Both routines return EOF on error. This will happen if the routines try to write on a file that has not been opened for writing.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

NOTES

Puts appends a new-line character while *fputs* does not.

NAME

scanf, fscanf, sscanf – convert formatted input

SYNTAX

#include <stdio.h>

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

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

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

DESCRIPTION

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

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

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character % an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** or **h** indicating the size of the receiving variable, and a conversion code.

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

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument should be given. The following conversion codes are legal:

%	a single %is expected in the input at this point; no assignment is done.
d	a decimal integer is expected; the corresponding argument should be an integer pointer.
u	an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.
o	an octal integer is expected; the corresponding argument should be an integer pointer.
x	a hexadecimal integer is expected; the corresponding argument should be an integer pointer.
e,f,g	a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optionally signed integer.
s	a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a white-space character.

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

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset,* and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex, (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first-last,* thus [0123456789] may be expressed [0-9]. Using this convention, *first* must be lexically less than or equal to *last,* or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which will be added automatically.

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

Scanf conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

EXAMPLES

The call:

```
int 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**, to *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

```
int i; float x; char name[50];
scanf ("%2d%f%*d %[0-9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

atof(3C), getc(3S), printf(3S), strtol(3C).

NOTE

Trailing white space (including a new-line) is left unread unless matched in the control string.

DIAGNOSTICS

These functions return EOF on end of input and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf, setvbuf – assign buffering to a stream

SYNTAX

#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;

DESCRIPTION

Setbuf may be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer input/output will be completely unbuffered.

A constant BUFSIZ, defined in the **<stdio.h>** header file, tells how big an array is needed:

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

Setvbuf may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type* (defined in stdio.h) are:

_IOFBF	causes input/output to be fully buffered.
_IOLBF	causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.
_IONBF	causes input/output to be completely unbuffered.

If *buf* is not the NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *Size* specifies the size of the buffer to be used. The constant BUFSIZ in **<stdio.h>** is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

By default, output to a terminal is line buffered and all other input/output is fully buffered.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S), stdio(3S).

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

NOTE

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

NAME

stdio – standard buffered input/output package

SYNTAX

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar*, *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type FILE. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

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

A constant NULL (0) designates a nonexistent pointer.

An integer constant EOF (– 1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

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

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that header file and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, *clearerr*, and *fileno*.

SEE ALSO

open(2B), close(2B), lseek(2B), read(2B), write(2B), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

system – issue a shell command

SYNTAX

```
#include <stdio.h>

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

DESCRIPTION

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

FILES

/bin/sh

SEE ALSO

sh(1), exec(2B).

DIAGNOSTICS

System forks to create a child process that in turn exec's */bin/sh* in order to execute *string*. If the fork or exec fails, *system* returns – 1 and sets *errno*.

NAME

tmpfile – create a temporary file

SYNTAX

```
#include <stdio.h>

FILE *tmpfile ( )
```

DESCRIPTION

Tmpfile creates a temporary file and returns a corresponding FILE pointer. The file will automatically be deleted when the process using it terminates. The file is opened for update.

SEE ALSO

creat(2B), unlink(2B), mktemp(3C), tmpnam(3S).

NAME

tmpnam, tempnam – create a name for a temporary file

SYNTAX

#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

Tmpnam always generates a file name using the path-name defined as **P_tmpdir** in the *<stdio.h>* header file. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in *<stdio.h>*; *tmpnam* places its result in that array and returns *s*.

Tempnam allows the user to control the choice of a directory. The argument *dir* points to the path-name of the directory in which the file is to be created. If *dir* is NULL or points to a string which is not a path-name for an appropriate directory, the path-name defined as **P_tmpdir** in the *<stdio.h>* header file is used. If that path-name is not accessible, **/tmp** will be used as a last resort. This entire sequence can be up-staged by providing an environment variable **TMPDIR** in the user's environment, whose value is a path-name for the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

Tempnam uses *malloc*(3C) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* (see *malloc*(3C)). If *tempnam* cannot return the expected result for any reason, i.e. *malloc* failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either *fopen* or *creat* are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2B) to remove the file when its use is ended.

SEE ALSO

creat(2B), unlink(2B), malloc(3C), mktemp(3C), tmpfile(3S).

BUGS

If called more than 17,576 times in a single process, these functions will start recycling previously used names.
Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

NAME

ungetc – push character back into input stream

SYNTAX

#include <stdio.h>

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

DESCRIPTION

Ungetc inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc* call on that *stream*. *Ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered.

If *c* equals EOF, *ungetc* does nothing to the buffer and returns EOF.

Fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S).

DIAGNOSTICS

In order that *ungetc* perform correctly, a read statement must have been performed prior to the call of the *ungetc* function. *Ungetc* returns EOF if it can't insert the character. In the case that *stream* is *stdin*, *ungetc* will allow exactly one character to be pushed back onto the buffer without a previous read statement.

BLANK

NAME

assert – verify program assertion

SYNTAX

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *assert* prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

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

SEE ALSO

cpp(1), abort(3C).

NAME

CopyBits – bit-oriented block transfer (BitBlt)

SYNTAX

#include <sys/graf.h>

CopyBits (rOp)
struct RasterOp *rOp;

DESCRIPTION

The *CopyBits* fuction provides pixel-level graphics operations for the Ridge Monochrome Display. It is used to create bitmap images in the calling process' own address space, a part of which is mapped to the display screen as described in the *Ridge Multi-Window Display Management Guide*.

Rectangular areas of arbitrary size can be filled with black, white, and halftone patterns. Text, in various typefaces or fonts, can be copied from stored images of the individual characters. Line-drawing and arbitrary graphics are created by repeated calls of the basic functions.

The *CopyBits* operation is modelled after the BitBlt (bit-oriented block transfer) operation found in Smalltalk (Goldberg and Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983).

A two-dimensional pixel image is represented by a data structure called a *Form*, whose definition is given in the include file as:

```
struct Form {
      int width;
      int height;
      unsigned int *bits;
};
```

A Form has height and width, measured in bits, and a pointer to an array of 32-bit words that contains a bitmap. The bitmap is a pattern of ones and zeros that indicate the black and white pixels of the image being represented. The height and width impose two-dimensional ordering on the otherwise unstructured bitmap data.

A new Form is usually created by allocating an integral number of words to contain the bits needed for the desired height and width, and placing a pointer to those words in the structure. A Form might also be initialized with the width and height of the display screen, and contain the starting address in virtual memory of the display bitmap. There is no difference between manipulating images internally and manipulating images that are mapped to the display screen.

Bitmaps are allocated with an integral number of 32-bit words for each row of pixels. This row size is referred to as the "raster" width. The integral word constraint on the raster size is related to the hardware organization of memory and the efficiency of processing 32 bits at a time. This facilitates movement from one row to the next, both during the hardware scanning of the display screen memory and during the operation of the *CopyBits* function. While this organization is significant for allocation purposes, it is encapsulated so that none of the higher-level callers of *CopyBits* need be concerned with the issue of word size.

Locations within a Form are designated using a two-dimensional coordinate system relative to the top left corner of the image rectangle. X increases to the right and Y increases down, consistent with the display device raster scanning, and with the layout of text on a page.

Two structures, defined in the include file, are often used when dealing with bitmap images:

```
struct Point {
      int x;
      int y;
};
```

```
struct Rectangle {
        struct Point origin;
        struct Point extent;
};
```

A Point contains x and y coordinate values that refer to a pixel location with a Form. A Rectangle contains two Points (the origin is the top left corner and the extent is the width and height) that define a rectangular area within a Form.

The *CopyBits* operation involves a source Form and a destination Form. For example, the source may be a set of character glyphs packed together horizontally as in a bit-matrix *font*(4). The destination might be the display bitmap. Pixels are copied out of the source and stored into the destination. The width and height of the transfer correspond to the character size. The source coordinates give the character's location in the font, and the destination coordinates specify the position on the display where the copy will appear.

A clipping rectangle is specified which limits the region of the destination that can be affected by the operation, independent from the other destination parameters. Often it is desirable to display a partial view of a larger image, and the clipping rectangle ensures that all picture elements fall inside the bounds of the view. Pixels that would have been placed outside the clipping rectangle are not transferred.

Occasionally it is desirable to fill areas with a regular pattern that gives the effect of a gray halftone or texture. The *CopyBits* operation allows a third Form to be specified containing the desired pattern, and is referred to as the mask. The mask Form must have a width and height of exactly 32 bits. When halftoning is specified, this pattern is replicated every 32 bits horizontally and vertically over the entire destination.

There are four possible ways to supply pixels from the source and mask Forms, depending on which Forms are used in the operation. To exclude use of a Form parameter, set the *bits* field within the Form to a pointer value 0:

1) no source, no mask – supplies solid black
2) source only – supplies source pixels
3) mask only – supplies halftone pattern
4) source AND mask – supplies source pixels logically masked by halftone pattern

There are 16 possible rules for combining each source element **S** (taken as the result of any halftone masking) with the corresponding destination element **D** to produce the new destination element **D'**. Each rule must specify a white (0) or black (1) result bit for each of the four cases of the source being white or black, and the destination being white or black. The following list gives the integer rule, the name from the include file, and the Boolean operation:

0	WHITing	D' = 0
1	ANDing	D' = S AND D
2	SANDNOTDing	D' = S AND NOT D
3	STORing	D' = S
4	NOTSANDDing	D' = NOT S AND D
5	NoOPing	D' = D
6	XORing	D' = S XOR D
7	ORing	D' = S OR D
8	NORing	D' = NOT S AND NOT D
9	NXORing	D' = NOT S XOR D
10	NOTing	D' = NOT D
11	SORNOTDing	D' = S OR NOT D
12	NOTSing	D' = NOT S
13	NOTSORDing	D' = NOT S OR D

14	NANDing	D' = NOT S OR NOT D
15	BLACKing	D' = 1

The *CopyBits* function is called with a pointer to a RasterOp structure which contains the parameters to the operation, as defined in the include file:

```
struct RasterOp {
        struct Form destForm;       /* destination Form into which bits will be stored */
        struct Form sourceForm;     /* a Form from which bits will be copies */
        struct Form maskForm;       /* a Form containing a halftone pattern */
        int rule;                   /* (least significant 4 bits used) specifies the
                                      Boolean rule for combining corresponding bits of
                                      the source and destination */
        int destX;                  /* X and Y coordinates of top left corner of rect- */
        int destY;                    angular subregion to be filled in destination form */
        int width;                  /* width and height of rectangular subregion in
        int height;                   destination form to be filled */
        int clipX;                  /* Top left corner, width, and height of a
        int clipY;                    rectangular area which restricts
        int ClipWidth;                the affected region
        int clipHeight;               of the destination form */
        int sourceX;                /* Top left corner of the rectangular subregion
        int sourceY;                  to be copied from the source form */
};
```

Parameter-passing efficiency is gained by using only a single pointer to a structure containing the needed parameters for each *CopyBits* operation. In addition, the state held in the RasterOp structure allows multiple operations in a related context to be performed without having to repeat all the initialization of parameters. For example, when displaying many lines of text, the destination Form and clipping rectangle will not change from one character operation to another.

FILES

/usr/lib/libgraf.a

SEE ALSO

graf(3X), *Ridge Multi-Window Display Management Guide*

NAME

curses – CRT screen handling and optimization package

SYNTAX

#include <curses.h>
cc [flags] files – **lcurses** [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. 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. To get character-at-a-time input without echoing, (most interactive, screen oriented-programs want this) after calling *initscr()* you should call *"nonl(); cbreak(); noecho();"*

The full curses interface permits manipulation of data structures called *windows* which can be thought of as two dimensional arrays of characters representing all or part of a CRT screen. A default window called **stdscr** is supplied, and others can be created with **newwin**. Windows are referred to by variables declared "WINDOW *", the type WINDOW is defined in curses.h to be a C structure. These data structures are manipulated with functions described below, among which the most basic are **move**, and **addch**. (More general versions of these functions are included with names beginning with 'w', allowing you to specify a window. The routines not beginning with 'w' affect **stdscr**.) Then *refresh()* is called, telling the routines to make the users CRT screen look like **stdscr**.

If the environment variable TERMINFO is defined, any program using curses will check for a local terminal definition before checking in the standard place. For example, if the standard place is **/usr/lib/terminfo**, and TERM is set to "vt100", then normally the compiled file is found in **/usr/lib/terminfo/v/vt100**. (The "v" is copied from the first letter of "vt100" to avoid creation of huge directories.) However, if TERMINFO is set to **/usr/mark/myterms**, curses will first check **/opusr/mark/myterms/v/vt100**, and if that fails, will then check **/usr/lib/terminfo/v/vt100**. This is useful for developing experimental definitions or when write permission in **/usr/lib/terminfo** is not available.

SEE ALSO

terminfo(4).

FUNCTIONS

Routines listed here may be called when using the full curses.

addch(ch)	add a character to *stdscr* (like putchar) (wraps to next line at end of line)
addstr(str)	calls addch with each character in *str*
attroff(attrs)	turn off attributes named
attron(attrs)	turn on attributes named
attrset(attrs)	set current attributes to *attrs*
baudrate()	current terminal speed
beep()	sound beep on terminal
box(win, vert, hor)	draw a box around edges of *win* *vert* and *hor* are chars to use for vert. and hor. edges of box
clear()	clear *stdscr*
clearok(win, bf)	clear screen before next redraw of *win*
clrtobot()	clear to bottom of *stdscr*
clrtoeol()	clear to end of line on *stdscr*
cbreak()	set cbreak mode
delay_output(ms)	insert ms millisecond pause in output

delch( )	delete a character
deleteln( )	delete a line
delwin(win)	delete *win*
doupdate( )	update screen from all wnooutrefresh
echo( )	set echo mode
endwin( )	end window modes
erase( )	erase *stdscr*
erasechar( )	return user's erase character
fixterm( )	restore tty to "in curses" state
flash( )	flash screen or beep
flushinp( )	throw away any typeahead
getch( )	get a char from tty
getstr(str)	get a string through *stdscr*
gettmode( )	establish current tty modes
getyx(win, y, x)	get (y, x) co-ordinates
has_ic( )	true if terminal can do insert character
has_il( )	true if terminal can do insert line
idlok(win, bf)	use terminal's insert/delete line if bf != 0
inch( )	get char at current (y, x) co-ordinates
initscr( )	initialize screens
insch(c)	insert a char
insertln( )	insert a line
intrflush(win, bf)	interrupts flush output if bf is TRUE
keypad(win, bf)	enable keypad input
killchar( )	return current user's kill character
leaveok(win, flag)	OK to leave cursor anywhere after refresh if flag!=0 for *win*, otherwise cursor must be left at current position.
longname( )	return verbose name of terminal
meta(win, flag)	allow meta characters on input if flag != 0
move(y, x)	move to (y, x) on *stdscr*
mvaddch(y, x, ch)	move(y, x) then addch(ch)
mvaddstr(y, x, str)	similar...
mvcur(oldrow, oldcol, newrow, newcol)	low level cursor motion
mvdelch(y, x)	like delch, but move(y, x) first
mvgetch(y, x)	etc.
mvgetstr(y, x)	
mvinch(y, x)	
mvinsch(y, x, c)	
mvprintw(y, x, fmt, args)	
mvscanw(y, x, fmt, args)	
mvwaddch(win, y, x, ch)	
mvwaddstr(win, y, x, str)	
mvwdelch(win, y, x)	
mvwgetch(win, y, x)	
mvwgetstr(win, y, x)	
mvwin(win, by, bx)	
mvwinch(win, y, x)	
mvwinsch(win, y, x, c)	
mvwprintw(win, y, x, fmt, args)	
mvwscanw(win, y, x, fmt, args)	

newpad(nlines, ncols)	create a new pad with given dimensions
newterm(type, fd)	set up new terminal of given type to output on fd
newwin(lines, cols, begin_y, begin_x)	create a new window
nl( )	set newline mapping
nocbreak( )	unset cbreak mode
nodelay(win, bf)	enable nodelay input mode through getch
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
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	like prefresh but with no output until doupdate called
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	refresh from pad starting with given upper left corner of pad with output to given portion of screen
printw(fmt, arg1, arg2, ...)	printf on *stdscr*
raw( )	set raw mode
refresh( )	make current screen look like *stdscr*
resetterm( )	set tty modes to "out of curses" state
resetty( )	reset tty flags to stored value
saveterm( )	save current modes as "in curses" state
savetty( )	store current tty flags
scanw(fmt, arg1, arg2, ...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win, flag)	allow terminal to scroll if flag != 0
set_term(new)	now talk to terminal new
setscrreg(t, b)	set user scrolling region to lines t through b
setterm(type)	establish terminal with given type
setupterm(term, filenum, errret)	
standend( )	clear standout mode attribute
standout( )	set standout mode attribute
subwin(win, lines, cols, begin_y, begin_x)	create a subwindow
touchwin(win)	change all of *win*
traceoff( )	turn off debugging trace output
traceon( )	turn on debugging trace output
typeahead(fd)	use file descriptor fd to check typeahead
unctrl(ch)	printable version of *ch*
waddch(win, ch)	add char to *win*
waddstr(win, str)	add string to *win*
wattroff(win, attrs)	turn off *attrs* in *win*
wattron(win, attrs)	turn on *attrs* in *win*
wattrset(win, attrs)	set attrs in *win* to *attrs*
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*
wnoutrefresh(win)	refresh but no screen output
wprintw(win, fmt, arg1, arg2, ...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win, fmt, arg1, arg2, ...)	scanf through *win*
wsetscrreg(win, t, b)	set scrolling region of *win*
wstandend(win)	clear standout attribute in *win*
wstandout(win)	set standout attribute in *win*

TERMINFO LEVEL ROUTINES

These routines should be called by programs wishing to deal directly with the terminfo database. Due to the low level of this interface, it is discouraged. Initially, *setupterm* should be called. This will define the set of terminal dependent variables defined in terminfo(4). The include files <curses.h> and <term.h> should be included to get the definitions for these strings, numbers, and flags. Parmeterized strings should be passed through *tparm* to instantiate them. All terminfo strings (including the output of tparm) should be printed with *tputs* or *putp* . Before exiting, *resetterm* should be called to restore the tty modes. (Programs desiring shell escapes or suspending with control Z can call *resetterm* before the shell is called and *fixterm* after returning from the shell.)

fixterm( )	restore tty modes for terminfo use (called by setupterm)
resetterm( )	reset tty modes to state before program entry
setupterm(term, fd, rc)	read in database. Terminal type is the character string *term*, all output is to UNIX System file descriptor *fd*. A status value is returned in the integer pointed to by *rc*: 1 is normal. The simplest call would be *setupterm(0, 1, 0)* which uses all defaults.
tparm(str, p1, p2, ..., p9)	instantiate string str with parms p_i.
tputs(str, affcnt, putc)	apply padding info to string *str*. *affcnt* is the number of lines affected, or 1 if not applicable. *Putc* is a putchar-like function to which the characters are passed, one at a time.
putp(str)	handy function that calls tputs (str, 1, putchar)
vidputs(attrs, putc)	output the string to put terminal in video attribute mode *attrs*, which is any combination of the attributes listed below. Chars are passed to putchar-like function *putc*.

vidattr(attrs)	Like vidputs but outputs through putchar

TERMCAP COMPATIBILITY ROUTINES

These routines were included as a conversion aid for programs that use termcap. Their parameters are the same as for termcap. They are emulated using the *terminfo* database. They may go away at a later date.

tgetent(bp, name)	look up termcap entry for name
tgetflag(id)	get boolean entry for id
tgetnum(id)	get numeric entry for id
tgetstr(id, area)	get string entry for id
tgoto(cap, col, row)	apply parms to given cap
tputs(cap, affcnt, fn)	apply padding to cap calling fn as putchar

ATTRIBUTES

The following video attributes can be passed to the functions *attron, attroff, attrset.*

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_BLANK	Blanking (invisible)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set

FUNCTION KEYS

The following function keys might be returned by *getch* if *keypad* has been enabled. Note that not all of these are currently supported, due to lack of definitions in *terminfo* or the terminal not transmitting a unique code when the key is pressed.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab

KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	soft (partial) reset (unreliable)
KEY_RESET	0531	reset or hard reset (unreliable)
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left)

WARNING

The plotting library *plot*(3X) and the curses library *curses*(3X) both use the names erase( ) and move( ). The curses versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3X) code, and/or #undef move( ) and erase( ) in the *plot*(3X) code.

NAME

graf – low-level graphics library for Ridge Monochrome Display

SYNTAX

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

int g_init (screenForm)
struct Form *screenForm;

int g_readpixel (x, y)
int x, y;

g_writepixel (x, y, color)
int x, y, color;

g_drawline (x0, y0, x1, y1)
int x0, y0, x1, y1;

g_flush ()

g_clbm ()

g_clscreen ()

g_pencolor (color)
int color;

g_penwidth (width)
int width;
```

DESCRIPTION

The **graf** library allows low-level graphics access to the Ridge Monochrome Display *disp*(7). These functions are used to create graphic images in the calling process' own address space, a part of which is mapped to the display screen as described in the *Ridge Multi-Window Display Management Guide*.

These functions operate directly on the display bitmap of the process, and provide no synchronization with other processes which may also be updating the display screen, and thus may interfere with manipulations of the screen by the multi-window display management software.

The *g_init* function must be called once before the other functions to allow internal data structures to be initialized. An *ioctl*(2) system call is made using the open file descriptor 2 *(stderr)* to determine the size of the screen and the starting address of the display bitmap. These values are returned in the structure pointed to by *screenForm,* as defined in the include file:

```
struct Form {
        int width;
        int height;
        unsigned int *bits;
};
```

The *height* and *width* of the screen are measured in pixels, and the address in virtual memory of the display bitmap is placed in the *bits* field. A non-zero value returned by *g_init* indicates an error, such as not having file descriptor 2 bound to a Ridge monochrome display.

The value returned by *g_readpixel* is the content of the bit found at the location specified by *x* and *y*. The address is relative to the top left corner of the screen, which is addressed as 0,0. The value returned is either Black or White, as defined in the include file; the value -1 is returned if the address is out of range for the screen.

The content of the bit found at the location specified by *x* and *y* is set to the given *color* by *g_writepixel*. The colors Black and White set the bit to the value as defined in the include file, while any other color value will cause the bit to be complemented.

G_drawline draws the pixels that correspond to a straight line segment from the point specified as *x0,y0* to the point specified as *x1,y1*. Display device coordinates are used, and the line is clipped to the edges of the screen. The default color of lines is Black, and the width is one pixel, but either of these attributes may be changed as described below. For efficiency, specialized algorithms are used for the common cases of horizontal or vertical lines, or lines that are one pixel wide.

The function *g_flush* is called to flush the display bitmap in main memory to the display controller refresh memory. Any pages in the display bitmap that have been modified will be written to the refresh memory, causing the screen image to be updated.

No change to the screen image occurs until the bitmap is flushed, so this function may be used to get the effect of "double-buffering", where the screen displays the previous image while a new image is being created in the bitmap. *G_clbm* clears the display bitmap to White, in preparation for further drawing. *G_clscreen* clears the screen by calling *g_clbm* followed by *g_flush*.

G_pencolor sets the color to be used for line-drawing according to *color*. The color value determines whether black or white pixels are used by *g_drawline*. The values Black (the initial color) and White are defined in the include file, while any other value will cause subsequent lines to be drawn using pixels which are the complement of the current background.

G_penwidth sets the width in pixels of lines according to *width*. The initial width is one, but any value greater than zero may be specified, and will be used for all subsequent lines drawn by *g_drawline*.

FILES

/usr/lib/libgraf.a

SEE ALSO

Ridge Multi-Window Display Management Guide
CopyBits(3X)

NAME

logname – return login name of user

SYNTAX

char *logname( )

DESCRIPTION

Logname returns a pointer to the null-terminated login name; it extracts the $LOGNAME variable from the user's environment.

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

FILES

/etc/profile

SEE ALSO

env(1), login(1), profile(4), environ(5).

BUGS

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

This method of determining a login name is subject to forgery.

NAME

plot – graphics interface subroutines

SYNTAX

openpl ()

erase ()

label (s)
char *s;

line (x1, y1, x2, y2)
int x1, y1, x2, y2;

circle (x, y, r)
int x, y, r;

arc (x, y, x0, y0, x1, y1)
int x, y, x0, y0, x1, y1;

move (x, y)
int x, y;

cont (x, y)
int x, y;

point (x, y)
int x, y;

linemod (s)
char *s;

space (x0, y0, x1, y1)
int x0, y0, x1, y1;

closepl ()

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. *Space* must be used before any of these functions to declare the amount of space necessary. See *plot*(4). *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

Circle draws a circle of radius *r* whose center is *(x,y)*.

Arc draws an arc of the circle whose center is *(x, y)*; the arc is drawn clockwise from *(x0, y0)* to *(x1, y1)*.

String arguments to *label* and *linemod* are terminated by nulls and do not contain new-lines.

See *plot*(4) for a description of the effect of the remaining functions.

The library files listed below provide several flavors of these routines.

FILES

/usr/lib/libplot.a	produces output for *tplot*(1G) filters
/usr/lib/lib300.a	for DASI 300
/usr/lib/lib300s.a	for DASI 300s
/usr/lib/lib450.a	for DASI 450
/usr/lib/lib4014.a	for Tektronix 4014
/usr/lib/libr15.a	for Ridge Monochrome Display

WARNINGS

In order to compile a program containing these functions in *file.c* it is necessary to use "cc *file.c* – lplot".

In order to execute it, it is necessary to use "a.out |tplot".

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

SEE ALSO
graph(1G), stat(1G), tplot(1G), plot(4).

NAME

regcmp, regex – compile and execute regular expression

SYNTAX

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

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

extern char *loc1;

DESCRIPTION

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

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *loc1* points to where the match began. *Regcmp* and *regex* were mostly borrowed from the editor, *ed*(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ] *.^ These symbols retain their 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), 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

wgraf – graphics library for Ridge Tek4014 windows

SYNTAX

#include <sys/graf.h>

int DrawInit (windowForm)
struct Form *windowForm;

DrawLine (x0, y0, x1, y1)
int x0, y0, x1, y1;

WriteScreen ()

ClearScreen ()

ClearBitmap ()

SetPenColor (newColor)
int newColor;

SetPenWidth (width)
int width;

DESCRIPTION

The **wgraf** library provides a simple interface for drawing lines in a window on the Ridge Monochrome Display. The calling process' control window must be in Tektronix 4014tm mode; these functions generate the appropriate Tektronix 4014 compatible graphics character sequences. Some of the functions interface directly to the display management software and allow control over several graphical attributes of the window.

The coordinate system used for specifying lines has 4096 points in the X direction, and 3120 points in the Y direction, with the origin in the lower left corner. This corresponds to the visible area on a Tektronix 4014 graphics terminal, which is then scaled to the actual size of the window measured in Ridge display device coordinates.

DrawInit must be called once before the other functions. A new file descriptor is opened using the name **/dev/tty**, which is used by the other functions to interact with the window. The actual size of the window is returned in the structure pointed to by *windowForm*. The width and height of the window are measured in Ridge display device coordinates, and the bits field is set to 0. A non-zero value returned by *DrawInit* indicates an error, such as using a control terminal that is not a window on the Ridge display.

DrawLine is used to draw a line from the point specified as *x0,y0* to the point specified as *x1,y1*. The line is clipped to the Tektronix 4014 visible display area coordinate system, and the proper character sequence for graphics mode output is generated. The output is buffered by *DrawLine*, and not actually written until either the buffer fills, or one of the other functions described below is called.

WriteScreen flushes the graphics output buffer to the window, and causes the display management software to update the display screen.

ClearScreen sends the ESC FF sequence, which causes the window to be cleared and the graphics cursor placed at the home position.

ClearBitmap clears the bitmap for the window, but does not cause the screen image to be updated. This allows a new image to be generated in the bitmap for the window while the previous image is still being displayed on the screen.

SetPenColor sets the color to be used for line-drawing in the window according to *newColor*. The color value determines whether black or white pixels are used to draw all lines for the window on the Ridge monochrome display. The values Black (the default) and White are defined in the include file, while any other value will cause subsequent lines to be drawn using pixels which are the complement of the current window background. The color is a window attribute, and remains in effect until explicitly changed by this function.

SetPenWidth sets the width in pixels of lines drawn in the window according to *width*. The default value is one, but any value greater than zero may be specified, and will be used for all subsequent lines drawn in the window. The line width is a window attribute, and remains in effect until explicitly changed by this function.

FILES

/usr/lib/libwgraf.a

SEE ALSO

Ridge Multi-Window Display Management Guide
settek(1), graf(3X), windows(3X), disp(7), mouse(7)

NAME

windows – window function library

SYNTAX

```
#include <sys/graf.h>
#include <sys/winctrl.h>

int GetWindowNumber (fileDesc)
int fileDesc;

int GetCurrentWindow ()

int GetNextWindow (wID)
int wID;

GetWindowFrame (wID, frame)
int wID;
struct Rectangle *frame;

SetWindowFrame (wID, frame)
int wID;
struct Rectangle *frame;

int FindWindow (at)
struct Point *at;

char *GetTitle (wID)
int wID;

SetTitle (wID, name)
int wID;
char *name;

int GetWFlags (wID)
int wID;

SetWFlags (wID, flags)
int wID;
int flags;

ShowWindow (wID)
int wID;

EraseWindow (wID)
int wID;

SelectWindow (wID)
int wID;

UnderWindow (wID)
int wID;

SetCtrlWindow (wID)
int wID;

KillWindow (wID)
int wID;

int DefineFont (name)
char *name;

char *GetFontName (fontID)
int fontID;
```

int GetFontID (wID)
int wID;

SetFont (wID, fontID)
int wID;
int fontID;

GetWCharSize (wID, charSize)
int wID;
struct Point *charSize;

SetCursorLocation (wID, at)
int wID;
struct Point *at;

int PopupMenu (text, button)
char *text;
int button;

CompFrame (frame)
struct Rectangle *frame;

DESCRIPTION

The **windows** subroutine library allows access to the various window attributes and operations supported by the Ridge display management software. These functions perform the appropriate *ioctl(2)* call for each operation, using file descriptor 2 *(stderr)* which should be bound to a window on the Ridge display. Functions that return an integer use the value -1 to indicate an error.

GetWindowNumber returns the window ID of the window associated with the open file descriptor *filedesc*. Each window is assigned a window ID when it is created; this integer is used to identify the window in all subsequent control operations applied to it. A value of -1 indicates that the file descriptor is not associated with a window on the display. *GetCurrentWindow* returns the window ID of the currently selected active input window, which is always at the front of the depth-sorted window list. *GetNextWindow* returns the window ID of the window which is next on the window list behind *wID*. The value -1 is returned if there are no more windows farther back in the list.

GetWindowFrame returns the current location and size of the rectangular frame for *wID*, while *SetWindowFrame* allows the location to be moved on the screen, or the size to be changed. The *frame* argument is defined as a pointer to a Rectangle structure, whose origin is the upper left corner, and extent is the width and height. The Point structure is a two-dimensional location specified in terms of display device coordinates. The <*sys/graf.h*> header file contains these definitions:

```
struct Point {
        int x;
        int y;
};

struct Rectangle {
        struct Point origin;
        struct Point extent;
};
```

FindWindow returns the window ID of the frontmost window that contains the point *at*, which is specified using display device coordinates. If a window can be found that contains the point either within its frame or title tab, then its window ID is returned; otherwise, the value -1 is returned.

GetTitle returns a pointer to a static area containing a null-terminated string, which should be copied elsewhere by the caller. This string is used as the text for displaying the title tab for the window *wID*, and is initialized to the window part of the pathname that was used to create the window. *SetTitle* changes the title string of the window *wID* to the null-terminated string pointed to by *name*. Changing the title string does not change the name the window is accessed by; the pathname of the window remains the same as when it was created. Changing a title to the null string will cause the title tab to disappear.

GetWFlags returns the window flags associated with the window *wID*, while *SetWFlags* sets the window flags for *wID* to the bit values specified in *flags*. The flag bits control various attributes of the window, as defined in the *<sys/winctrl.h>* header file:

WFMode	mode bit field
WFASCII	ASCII mode
WFANSIX3_64	ANSI X3.64 (VT-100) mode
WFTek4014	Tektronix 4014 mode
WFAwake	window is "awake" allowing I/O
WFCurson	text input cursor is shown
WFRetainGraf	window retains graphics
EMQueueKB	queue keyboard events
EMSigIOKB	signal I/O from keyboard
EMQueueLoc	queue locator (mouse) events
EMLocCoords	window or screen relative locator coords
EMLocMotion	only button changes, or any motion
EMButtonMask	button bits
EMRightButton	right button
EMMiddleButton	middle button
EMLeftButton	left button
EMOtherButtons	other buttons
EMSigIOLoc	signal I/O from locator

ShowWindow causes the window identified by *wID* to be redrawn on the screen, while *EraseWindow* causes the window *wID* to be erased by repainting its area and title tab using the gray background pattern. These two functions are not normally used since window updates are performed automatically by the display management software when windows are created, modified, or destroyed.

SelectWindow causes the window identified by *wID* to become the currently active input window, moving it to the front of the depth-sorted window list. The selected window is displayed in front of all other windows, and its title tab is highlighted. All keyboard and mouse input events are directed to the currently selected window. *UnderWindow* moves the window *wID* behind all awake windows in the depth-sorted window list. This causes the new frontmost window in the list to become the currently selected window.

SetCtrlWindow changes the window which acts as the control terminal for the calling process to become *wID*. The control window for a process is inherited from its parent process when it is created. The signal SIGHUP is sent to all processes which have *wID* as their control window by the function *KillWindow*.

DefineFont makes the font file specified as the full pathname *name* known to the display management software. The value returned is a font ID that is used in further operations on that font; the value -1 is returned if the file cannot be read, or if it does not contain valid font information. *GetFontName* returns a pointer to a null-terminated string that is the pathname corresponding to *fontID*. The string is placed in a static area which should be copied to another area by the caller.

GetFontID returns the current font ID associated with the window *wID*, while *SetFont* sets the window font for *wID* to the font specified as *fontID*. *GetWCharSize* returns the size of a single character in the current bit-matrix font associated with window *wID*. The size is returned in the Point structure pointed to by *charSize*, and is measured in pixels. This may be used to calculate the number of lines and columns of text that can be displayed based on the window's size.

SetCursorLocation sets the current cursor location relative to the window *wID*. The coordinate system used for the X,Y position pointed to by *at* depends on the emulation mode of the window, and its event mode bits in the flags word

If the EMLocCoords bit is set, then display device coordinates are used, and the location can be set to any value. If the EMLocCoords bit is not set, then the coordinate system is translated relative to the location of the window on the display surface, and may be scaled depending on the current mode of the window.

If the window is emulating a Tektronix 4014 terminal, then the X coordinates range from 0 to 1024, and the Y coordinates range from 0 to 780, with the origin in the lower left corner of the window. The display device coordinates of the cursor are appropriately scaled based on the size of the window.

If the window is not in Tektronix 4014 mode, then display device coordinates are used, but are first translated relative to the origin of the window. The X coordinates range from 0 to one less than the width of the window, and the Y coordinates range from 0 to one less than the height of the window, with the origin in the upper left corner of the window.

PopupMenu invokes a pop-up menu at the current location of the cursor. The null-terminated string of text pointed to by *text* contains several short lines separated by Newline characters. The *button* parameter contains a bit mask in the lowest three bits, with the least significant bit representing the right button, the middle bit the middle button, and the most significant bit the left button. The text is shown on the screen, and selection is allowed, while a button is depressed which has the corresponding bit set to 1. When the button is released, the selected line's index (counting from one) is returned. If the button is released outside of the menu, the value zero is returned.

CompFrame outlines the Rectangle pointed to by *frame*, drawing the outline once using the complement of the background, updating the screen, and then drawing the outline again using the complement, effectively restoring the original bitmap. The non-destructive outline is typically used repeatedly to draw a rectangle that tracks the cursor. The location and size are specified using display device coordinates.

FILES

/usr/lib/libwindows.a

SEE ALSO

Ridge Multi-Window Display Management Guide
setfont(1), settek(1), setx3.64(1), wgraf(3X), font(4), disp(7), mouse(7)

BLANK

NAME

intro – introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys**.

NAME

a.out – assembler and link editor output

SYNTAX

#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 Ridge is:

```
/*
 * Header prepended to each a.out file.
 */
struct exec {
        unsigned long   a_magic;        /* magic number */
        unsigned short r_brofsett;      /* offset for previous magic */
        unsigned short a_type;          /* type field  */
        unsigned long   a_version;      /* version field */
        unsigned long   a_text;         /* size of text segment */
        unsigned long   a_data;         /* size of initialized data */
        unsigned long   a_bss;          /* size of uninitialized data */
        unsigned long   a_syms;         /* size of symbol table */
        unsigned long   a_entry;        /* entry point */
        unsigned long   a_trsize;       /* size of text relocation */
        unsigned long   a_drsize;       /* size of data relocation */
};

#define RMAGIC  0x9b000000                      /* Ridge magic */
#define RMAGIC1 0x9b010000                      /* Ridge magic - shared code & data */
#define NEWHDR  (sizeof (struct exec))          /* header size */

/*
 * Macros which take exec structures as arguments and tell whether
 * the file has a reasonable magic number or offsets to text, symbols, or strings.
 * A check has been added to make sure that the header is an exec struct.
 */
#define N_BADMAG(x) .
        ((((x).a_magic & 0xff000000) != RMAGIC) |
        ((x).r_brofsett != NEWHDR) |
        ((x).a_type == 0) |
        ((x).a_version == 0))

#define N_TXTOFF(x)          (sizeof (struct exec))
#define N_SYMOFF(x)          (N_TXTOFF(x) + (x).a_text+ (x).a_data + (x).a_trsize+ (x).a_drsize)
#define N_STROFF(x)          (N_SYMOFF(x) + (x).a_syms)
```

The file has five sections: a header, the program text and data, relocation information, a symbol table and a string table (in that order). The last three may be omitted 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. The size of the header is not included in any of the other sizes.

When an a.out file is executed, two physical segments are set up: the code and data segments.

The code segment begins at location 0, and consists of the program header and the text segment from the a.out file. This is the executable code of the program.

The data segment consists of initialized data, unisitialized data (bss), and a runtime stack. Normally, initialized data begins at location 0 of the data segment, and is copied from the a.out file to the data segment when the program begins execution. The initialized data is immediately followed by uninitialized data (bss). Uninitialized data starts out as all zeros. The runtime stack begins at the highest possible location and grows downward.

The stack will occupy the highest possible locations in the data segment: growing downwards. The stack is automatically extended as required. The data/bss area is only extended as requested by *brk*(2).

After the header in the file follow the text, data, text relocation data relocation, symbol table and string table in that order. The text begins immediately after the header. The N_TXTOFF macro returns this absolute file position when given the name of an exec structure as argument. Initialized data is contiguous with the text and immediately followed by the text relocation and then the data relocation information. The symbol table follows all this; its position is computed by the N_SYMOFF macro. Finally, the string table immediately follows the symbol table at a position which can be gotten easily using N_STROFF. The first 4 bytes of the string table are not used for string storage, but rather contain the size of the string table; this size INCLUDES the 4 bytes, the minimum string table size is thus 4.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file **/usr/include/a.out.h** as follows:

```
/*
 * Format of a symbol table entry; this file is included by <a.out.h>
 * and should be used if you aren't interested the a.out header
 * or relocation information.
 */
struct nlist {
        union {
                char    *n_name;        /* for use when in-core */
                long    n_strx;         /* index into file string table */
        } n_un;
        unsigned char n_type;           /* type flag, i.e. N_TEXT etc; see below */
        char    n_other;                /* unused */
        short   n_desc;                 /* see <stab.h> */
unsigned long   n_value;                /* value of this symbol (or dbx offset) */
};
#define         n_hash n_desc  /* used internally by ld */

/*
 * Simple values for n_type.
 */
#define N_UNDF      0x0     /* undefined */
#define N_ABS       0x2     /* absolute */
#define N_TEXT      0x4     /* text */
#define N_DATA      0x6     /* data */
#define N_BSS 0x8   /* bss */
#define N_COMM      0x12    /* common (internal to ld) */
#define N_FN 0x1f   /* file name symbol */
```

```
#define N_EXT       01      /* external bit, or'ed in */
#define N_TYPE      0x1e    /* mask for all the type bits */

/*
 * Dbx entries have some of the N_STAB bits set.
 * These are given in <stab.h>
 */
#define N_STAB  0xe0 /* if any of these bits set, a Dbx entry */

/*
 * Format for namelist values.
 */
#define N_FORMAT   "%08x"
```

In the *a.out* file a symbol's n_un.n_strx field gives an index into the string table. A n_strx value of 0 indicates that no name is associated with a particular symbol table entry. The field n_un.n_name can be used to refer to the symbol name only if the program sets this up using n_strx and appropriate data from the string table.

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 byte in the text or data which is not a portion of a reference to an undefined external symbol is exactly that value which will appear in memory when the file is executed. If a byte in the text or data involves a reference to an undefined external symbol, as indicated by the relocation information, then the value 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 to the bytes in the file.

If relocation information is present, it amounts to eight bytes per relocatable datum as in the following structure:

```
/*
 * Format of a relocation datum.
 */
struct relocation_info {
        int       r_address;        /* address which is relocated */
unsigned int      r_symbolnum:24,         /* local symbol ordinal */
                  r_pcrel:1,        /* was relocated pc relative already */
                  r_length:2,       /* 0=byte, 1=word, 2=long */
                  r_extern:1,       /* does not include value of sym referenced */
                  r_relocate:4;     /* 1 => ridge specific load or laddr, fixup */
};
```

There is no relocation information if a_trsize+a_drsize==0. If r_extern is 0, then r_symbolnum is actually a n_type for the relocation (i.e. N_TEXT meaning relative to segment text origin.)

SEE ALSO

as(1), ld(1), nm(1), dbx(1), stab(4), strip(1), map(1)

NAME

ar – archive (library) file format

SYNTAX

#include <ar.h>

DESCRIPTION

The archive command *ar* combines several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld.*

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

```
#define ARMAG   "!<arch>\n"
#define SARMAG 8
#define ARFMAG  "`\n"
struct   arf_hdr {   /* archive file member header */
         char        arf_name[16];/* file member name */
         char        arf_date[12];/* file member date */
         char        arf_uid[6];/* file member user identification */
         char        arf_gid[6];/* file member group identification */
         char        arf_mode[8];/* file member mode */
         char        arf_size[10];/* file member size */
         char        arf_fmag[2];/*              */
};
```

The name is a blank-padded string. The *ar_fmag* field contains ARFMAG to help verify the presence of a header. The other fields are left-adjusted, blank-padded numbers. They are decimal except for *ar_mode,* which is octal. The date is the modification date of the file at the time of its insertion into the archive.

Each file begins on an even (0 mod 2) boundary; a new-line is inserted between files if necessary. Nevertheless, the size given reflects the actual size of the file exclusive of padding. There is no provision for empty areas in an archive file.

The encoding of the header is the same on UNIX System V release 2, bsd 4.1, and bsd 4.2. If an archive contains printable files, the archive itself is printable.

SEE ALSO

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

BUGS

File names lose trailing blanks. Most software dealing with archives takes even an included blank as a name terminator.

NAME

badblocks - media defect list for hard discs

DESCRIPTION

Badblocks is the only file on the "badblocks" floppy disc shipped with each 445-Mb disc.

Winchester-type hard discs (including all Ridge discs), come from the factory with a list of their own manufacturing defects. For the 60- or 142-Mb disc in the Ridge 32C cabinet, the bad block table is permanently encoded on the disc and printed on paper in a plastic pocket that stays with the unit. For the 445-Mb disc unit in the companion cabinet, and the 80- or 180-Mb disc in the Ridge 32S, the bad block information is distributed on a separate floppy disc.

The **badblocks** file on the **badblocks** floppy disc describes the cylinder, surface (or head), byte offset from the disc index mark, and length in bits of each defect.

On ROS 3.1 and subsequent releases, the defect list is kept on the floppy in the following format:

cyl-num head-num byte-offset bit-length

example:

```
 12  4  2359  16
104  7 12304   3
 66  5  3412   2
```

In the sad event that the system must be rebuilt, the the 445-Mb badblocks floppy must be available.

NAME

Conf – Device Configuration File

DESCRIPTION

/ros/conf is the device configuration file which is read at system startup time to automatically configure the device boards.

Each board in the system is assigned a unique physical identifier; each type of board has a unique type identifier. For example, the tape drive has a device type equal to 32 (20 hex).

At system initialization time, all the device types are found by reading all the boards in the system.

Entries in the configuration file contain a device type name and an associated device driver program file name. Each field is delimited by a ':' and each entry appears on a separate line.

```
device type : file name

For example: :1:/drivers/fdlp
```

When the system is started, the system detects the installed boards and searches the **conf** file for an entry of its type. If found, the associated file is assumed to be a device driver and as loaded and started. There is a one-to-one relationship between the boards and the conf entries. If there are multiple boards of the same type, there must exist multiple entries to start a driver for each board.

EXAMPLE

The following device boards are present in the system:

board	*qty*	*type*
FDLP	2	1
Tape	1	32
Ether	1	48

```
file:   :1:/drivers/fdlp
        :32:/drivers/td
        :48:/drivers/ether
        :49:/drivers/colordisp
```

Files corresponding to device type 1, 32, 48 are loaded. Even though there are two FDLP boards, only one is started. No driver is started for device type 49 (Ridge color display) because the board is not present in the system.

With the load enable switch set (warm-boot, cold-boot) the device types for each device in the system is listed.

A complete list of the devices types is presented below:

Ridge I/O Device assigments:

Device Type
decimal hex *Device Description*

0	0	
1	1	Floppy Disk
2	2	ANSI Disk controller
3	3	SMD disk controller
4	4	Ridge Monochrome Display
5	5	Ridge Monochrome Disp Keyboard
6	6 }	
.	. }	--> Ridge Devices (reserved)
31	1F}	
32	20	Tape controller
21	2F	Ridge devices (reserved)
48	30	DR11 to ethernet
49	31	DR11 keyboard
50	32	DR11 to Color Display
51	33-3B	Ridge DR11 devices (reserved)
51	3C-3F	Customer DR11 devices (available)
51	40-DF	Ridge devices (reserved)
51	E0-EF	Customer devices (available)
51	F0-FF	Ridge devices (reserved)

NAME

cpio – format of cpio archive

DESCRIPTION

The *header* structure, when the – **c** option of *cpio*(1) is not used, is:

```
1 1 l.  struct {         short   h_magic,                   h_dev;          ushort  h_ino,
                h_mode,                          h_uid,                               h_gid;
        short   h_nlink,                      h_rdev,                           h_mtime[2],
                h_namesize,                                                   h_filesize[2];
        char    h_name[h_namesize rounded to word]; } Hdr;
```

When the – **c** option is used, the *header* information is described by:

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%11lo%s",
        &Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
        &Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
        &Longtime, &Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file are recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

SEE ALSO

cpio(1), find(1), stat(2B).

NOTES

For reasons of compatibility, the inode (file designator), device, and rdevice fields are kept as short integers. These fields are actually kept as 32-bit integers within the ROS system.

NAME

dir – format of directories

SYNTAX

#include <sys/dir.h>

DESCRIPTION

A directory is like an ordinary file, except that no user may write into a one. A directory is distinguished from a file by a bit in the flag word of its file table entry (see *fs*(4)). The structure of a directory entry (as given in **/usr/include/sys/dir.h**) is:

```
#ifndef DIRSIZ
#define DIRSIZ 16
#endif
struct direct
{
      vers_t d_vers;
      ino_t d_ino;
      char d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are "." for the directory itself, and ".." for the parent directory. Because the root directory ("/") has no parent, ".." has the same meaning as "." in that case.

The *ls(1)* command with the **-al** option lists the "." file.

FILES

/usr/include/sys/dir.h

SEE ALSO

fs(4).

NAME

font – format of Ridge bit-matrix fonts

SYNTAX

#include <sys/font.h>

DESCRIPTION

A Ridge bit-matrix font file contains a header and a sequence of pixel patterns or "glyphs" corresponding to the images of individual characters. The file also has a table of indices into the bit-matrix patterns for each character in the font.

The structure of a font file is given in the include file, where the last two fields are actually variable length arrays of 32-bit words:

```
struct Font {
        int format;
        int min;
        int max;
        int maxWidth;
        int length;
        int ascent;
        int descent;
        int xOffset;
        int formWidth;
        int glyphs[1];  /* variable length */
        int xTable[1];  /* variable length */
};
```

The *format* field contains a magic number which identifies the file as containing bit-matrix font information. The value for this field is given by the following define in the include file:

```
#define FontMagic 0xABCDDCBA
```

The *min* field contains the smallest legal ASCII value for this font. Characters less than this value will be displayed using the glyph for the illegal character (indexed by *max+ 1).* The *max* field contains the largest legal ASCII value for this font. There should always be a glyph for each character from *min* to *max+ 1,* where the extra glyph is used for displaying characters that fall outside the range.

The *maxWidth* field contains the width in bits of the widest character in the font. The height in bits above (and including) the baseline is kept in *ascent,* while the height in bits below the baseline is kept in *descent.* All characters in a font have the same bit-matrix height (ascent + descent), but may vary in width. The bit-matrix for each character should include appropriate white space around each character so that bit-matrices may be placed side-by-side in both the horizontal and vertical directions. The *xOffset* field is normally zero, but can be negative for a font with ligatures. The offset value is added to the normal width of each character before advancing to the next character in a string being printed horizontally.

The *glyphs* array of 32-bit words contains the actual bits for all the characters in a form suitable for the *CopyBits*(3) operation. The array is of variable length, composed of a number of rasters equal to the font's height, with each raster measuring *formWidth* bits long (rounded up to the next integral 32-bit word). Each raster corresponds to a row slice through the bit-matrices of all the characters packed together horizontally. Thus, *formWidth* is the sum of all the characters' widths in bits.

The *xTable* array of words is of variable length and is indexed by a character's ASCII value, justified to index from *min* through *max+ 2.* The contents of each entry is the left X (measured in bits) within the glyphs for the selected character. The entries are ordered in increasing value, and thus the difference between two adjacent entries represents the width of the character indexed by the smaller indice. The *max+ 1* entry is the left X for the "illegal" character glyph, and the *max+ 2* entry contains the value one greater than the right X of the "illegal" character glyph.

The ***length*** field contains the number of 32-bit words in the font following this field. This value can be computed as:

```
7 + (max - min) +
(((formWidth + 31) / 32) * (ascent + descent))
```

FILES

/usr/include/sys/font.h

SEE ALSO

setfont(1), windows(3X), disp(7)

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the UNIX System with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX System commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets **<:** and **:>**. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a – followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a – followed by the name of a "canned" tab specification.

Standard tabs are specified by **t– 8**, or equivalently, **t1,9,17,25,**etc. The canned tabs which are recognized are defined by the *tabs*(1) command.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin* The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t– 8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

```
* <:t5,10,15 s72:> *
```

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

Several UNIX System commands correctly interpret the format specification for a file. Among them is *gath* (see *send*(1C)) which may be used to convert files to a standard format acceptable to other UNIX System commands.

SEE ALSO

ed(1), newform(1), send(1C), tabs(1).

NAME

gettydefs – speed and terminal settings used by getty

DESCRIPTION

The **/etc/gettydefs** file contains information used by *getty(1)* to set up the speed and terminal settings for a line. It supplies information on what the *login* prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a <*break*> character.

Each entry in **/etc/gettydefs** has the following format:

```
label# initial-flags # final-flags # login-prompt #next-label
```

Lines that begin with # are ignored and may be used to comment on the file. Each entry, including comment entries, must be followed by a blank line. The various fields can contain quoted characters of the form **\b, \n, \c,**, etc., as well as \nnn, where *nnn* is the octal value of the desired character. The various fields are:

label This is the string to which *getty* tries to match its second argument. It is often the speed, such as **1200**, at which the terminal is supposed to run, but it needn't be (see below).

initial-flags
These flags are the initial *ioctl* (2) settings to which the terminal is to be set if a terminal type is not specified to *getty*. *Getty* understands the symbolic names specified in **/usr/include/sys/termio.h** (see *termio* (7)). Normally, only the speed flag is requried in the *initial-flags*. *Getty* automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty* executes *login*(1).

final-flags These flags assume the values of *initial flags* and are set just before *getty* executes *login*. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL**, so that the line is hung up on the final close.

login-prompt
This entire field is printed as the *login-prompt*. Unlike the above fields where white space is ignored (a space, tab, or new-line), they are included in the *login-prompt* field.

next-label This indicates the next *label* of the entry in the table that *getty* should use if the user types a <*break*> or the input cannot be read. Usually, a series of speeds are linked together in this fashion, into a closed set. For instance, **2400** linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

If *getty* is called without a second argument, then the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. It is also used if *getty* can't find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command which will bring up a terminal at **9600** baud.

It is strongly recommended that after making or modifying /fB/etc/gettydefs, it be run through *getty* with the check option to be sure there are no errors.

FILES

/etc/gettydefs

NAME

group – group file

DESCRIPTION

/etc/group allows a password to be associated with user groups, and records which users are allowed to execute newgrp(1) to gain access to each group. (Groups are established by assigning group numbers to users in the **/etc/passwd** file.) **/etc/group** is an ascii file for associating passwords with groups. It contains the following information for each group:

name : encrypted-password : group-ID-num : list-of-users

The fields are separated by colons. Additional group entries appear on separate lines.

Name is the group-name used in newgrp(1) and chgrp(1).

Encrypted-password should not be used. If the password field is null, no password is demanded.

Group-ID-num corresponds to the field of the same name in the **/etc/passwd** file.

List-of-users is a list of user ID names, separated by commas, who are allowed to use newgrp(1) to enter this group.

This file has general read permission so that the encrypted passwords can be read. It can be used, for example, to map numerical group ID's to names.

EXAMPLE

```
root::0:root
other::1:
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
adm::4:root,adm,daemon
mail::6:root
rje::8:rje,shqer
daemon::12:root,daemon
cad::13:bill,ron,kevin
cam::14:glenn,dan
aok::15:bruce,carol,cheryl
```

FILES

/etc/group
/etc/passwd

SEE ALSO

chgrp(1), newgrp(1), passwd(1), passwd(4)

NAME

hosts – host name data base

DESCRIPTION

The *hosts* file contains information regarding the known hosts on the DARPA Internet. For each host a single line should be present with the following information:

official host name
Internet address
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official host data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional "." notation using the *inet_addr*() routine from the Internet address manipulation library, *inet*(3N). Host names may contain any printable character other than a field delimiter, newline, or comment character.

EXAMPLE

```
#
# local hosts database

126.1   machine1 manufacturing mailroom
126.2   machine2 marketing guardhouse
126.3   machine3 gardencenter
```

FILES

/etc/hosts

NAME

inittab – script for the startup process

DESCRIPTION

The *inittab* file supplies the script to startup's role as a general process dispatcher. The process that constitutes the majority of *startup* process dispatching activities is the line process */etc/getty* that initiates individual terminal lines. Other processes typically dispatched by *startup* are daemons and the shell.

The *inittab* file is composed of entries that are position dependent and have the following format:

```
id:rstate:action:process
```

Each entry appears on a separate line. Backslash \ in the first character position indicates a continuation line. Up to 512 characters per entry are permitted. Comments may be inserted in the *process* field using the *sh* (1) convention for comments. Comments for lines that spawn *gettys* are displayed by the *who* (1) command. It is expected that they will contain some information about the line such as the location. There are no limits (other than maximum entry size) imposed on the number of entries within the *inittab* file. The entry fields are:

id This is one to four characters used to uniquely identify an entry.

rstate

THIS FIELD IS CURRENTLY IGNORED. For process scheduling, this field defines the *run-level* at which this entry is to be processed. Each process spawned by *startup* is assigned one or more *run-levels*, ranging from 0 to 6, which determine the the machine states under which that process can run. If the system is operating in run-level 1, for example, only the processes having a "1" in the *rstate* field will be processed. When *startup* is requested to change run-levels, all processes wich do not have an entry in the *rstate* field for the target *run-level* will be sent the warning signel (**SIGTERM**) and allowed a 20 second grace period before being forcibly terminated by a kill signal (**SIGKILL**). The *rstate* field can define multiple *run-levels* for a process by selecting more than one *run-level* in any combination from **0-6**. If no *run-level* is specified, then *action* will be taken on this *process* for all *run-levels* **0-6**. The *state* field can also contain **a, b,** *or* **c**. Entries with these characters are processed only when the *telstartup* process requests them to be run, regardless of the current system run-level. They differ from run-levels in that the system is only in these states for as long as it takes to execute all the entries associated with the states. A process started by an **a, b,** or **c** command is not killed when *startup* changes levels. They are only killed when *startup* changes levels. They are only killed if their line in **/etc/inittab** is marked **off** in the *action* field, their line is deleted entirely from **/etc/inittab**, or *startup* goes into the *SINGLE USER* state.

action Keywords in this field tell *startup* how to treat the process specified in the *process* field. The actions recognized by *startup* are currently limited to the following:

respawn

If the process does not exist, start the process, do not wait for its termination (continue scanning the *inittab* file), and when it dies restart the process. If the process currently exists then do nothing and continue scanning the *inittab* file.

off

If the process associated with this entry is currently running, send the warning signal (**SIGTERM**) and wait 20 seconds before forcibly terminating the process via the fill signal (**SIGKILL**). If the process is nonexistent, ignore the entry.

process

This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh** -c 'exec *command'*. For this reason, any legal *sh* syntax can appear in the *process* field. Comments can be inserted with the ;#*comment* syntax. Only entries with **/etc/getty** or **/etc/mount** in this field are currently recognized.

FILES

/etc/inittab

SEE ALSO

getty(1), sh(1), who(1), exec(2B), open(2B), signal(2B)

NAME

issue – issue identification file

DESCRIPTION

The file **/etc/issue** contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file which is read by program *getty* and then written to any terminal spawned or respawned from the *inittab* file.

FILES

/etc/issue
/etc/inittab

SEE ALSO

login(1).

NAME

mnttab – mounted file system table

SYNTAX

#include <mnttab.h>

DESCRIPTION

Mnttab resides in directory **/etc** and contains a table of devices, mounted by the *mount*(1) command, in the following structure as defined by **<mnttab.h>**:

```
struct  mnttab {
        char    mt_dev[10];
        char    mt_filsys[10];
        short   mt_ro_flg;
        time_t  mt_time;
};
```

Each entry is 26 bytes in length; the first 10 bytes are the null-padded name of the place where the *special file* is mounted; the next 10 bytes represent the null-padded root name of the mounted special file; the remaining 6 bytes contain the mounted *special file*'s read/write permissions and the date on which it was mounted.

The maximum number of entries in *mnttab* is based on the system parameter NMOUNT located in **/usr/src/uts/cf/conf.c**, which defines the number of allowable mounted special files.

SEE ALSO

mount(1), setmnt(1).

NAME

.netrc – login profile for network users

DESCRIPTION

.netrc is a login profile for users of **ftp**(1).

.netrc is located in the user's home directory ($HOME), similar to **.profile**. **.netrc** is normally read upon using the **ftp** "open" command, in conjunction with the **ftp** automatic login feature.

The file consists of one or more entries, formatted as follows:

machine *machinename* **login** *loginname* [**password** *password*]

A user may have one such entry for each machine on which he wants automatic login. If the **password** *password* fields exist in the entry, the **.netrc** file must not have "read access" for the group and other users. The access mode must be **-rw-------**.

If the user has no **.netrc** file, he will be prompted for a login name and password upon attempting to connect to a remote system (via the **ftp** "open" command).

FILES

$HOME/.netrc

SEE ALSO

ftp(1), hosts(4)

NAME

networks – network name data base

DESCRIPTION

The *networks* file contains information regarding the known networks which comprise the DARPA Internet. For each network a single line should be present with the following information:

official network name
network number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates a comment line. This file is normally created from the official network data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown networks.

Network number may be specified in the conventional "." notation using the *inet_network*() routine from the Internet address manipulation library, *inet*(3N). Network names may contain any printable character other than a field delimiter, newline, or comment character.

EXAMPLE

```
#
# network table
#

mynet  126     ours
```

FILES

/etc/networks

NAME

passwd – password file

DESCRIPTION

Passwd contains for each user the following information:

login-name
encrypted-password
user-ID-num
group-ID-num
optional field
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. Only the first eight characters of the *login-name* are significant. The optional field typically contains the user's full name, but may contain any information, or be empty. 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 numbers to names.

The encrypted password consists of 13 characters chosen from a 64 character alphabet (**.**, **/**, **0**– **9**, **A**– **Z**, **a**– **z**), except when the password is null in which case the encrypted password is also null. Password aging is effected for a particular user if his encrypted password in the password file is followed by a comma and a non-null string of characters from the above alphabet. (Such a string must be introduced in the first instance by the super-user.)

The first character of the age, *M* say, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after his password has expired will be forced to supply a new one. The next character, *m* say, denotes the minimum period in weeks which must expire before the password may be changed. The remaining characters define the week (counted from the beginning of 1970) when the password was last changed. (A null string is equivalent to zero.) *M* and *m* have numerical values in the range 0– 63 that correspond to the 64 character alphabet shown above (i.e. **/** = 1 week; **z** = 63 weeks). If $m = M = 0$ (derived from the string **.** or **..**) the user will be forced to change his password the next time he logs in (and the "age" will disappear from his entry in the password file). If $m > M$ (signified, e.g., by the string **./**) only the super-user will be able to change the password.

FILES

/etc/passwd

SEE ALSO

login(1), passwd(1), a64l(3C), crypt(3C), getpwent(3C), group(4).

NAME

plot – graphics interface

DESCRIPTION

Files of this format are produced by routines described in *plot*(3X) and are interpreted for various devices by commands described in *tplot*(1G). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by four bytes representing the **x** and **y** values; each value is a signed integer. The last designated point in an **l, m, n,** or **p** instruction becomes the "current point" for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in *plot*(3X).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes. See *tplot*(1G).

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a new-line.

e erase: Start another frame of output.

f linemod: Take the following string, up to a new-line, as the style for drawing further lines. The styles are "dotted", "solid", "longdashed", "shortdashed", and "dotdashed". Effective only for the **– T4014** and **– Tver** options of *tplot*(1G) (Tektronix 4014 terminal and Versatec plotter).

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *tplot*(1G). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points outside may be displayable on devices whose face is not square.

DASI 300	space(0, 0, 4096, 4096);
DASI 300s	space(0, 0, 4096, 4096);
DASI 450	space(0, 0, 4096, 4096);
Tektronix 4014	space(0, 0, 3120, 3120);
Versatec plotter	space(0, 0, 2048, 2048);
Monochrome disp	space(0, 0, 1024, 800);

SEE ALSO

graph(1G), tplot(1G), plot(3X), term(5).

NAME

profile – setting up an environment at login time

DESCRIPTION

If your login directory contains a file named **.profile**, that file will be executed (via the shell's **exec .profile**) before your session begins; **.profiles** are handy for setting exported environment variables and terminal modes. If the file **/etc/profile** exists, it will be executed for every user before the **.profile**. The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 22
# Tell me when new mail comes in
MAIL=/usr/mail/myname
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
# Set terminal type
echo "terminal: \c"
read TERM
case $TERM in
        300)        stty cr2 nl0 tabs; tabs;;
        300s)       stty cr2 nl0 tabs; tabs;;
        450)        stty cr2 nl0 tabs; tabs;;
        hp)         stty cr0 nl0 tabs; tabs;;
        745|735)    stty cr1 nl1 – tabs; TERM=745;;
        43)         stty cr1 nl0 – tabs;;
        4014|tek)   stty cr0 nl0 – tabs ff1; TERM=4014; echo "\33;";;
        *)          echo "$TERM unknown";;
esac
```

FILES

$HOME/.profile
/etc/profile

SEE ALSO

env(1), login(1), mail(1), sh(1), stty(1), su(1), environ(5), term(5).

NAME

protocols – protocol name data base

DESCRIPTION

The *protocols* file contains information regarding the known protocols used in the DARPA Internet. For each protocol a single line should be present with the following information:

official protocol name
protocol number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates a comment line.

Protocol names may contain any printable character other than a field delimiter, newline, or comment character.

EXAMPLE

```
#
# Internet protocols
#

ip      0       IP
icmp    1       ICMP
tcp     6       TCP
udp     17      UDP
```

FILES

/etc/protocols

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the ***checksum***, the ***delta table*** (contains information about each delta), ***user names*** (contains login names and/or numerical group IDs of users who may add deltas), ***flags*** (contains definitions of internal keywords), ***comments*** (contains arbitrary descriptive information about the file), and the ***body*** (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the **ASCII SOH** (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form DDDDD represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters, except those of the first line. The @**h** provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
 .
 .
 .
@c <comments> ...
 .
 .
 .
@e
```

The first line (@**s**) contains the number of lines inserted/deleted/unchanged respectively. The second line (@**d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The @**i**, @**x**, and @**g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The @**m** lines (optional) each contain one MR number associated with the delta; the @**c** lines contain comments associated with the delta.

The @**e** line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @**u** and @**U**. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see *admin*(1) for more information on their use). Each flag line takes the form:

```
@f <flag>      <optional text>
```

The following flags are defined:

```
@f t    <type of program>
@f v    <program name>
@f i
@f b
@f m    <module name>
@f f    <floor>
@f c    <ceiling>
@f d    <default-sid>
@f n
@f j
@f l    <lock-releases>
@f q    <user defined>
@f z    <reserved for use in interfaces>
```

The **t** flag defines the replacement for the %Y% identification keyword. The **v** flag controls prompting for MR numbers in addition to comments; if the optional text is present it defines an MR number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **– b** keyletter may be used on the *get* command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the %M% identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing (*get*(1) with the **– e** keyletter). The **q** flag defines the replacement for the %Q% identification keyword. **z** flag is used in certain specialized interface programs.

Comments

Arbitrary text surrounded by the bracketing lines @**t** and @**T**. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert, delete,* and *end,* represented by:

```
@I DDDDD
@D DDDDD
@E DDDDD
```

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).
Source Code Control System User's Guide in the *Interactive Access to ROS* manual.

NAME

services – service name data base

DESCRIPTION

The *services* file contains information regarding the known services available in the DARPA Internet. For each service a single line should be present with the following information:

official service name
port number
protocol name
aliases

Items are separated by any number of blanks and/or tab characters. The port number and protocol name are considered a single *item*; a "/" is used to separate the port and protocol (e.g. "512/tcp"). A "#" indicates a comment line.

Service names may contain any printable character other than a field delimiter, newline, or comment character.

EXAMPLE

```
#
# services table: map service names to ports/protocol
#

telnet  23/tcp  TELNET
ftp     21/tcp  FTP
```

FILES

/etc/services

NAME

stab – symbol table types

SYNTAX

#include <stab.h>

DESCRIPTION

Stab.h defines some values of the n_type field of the symbol table of a.out files. These are the types for permanent symbols (i.e. not local labels, etc.) used by the **dbx** debugger. Symbol table entries can be produced by the *stabs* assembler directive. This allows one to specify a double-quote delimited name, a symbol type, one char and one short of information about the symbol, and an unsigned long (usually an address). To avoid having to produce an explicit label for the address field, the *stabd* directive can be used to implicitly address the current location. If no name is needed, symbol table entries can be generated using the *stabn* directive. The loader promises to preserve the order of symbol table entries produced by *stab* directives. As described in *a.out(4)*, an element of the symbol table consists of the following structure:

```
/*
 * Format of a symbol table entry.
 */
struct nlist {
        union {
                char  *n_name; /* for use when in-core */
                long  n_strx;   /* index into file string table */
        } n_un;
        unsigned char n_type;   /* type flag */
        char          n_other;  /* unused */
        short         n_desc;   /* see struct desc, below */
        unsigned n_value;       /* address or offset or line */
};
```

The low bits of the n_type field are used to place a symbol into at most one segment, according to the following masks, defined in <*a.out.h*>. A symbol can be in none of these segments by having none of these segment bits set.

```
/*
 * Simple values for n_type.
 */
#define N_UNDF   0x0   /* undefined */
#define N_ABS    0x2   /* absolute */
#define N_TEXT   0x4   /* text */
#define N_DATA   0x6   /* data */
#define N_BSS    0x8   /* bss */

#define N_EXT    01    /* external bit, or'ed in */
```

The n_value field of a symbol is relocated by the linker, *ld*(1) as an address within the appropriate segment. N_value fields of symbols not in any segment are unchanged by the linker. In addition, the linker will discard certain symbols, according to rules of its own, unless the n_type field has one of the following bits set:

```
/*
 * Other permanent symbol table entries have some of the N_STAB bits set.
 * These are given in <stab.h>
 */
#define N_STAB          0xe0/* if any of these bits set, don't discard */
```

This allows up to 112 (7 * 16) symbol types, split between the various segments. Some of these have already been claimed. The **dbx** symbolic debugger uses the following n_type values:

```
/* This file gives definitions supplementing <a.out.h>
 * for permanent symbol table entries.
 * These must have one of the N_STAB bits on,
 * and are subject to relocation according to the masks in <a.out.h>.
 *
 * for symbolic debugger, dbx(1):
 */
#defineN_GSYM      0x20        /* global symbol: name,,0,type,0 */
#defineN_FNAME     0x22        /* procedure name (f77 kludge): name,,0 */
#defineN_FUN       0x24        /* procedure: name,,0,linenumber,address */
#defineN_STSYM     0x26        /* static symbol: name,,0,type,address */
#defineN_LCSYM     0x28        /* .lcomm symbol: name,,0,type,address */
#defineN_RSYM      0x40        /* register sym: name,,0,type,register */
#defineN_SLINE     0x44        /* src line: 0,,0,linenumber,address */
#defineN_SSYM      0x60        /* structure elt: name,,0,type,struct_offset */
#defineN_SO        0x64        /* source file name: name,,0,0,address */
#defineN_LSYM      0x80        /* local sym: name,,0,type,offset */
#defineN_SOL       0x84        /* #included file name: name,,0,0,address */
#defineN_PSYM      0xa0        /* parameter: name,,0,type,offset */
#defineN_ENTRY     0xa4        /* alternate entry: name,linenumber,address */
#defineN_LBRAC     0xc0        /* left bracket: 0,,0,nesting level,address */
#defineN_RBRAC     0xe0        /* right bracket: 0,,0,nesting level,address */
#defineN_BCOMM     0xe2        /* begin common: name,, */
#defineN_ECOMM     0xe4        /* end common: name,, */
#defineN_ECOML     0xe8        /* end common (local name): ,,address */
#defineN_LENG      0xfe        /* second stab entry with length information */
/*
 * for the berkeley pascal compiler, pc(1):
 */
#defineN_PC  0x30          /* global pascal symbol: name,,0,subtype,line */
```

Dbx uses the n_desc field to hold a type specifier in the form used by the Portable C Compiler, *cc*(1), in which a base type is qualified in the following structure:

```
struct desc {
        short  q6:2,
               q5:2,
               q4:2,
               q3:2,
               q2:2,
               q1:2,
               basic:4;
};
```

There are four qualifications, with q1 the most significant and q6 the least significant:

0	none
1	pointer
2	function
3	array

The sixteen basic types are assigned as follows:

0	undefined

1	function argument
2	character
3	short
4	int
5	long
6	float
7	double
8	structure
9	union
10	enumeration
11	member of enumeration
12	unsigned character
13	unsigned short
14	unsigned int
15	unsigned long

SEE ALSO

as(1), ld(1), dbx(1), a.out(4)

BLANK

NAME

term – format of compiled term file.

SYNTAX

term

DESCRIPTION

Compiled terminfo descriptions are placed under the directory **/usr/lib/terminfo**. In order to avoid a linear search of a huge UNIX system directory, a two-level scheme is used: **/usr/lib/terminfo/c/name** where *name* is the name of the terminal, and *c* is the first character of *name*. Thus, *act4* can be found in the file **/usr/lib/terminfo/a/act4**. Synonyms for the same terminal are implemented by multiple links to the same compiled file.

The format has been chosen so that it will be the same on all hardware. An 8 or more bit byte is assumed, but no assumptions about byte ordering or sign extension are made.

The compiled file is created with the *compile* program, and read by the routine *setupterm*. Both of these pieces of software are part of *curses*(3X). The file is divided into six parts: the header, terminal names, boolean flags, numbers, strings, and string table.

The header section begins the file. This section contains six short integers in the format described below. These integers are (1) the magic number (octal 0432); (2) the size, in bytes, of the names section; (3) the number of bytes in the boolean section; (4) the number of short integers in the numbers section; (5) the number of offsets (short integers) in the strings section; (6) the size, in bytes, of the string table.

Short integers are stored in two 8-bit bytes. The first byte contains the least significant 8 bits of the value, and the second byte contains the most significant 8 bits. (Thus, the value represented is 256*second+ first.) The value – 1 is represented by 0377, 0377, other negative value are illegal. The – 1 generally means that a capability is missing from this terminal. Note that this format corresponds to the hardware of the VAX and PDP-11. Machines where this does not correspond to the hardware read the integers as two bytes and compute the result.

The terminal names section comes next. It contains the first line of the terminfo description, listing the various names for the terminal, separated by the '| character. The section is terminated with an ASCII NUL character.

The boolean flags have one byte for each flag. This byte is either 0 or 1 as the flag is present or absent. The capabilities are in the same order as the file <term.h>.

Between the boolean section and the number section, a null byte will be inserted, if necessary, to ensure that the number section begins on an even byte. All short integers are aligned on a short word boundary.

The numbers section is similar to the flags section. Each capability takes up two bytes, and is stored as a short integer. If the value represented is – 1, the capability is taken to be missing.

The strings section is also similar. Each capability is stored as a short integer, in the format above. A value of – 1 means the capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table. Special characters in ^X or \c notation are stored in their interpreted form, not the printing representation. Padding information $<nn> and parameter information %x are stored intact in uninterpreted form.

The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each string is null terminated.

Note that it is possible for *setupterm* to expect a different set of capabilities than are actually present in the file. Either the database may have been updated since *setupterm* has been recompiled (resulting in extra unrecognized entries in the file) or the program may have been recompiled more recently than the database was updated (resulting in missing entries). The routine *setupterm* must be prepared for both possibilities – this is why the numbers and sizes are included. Also, new capabilities must always be added at the end of the lists of boolean, number, and string capabilities.

As an example, an octal dump of the description for the Microterm ACT 4 is included:

```
microterm|act4|microterm act iv,
   cr=^M, cud1=^J, ind=^J, bel=^G, am, cub1=^H,
   ed=^_, el=^^, clear=^L, cup=^T%p1%c%p2%c,
   cols#80, lines#24, cuf1=^X, cuu1=^Z, home=^],
```

```
000 032 001      \0 025  \0  \b  \0 212  \0   "  \0   m   i   c   r
020   o   t   e   r   m   |   a   c   t   4   |   m   i   c   r   o
040   t   e   r   m       a   c   t       i   v  \0  \0 001  \0  \0
060  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
100  \0  \0   P  \0 377 377 030  \0 377 377 377 377 377 377 377 377
120 377 377 377 377  \0  \0 002  \0 377 377 377 377 004  \0 006  \0
140  \b  \0 377 377 377 377  \n  \0 026  \0 030  \0 377 377 032  \0
160 377 377 377 377 034  \0 377 377 036  \0 377 377 377 377 377 377
200 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
520 377 377 377 377      \0 377 377 377 377 377 377 377 377 377 377
540 377 377 377 377 377 377 007  \0  \r  \0  \f  \0 036  \0 037  \0
560 024   %   p   1   %   c   %   p   2   %   c  \0  \n  \0 035  \0
600  \b  \0 030  \0 032  \0  \n  \0
```

Some limitations: total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

FILES

/usr/lib/terminfo/*/* compiled terminal capability data base

SEE ALSO

curses(3X), terminfo(4).

NAME

terminfo – terminal capability data base

SYNTAX

/usr/lib/terminfo/*/*

DESCRIPTION

Terminfo is a directory of files that describe terminal capabilities. The capabilities are read by programs like vi(1) and curses(3x). The files list the capabilities and how operations are performed, and the padding requirements and initialization sequences of various terminal devices.

Terminfo entries consist of fields separated by ','. White space after each ',' is ignored.

The first entry for a terminal contains the names by which the device is known, each separated by a '|' character. The first name in the first entry is the most most common abbreviation for the terminal; the next names are synonyms for the terminal name; the last name fully identifies the terminal. All names but the last should be in lowercase and contain no blanks; the last name may contain uppercase and blanks.

Terminal names (except for the last, verbose name) are selected using the following conventions: The particular piece of hardware making up the terminal should have a root name chosen, thus "hp2621". This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. Thus, a vt100 in 132 column mode would be vt100-w. The following suffixes should be used where possible:

Suffix	Meaning	Example
-w	Wide mode (more than 80 columns)	vt100-w
-am	With auto. margins (usually default)	vt100-am
-nam	Without automatic margins	vt100-nam
-*n*	Number of lines on the screen	aaa-60
-na	No arrow keys (leave them in local)	c100-na
-*n*p	Number of pages of memory	c100-4p
-rv	Reverse video	c100-rv

CAPABILITIES

The variable is the name by which the programmer (at the terminfo level) accesses the capability. The capname is the short name used in the text of the database, and is used by a person updating the database. The i.code is the two-letter internal code used in the compiled database, and always corresponds to the old **termcap** capability name.

Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short and to allow the tabs in the source file **caps** to line up nicely. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64-1979 standard. Semantics are also intended to match those of the specification.

(P)	indicates that padding may be specified
(G)	indicates that the string is passed through tparm, with parms as given ($\#_i$).
(*)	indicates that padding may be based on the number of lines affected
($\#_i$)	indicates the i^{th} parameter.

Variable	Cap-name	I. Code	Description
Booleans			
auto_left_margin,	bw	bw	cub1 wraps from column 0 to last column
auto_right_margin,	am	am	Terminal has automatic margins
beehive_glitch,	xsb	xb	Beehive (f1=escape, f2=ctrl C)
ceol_standout_glitch,	xhp	xs	Standout not erased by overwriting (hp)
eat_newline_glitch,	xenl	xn	newline ignored after 80 cols (Concept)
erase_overstrike,	eo	eo	Can erase overstrikes with a blank
generic_type,	gn	gn	Generic line type (e.g.,, dialup, switch).
hard_copy,	hc	hc	Hardcopy terminal
has_meta_key,	km	km	Has a meta key (shift, sets parity bit)
has_status_line,	hs	hs	Has extra "status line"
insert_null_glitch,	in	in	Insert mode distinguishes nulls
memory_above,	da	da	Display may be retained above the screen
memory_below,	db	db	Display may be retained below the screen
move_insert_mode,	mir	mi	Safe to move while in insert mode
move_standout_mode,	msgr	ms	Safe to move in standout modes
over_strike,	os	os	Terminal overstrikes
status_line_esc_ok,	eslok	es	Escape can be used on the status line
teleray_glitch,	xt	xt	Tabs ruin, magic so char (Teleray 1061)
tilde_glitch,	hz	hz	Hazeltine; can not print ~'s
transparent_underline,	ul	ul	underline character overstrikes
xon_xoff,	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns,	cols	co	Number of columns in a line
init_tabs,	it	it	Tabs initially every # spaces
lines,	lines	li	Number of lines on screen or page
lines_of_memory,	lm	lm	Lines of memory if > lines. 0 means varies
magic_cookie_glitch,	xmc	sg	Number of blank chars left by smso or rmso
padding_baud_rate,	pb	pb	Lowest baud where cr/nl padding is needed
virtual_terminal,	vt	vt	Virtual terminal number (UNIX system)
width_status_line,	wsl	ws	No. columns in status line
Strings:			
back_tab,	cbt	bt	Back tab (P)
bell,	bel	bl	Audible signal (bell) (P)
carriage_return,	cr	cr	Carriage return (P*)
change_scroll_region,	csr	cs	change to lines #1 through #2 (vt100) (PG)
clear_all_tabs,	tbc	ct	Clear all tab stops (P)
clear_screen,	clear	cl	Clear screen and home cursor (P*)
clr_eol,	el	ce	Clear to end of line (P)
clr_eos,	ed	cd	Clear to end of display (P*)
column_address,	hpa	ch	Set cursor column (PG)
command_character,	cmdch	CC	Term. settable cmd char in prototype
cursor_address,	cup	cm	Screen rel. cursor motion row #1 col #2 (PG)
cursor_down,	cud1	do	Down one line
cursor_home,	home	ho	Home cursor (if no cup)
cursor_invisible,	civis	vi	Make cursor invisible

cursor_left,	cub1	le	Move cursor left one space
cursor_mem_address,	mrcup	CM	Memory relative cursor addressing
cursor_normal,	cnorm	ve	Make cursor appear normal (undo vs/vi)
cursor_right,	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll,	ll	ll	Last line, first column (if no cup)
cursor_up,	cuu1	up	Upline (cursor up)
cursor_visible,	cvvis	vs	Make cursor very visible
delete_character,	dch1	dc	Delete character (P*)
delete_line,	dl1	dl	Delete line (P*)
dis_status_line,	dsl	ds	Disable status line
down_half_line,	hd	hd	Half-line down (forward 1/2 linefeed)
enter_alt_charset_mode,	smacs	as	Start alternate character set (P)
enter_blink_mode,	blink	mb	Turn on blinking
enter_bold_mode,	bold	md	Turn on bold (extra bright) mode
enter_ca_mode,	smcup	ti	String to begin programs that use cup
enter_delete_mode,	smdc	dm	Delete mode (enter)
enter_dim_mode,	dim	mh	Turn on half-bright mode
enter_insert_mode,	smir	im	Insert mode (enter);
enter_protected_mode,	prot	mp	Turn on protected mode
enter_reverse_mode,	rev	mr	Turn on reverse video mode
enter_secure_mode,	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode,	smso	so	Begin stand out mode
enter_underline_mode,	smul	us	Start underscore mode
erase_chars	ech	ec	Erase #1 characters (PG)
exit_alt_charset_mode,	rmacs	ae	End alternate character set (P)
exit_attribute_mode,	sgr0	me	Turn off all attributes
exit_ca_mode,	rmcup	te	String to end programs that use cup
exit_delete_mode,	rmdc	ed	End delete mode
exit_insert_mode,	rmir	ei	End insert mode
exit_standout_mode,	rmso	se	End stand out mode
exit_underline_mode,	rmul	ue	End underscore mode
flash_screen,	flash	vb	Visible bell (may not move cursor)
form_feed,	ff	ff	Hardcopy terminal page eject (P*)
from_status_line,	fsl	fs	Return from status line
init_1string,	is1	i1	Terminal initialization string
init_2string,	is2	i2	Terminal initialization string
init_3string,	is3	i3	Terminal initialization string
init_file,	if	if	Name of file containing is
insert_character,	ich1	ic	Insert character (P)
insert_line,	il1	al	Add new blank line (P*)
insert_padding,	ip	ip	Insert pad after character inserted (p*)
key_backspace,	kbs	kb	Sent by backspace key
key_catab,	ktbc	ka	Sent by clear-all-tabs key
key_clear,	kclr	kC	Sent by clear screen or erase key
key_ctab,	kctab	kt	Sent by clear-tab key
key_dc,	kdch1	kD	Sent by delete character key
key_dl,	kdl1	kL	Sent by delete line key
key_down,	kcud1	kd	Sent by terminal down arrow key
key_eic,	krmir	kM	Sent by rmir or smir in insert mode
key_eol,	kel	kE	Sent by clear-to-end-of-line key
key_eos,	ked	kS	Sent by clear-to-end-of-screen key
key_f0,	kf0	k0	Sent by function key f0

key_f1,	kf1	k1	Sent by function key f1
key_f10,	kf10	ka	Sent by function key f10
key_f2,	kf2	k2	Sent by function key f2
key_f3,	kf3	k3	Sent by function key f3
key_f4,	kf4	k4	Sent by function key f4
key_f5,	kf5	k5	Sent by function key f5
key_f6,	kf6	k6	Sent by function key f6
key_f7,	kf7	k7	Sent by function key f7
key_f8,	kf8	k8	Sent by function key f8
key_f9,	kf9	k9	Sent by function key f9
key_home,	khome	kh	Sent by home key
key_ic,	kich1	kI	Sent by ins char/enter ins mode key
key_il,	kil1	kA	Sent by insert line
key_left,	kcub1	kl	Sent by terminal left arrow key
key_ll,	kll	kH	Sent by home-down key
key_npage,	knp	kN	Sent by next-page key
key_ppage,	kpp	kP	Sent by previous-page key
key_right,	kcuf1	kr	Sent by terminal right arrow key
key_sf,	kind	kF	Sent by scroll-forward/down key
key_sr,	kri	kR	Sent by scroll-backward/up key
key_stab,	khts	kT	Sent by set-tab key
key_up,	kcuu1	ku	Sent by terminal up arrow key
keypad_local,	rmkx	ke	Out of "keypad transmit" mode
keypad_xmit,	smkx	ks	Put terminal in "keypad transmit" mode
lab_f0,	lf0	l0	Labels on function key f0 if not f0
lab_f1,	lf1	l1	Labels on function key f1 if not f1
lab_f10,	lf10	la	Labels on function key f10 if not f10
lab_f2,	lf2	l2	Labels on function key f2 if not f2
lab_f3,	lf3	l3	Labels on function key f3 if not f3
lab_f4,	lf4	l4	Labels on function key f4 if not f4
lab_f5,	lf5	l5	Labels on function key f5 if not f5
lab_f6,	lf6	l6	Labels on function key f6 if not f6
lab_f7,	lf7	l7	Labels on function key f7 if not f7
lab_f8,	lf8	l8	Labels on function key f8 if not f8
lab_f9,	lf9	l9	Labels on function key f9 if not f9
meta_on,	smm	mm	Turn on "meta mode" (8th bit)
meta_off,	rmm	mo	Turn off "meta mode"
newline,	nel	nw	Newline (behaves like cr followed by lf)
pad_char,	pad	pc	Pad character (rather than null)
parm_dch,	dch	DC	Delete #1 chars (PG*)
parm_delete_line,	dl	DL	Delete #1 lines (PG*)
parm_down_cursor,	cud	DO	Move cursor down #1 lines (PG*)
parm_ich,	ich	IC	Insert #1 blank chars (PG*)
parm_index,	indn	SF	Scroll forward #1 lines (PG)
parm_insert_line,	il	AL	Add #1 new blank lines (PG*)
parm_left_cursor,	cub	LE	Move cursor left #1 spaces (PG)
parm_right_cursor,	cuf	RI	Move cursor right #1 spaces (PG*)
parm_rindex,	rin	SR	Scroll backward #1 lines (PG)
parm_up_cursor,	cuu	UP	Move cursor up #1 lines (PG*)
pkey_key,	pfkey	pk	Prog funct key #1 to type string #2
pkey_local,	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit,	pfx	px	Prog funct key #1 to xmit string #2

print_screen,	mc0	ps	Print contents of the screen
prtr_off,	mc4	pf	Turn off the printer
prtr_on,	mc5	po	Turn on the printer
repeat_char,	rep	rp	Repeat char #1 #2 times. (PG*)
reset_1string,	rs1	r1	Reset terminal completely to sane modes.
reset_2string,	rs2	r2	Reset terminal completely to sane modes.
reset_3string,	rs3	r3	Reset terminal completely to sane modes.
reset_file,	rf	rf	Name of file containing reset string
restore_cursor,	rc	rc	Restore cursor to position of last sc
row_address,	vpa	cv	Vertical position absolute (set row) (PG)
save_cursor,	sc	sc	Save cursor position (P)
scroll_forward,	ind	sf	Scroll text up (P)
scroll_reverse,	ri	sr	Scroll text down (P)
set_attributes,	sgr	sa	Define the video attributes (PG9)
set_tab,	hts	st	Set a tab in all rows, current column
set_window,	wind	wi	Current window is lines #1-#2 cols #3-#4
tab,	ht	ta	Tab to next 8 space hardware tab stop
to_status_line,	tsl	ts	Go to status line, column #1
underline_char,	uc	uc	Underscore one char and move past it
up_half_line,	hu	hu	Half-line up (reverse 1/2 linefeed)
init_prog,	iprog	iP	Path name of program for init
key_a1,	ka1	K1	Upper left of keypad
key_a3,	ka3	K3	Upper right of keypad
key_b2,	kb2	K2	Center of keypad
key_c1,	kc1	K4	Lower left of keypad
key_c3,	kc3	K5	Lower right of keypad
prtr_non,	mc5p	pO	Turn on the printer for #1 bytes

A Sample Entry

The following entry, which describes the Concept– 100, is among the more complex entries in the *terminfo* file as of this writing.

```
concept100 |c100| concept |c104 |c100-4p |concept 100,
        am, bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>, cnorm=\Ew,
        cols#80, cr=^M$<9>, cub1=^H, cud1=^J, cuf1=\E=,
        cup=\Ea%p1%' '%+ %c%p2%' '%+ %c,
        cuu1=\E;, cvvis=\EW, db, dch1=\E^A$<16*>, dim=\EE, dl1=\E^B$<3*>,
        ed=\E^C$<16*>, el=\E^U$<16>, eo, flash=\Ek$<20>\EK, ht=\t$<8>,
        il1=\E^R$<3*>, in, ind=^J, .ind=^J$<9>, ip=$<16*>,
        is2=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200\Eo\47\E,
        kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
        kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
        lines#24, mir, pb#9600, prot=\EI, rep=\Er%p1%c%p2%' '%+ %c$<.2*>,
        rev=\ED, rmcup=\Ev    $<6>\Ep\r\n, rmir=\E\200, rmkx=\Ex,
        rmso=\Ed\Ee, rmul=\Eg, rmul=\Eg, sgr0=\EN\200,
        smcup=\EU\Ev  8p\Ep\r, smir=\E^P, smkx=\EX, smso=\EE\ED,
        smul=\EG, tabs, ul, vt#8, xenl,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Comments may be included on lines beginning with "#". Capabilities in *terminfo* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have mnemonic names. The automatic margin feature of the Concept is called **am**. Numeric capabilities, such as column-width, are indicated by a mnemonic, then the '#' character, then the numeric value, like **col#80** to indicate 80 columns.

String-valued capabilities, such as the character string to achieve a function, are indicated by a mnemonic, then the '=' character, then the string value, like **el=^y** (control-y clears display to end of line). A delay in milliseconds may appear anywhere in such a capability, enclosed in $<..> brackets, as in **el=\EK$<3>**, and padding characters are supplied by *tputs* to provide this delay. The delay can be either a number like '20', or a number followed by an '*' like '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of *lines* affected. This is always one unless the terminal has **xenl** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.)

A number of escape sequences are provided in the string-valued capabilities for easy encoding of characters there. Both **\E** and **\e** map to an ESCAPE character, **^x** maps to a control-x, where x is any character, and the sequences **\n \l \r \t \b \f \s** give a newline, linefeed, return, tab, backspace, formfeed, and space. Other escapes include \^ for ^, \\ for \, \, for comma, \: for :, and \0 for null. (\0 will produce \200, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a \.

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above.

Preparing Descriptions

To write your own description file, copy a similar terminfo file. Build the file piece by piece, testing it with vi(1) as you go. (A very unusual terminal may expose deficiencies in the ability of the terminfo file to describe it, or bugs in vi(1)).

To test a new terminal description, set the environment variable TERMINFO to a pathname of a directory containing the compiled description you are working on. With TERMINFO set, programs like vi(1) will look there rather than in */usr/lib/terminfo*. To get the padding for insert-line correct (if the terminal manufacturer did not document it) a severe test is to edit **/etc/passwd** at 9600 baud, delete about 16 lines from the middle of the screen, then hit the 'u' key several times quickly. If the terminal craps out, more padding is usually needed. A similar test can be used for insert-character.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **lines** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (**os** applies to storage scope terminals, such as TEKTRONIX 4010 series, as well as hard copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc) give this as **bel**.

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1**. Similarly, codes to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**. These local cursor motions should not alter the text they pass over, for example, you would not normally use '**cuf1**= ' because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a CRT terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe teletypestyle and screen-type terminals. Thus, the model 33 teletype is described as

```
33|tty33|tty|model 33 teletype,
bel=^G, cols#72, cr=^M, cud1=^J, hc, ind=^J, os,
```

while the Lear Siegler ADM- 3 is described as

```
adm3|3|lsi adm3,
am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H, cud1=^J,
ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with printf(3S)-like escapes **%x** in it. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The parameter mechanism uses a stack and special **%** codes to manipulate it. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary.

The **%**encodings have the following meanings:

%%	outputs '%'
%d	print pop() as in printf
%2d	print pop() like %2d
%3d	print pop() like %3d
%02d	

%03d	as in printf
%c	print pop() gives %c
%s	print pop() gives %s
%p[1-9]	push ith parm
%P[a-z]	set variable [a-z] to pop()
%g[a-z]	get variable [a-z] and push it
%'c'	char constant c
%{nn}	integer constant nn
%+ %- %* %/ %m	
	arithmetic (%m is mod): push(pop() op pop())
%& %| %^	bit operations: push(pop() op pop())
%= %> %<	logical operations: push(pop() op pop())
%! %~	unary operations push(op pop())
%i	add 1 to first two parms (for ANSI terminals)
%? expr %t thenpart %e elsepart %;	
	if-then-else, %e elsepart is optional.
	else-if's are possible ala Algol 68:
	%? c_1 %t b_1 %e c_2 %t b_2 %e c_3 %t b_3 %e c_4 %t b_4 %e %;
	c_i are conditions, b_i are bodies.

Binary operations are in postfix form with the operands in the usual order. That is, to get x-5 one would use "%gx%{5}%-".

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cup** capability is cup=6\E&%p2%2dc%p1%2dY.

The Microterm ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary, cup=^T%p1%c%p2%c. Terminals which use %c need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **\n** **^D** and **\r**, as the system may change or discard them. (The library routines dealing with terminfo set tty modes so that tabs are never expanded, so \t is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus cup=\E=%p1%' '%+%c%p2%' '%+%c. After sending '\E=', this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values) and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

If the terminal has row or column absolute cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two parameter sequence (as with the hp2645) and can be used in preference to **cup** . If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup**, such as the TEKTRONIX 4025.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the \EH sequence on HP terminals cannot be used for **home.**)

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el.** If the terminal can clear from the current position to the end of the display, then this should be given as **ed. Ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl.** If the terminal has a settable scrolling region (like the vt100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command – the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind.** The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *terminfo.* The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type abc   def using local cursor motions (not spaces) between the abc and the def. Then position the cursor before the abc and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the abc shifts over to the def which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for insert null. While these are two logically separate attributes (one line vs. multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

Terminfo can describe both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ich1** *n* times.

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

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n*, to delete *n characters*, and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode*, representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If you have a choice, reverse video plus half-bright is good, or reverse video alone.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking) **bold** (bold or extra bright) **dim** (dim or half-bright) **invis** (blanking or invisible text) **prot** (protected) **rev** (reverse video) **sgr0** (turn off *all* attribute modes) **smacs** (enter alternate character set mode) and **rmacs** (exit alternate character set mode). Turning on any of these modes singly may or may not turn off other modes.

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking 9 parameters. Each parameter is either 0 or 1, as the corresponding attribute is on or off. The 9 parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist.

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **flash**; it must not move the cursor.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given which undoes the effects of both of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the TEKTRONIX 4025, where **smcup** sets the command character to be the one used by terminfo.

If 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 **smkx** and **rmkx**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome** respectively. If there are function keys such as f0, f1, ..., f10, the codes they send can be given as **kf0, kf1, ..., kf10**. If these keys have labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10**. The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1**, **ka3**, **kb2**, **kc1**, and **kc3**. These keys are useful when the effects of a 3 by 3 directional pad are needed.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt**. By convention, if the teletype modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by the *tset* command to determine whether to set the mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the terminfo description can assume that they are properly set.

Other capabilities include **is1**, **is2**, and **is3**, initialization strings for the terminal, **iprog**, the path name of a program to be run to initialize the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the terminfo description. They are normally sent to the terminal, by the *tset* program, each time the user logs in. They will be printed in the following order: **is1**; **is2**; setting tabs using **tbc** and **hts**; **if**; running the program **iprog**; and finally **is3**. Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is2** and **if**. These strings are output by the *reset* program, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs2** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the vt100 into 80-column mode would normally be part of **is2**, but it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80 column mode.

If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row). If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in **is2** or **if**.

Delays

Certain capabilities control padding in the teletype driver. These are primarily needed by hard copy terminals, and are used by the *tset* program to set teletype modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** will cause the appropriate delay bits to be set in the teletype driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used.

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit h19's 25th line, or the 24th line of a vt100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings to go to the beginning of the status line and to return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The parameter **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to. If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, tparm(repeat_char, 'x', 10) is the same as 'xxxxxxxxxx'.

If the terminal has a settable command character, such as the TEKTRONIX 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some UNIX systems: The environment is to be searched for a **CC** variable, and if found, all occurrences of the prototype character are replaced with the character in the environment variable.

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*, *dialup*, *patch*, and *network*, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.)

If the terminal uses xon/xoff handshaking for flow control, give **xon**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm#0** indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

If the terminal is one of those supported by the UNIX virtual terminal protocol, the terminal number can be given as **vt**.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **mc5p** takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Strings to program function keys can be given as **pfkey**, **pfloc**, and **pfx**. Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local; and **pfx** causes the string to be transmitted to the computer.

Glitches and Braindamage

Hazeltine terminals, which do not allow '~' characters to be displayed should indicate **hz**.

Terminals which ignore a linefeed immediately after an **am** wrap, such as the Concept and vt100, should indicate **xenl**.

If **el** is required to get rid of standout (instead of merely writing normal text on top of it), **xhp** should be given.

Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This glitch is also taken to mean that it is not possible to position the cursor on top of a "magic cookie", that to erase standout mode it is instead necessary to use delete and insert line.

The Beehive Superbee, which is unable to correctly transmit the escape or control C characters, has **xsb**, indicating that the f1 key is used for escape and f2 for control C. (Only certain Superbees have this problem, depending on the ROM.)

Other specific terminal problems may be corrected by adding more capabilities of the form **x***x*.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be cancelled by placing **xx@** to the left of the capability definition, where xx is the capability. For example, the entry

```
2621-nl, smkx@, rmkx@, use=2621,
```

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

FILES

/usr/lib/terminfo/?/* files containing terminal descriptions

SEE ALSO

tic(1), curses(3X), printf(3S), term(4), term(5).

NAME

utmp, wtmp – utmp and wtmp entry formats

SYNTAX

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

DESCRIPTION

These files, which hold user and accounting information for such commands as *who*(1), *write*(1), and *login*(1), have the following structure as defined by **<utmp.h>**:

```
#define   UTMP_FILE   "/etc/utmp"
#define   WTMP_FILE   "/etc/wtmp"
#define   ut_name     ut_user

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

/* Definitions for ut_type */
#define EMPTY          0
#define RUN_LVL        1
#define BOOT_TIME      2
#define OLD_TIME       3
#define NEW_TIME       4
#define INIT_PROCESS   5
#define LOGIN_PROCESS  6        /* Process spawned by "init" */
#define USER_PROCESS   7        /* A "getty" process waiting for login */
#define DEAD_PROCESS   8        /* A user process */
#define ACCOUNTING     9
#define UTMAXTYPE   ACCOUNTING /* Largest legal value of ut_type */

/* Special strings or formats used in the "ut_line" field when */
/* accounting for something other than a process.          */
/* No string for the ut_line field can be more than 11 chars + */
/* a NULL in length.                                 */
#define RUNLVL_MSG "run-level %c"
#define BOOT_MSG   "system boot"
#define OTIME_MSG  "old time"
#define NTIME_MSG  "new time"
```

FILES

/usr/include/utmp.h

/etc/utmp
/etc/wtmp

SEE ALSO

login(1), who(1), write(1), getut(3C).

NAME

uuencode – format of an encoded uuencode file

DESCRIPTION

Files output by *uuencode(1)* consist of a header line, followed by a number of body lines, and a trailer line. *Uudecode(1)* will ignore any lines preceding the header or following the trailer. Lines preceding a header must not, of course, look like a header.

The header line is distinguished by having the first 6 characters begin . The word *begin* is followed by a mode (in octal), and a string which names the remote file. A space separates the three items in the header line.

The body consists of a number of lines, each at most 62 characters long (including the trailing newline). These consist of a character count, followed by encoded characters, followed by a newline. The character count is a single printing character, and represents an integer, the number of bytes the rest of the line represents. Such integers are always in the range from 0 to 63 and can be determined by subtracting the character space (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a space to make the characters printing. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra garbage will be included to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII space.

The trailer line consists of end on a line by itself.

SEE ALSO

uuencode(1), uudecode(1), uusend(1), uucp(1), mail(1)

BLANK

NAME

intro – introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

NAME

ascii – map of ASCII character set

SYNTAX

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, to be printed as needed. It contains:

```
|000 nul|001 soh|002 stx|003 etx|004 eot|005 enq|006 ack|007 bel|
|010 bs |011 ht |012 nl |013 vt |014 np |015 cr |016 so |017 si |
|020 dle|021 dc1|022 dc2|023 dc3|024 dc4|025 nak|026 syn|027 etb|
|030 can|031 em |032 sub|033 esc|034 fs |035 gs |036 rs |037 us |
|040 sp |041  ! |042  " |043  # |044  $ |045  % |046  & |047  ' |
|050  ( |051  ) |052  * |053  + |054  , |055  - |056  . |057  / |
|060  0 |061  1 |062  2 |063  3 |064  4 |065  5 |066  6 |067  7 |
|070  8 |071  9 |072  : |073  ; |074  < |075  = |076  > |077  ? |
|100  @ |101  A |102  B |103  C |104  D |105  E |106  F |107  G |
|110  H |111  I |112  J |113  K |114  L |115  M |116  N |117  O |
|120  P |121  Q |122  R |123  S |124  T |125  U |126  V |127  W |
|130  X |131  Y |132  Z |133  [ |134  \ |135  ] |136  ^ |137  _ |
|140  ` |141  a |142  b |143  c |144  d |145  e |146  f |147  g |
|150  h |151  i |152  j |153  k |154  l |155  m |156  n |157  o |
|160  p |161  q |162  r |163  s |164  t |165  u |166  v |167  w |
|170  x |171  y |172  z |173  { |174  | |175  } |176  ~ |177 del|

| 00 nul| 01 soh| 02 stx| 03 etx| 04 eot| 05 enq| 06 ack| 07 bel|
| 08 bs | 09 ht | 0a nl | 0b vt | 0c np | 0d cr | 0e so | 0f si |
| 10 dle| 11 dc1| 12 dc2| 13 dc3| 14 dc4| 15 nak| 16 syn| 17 etb|
| 18 can| 19 em | 1a sub| 1b esc| 1c fs | 1d gs | 1e rs | 1f us |
| 20 sp | 21  ! | 22  " | 23  # | 24  $ | 25  % | 26  & | 27  ' |
| 28  ( | 29  ) | 2a  * | 2b  + | 2c  , | 2d  - | 2e  . | 2f  / |
| 30  0 | 31  1 | 32  2 | 33  3 | 34  4 | 35  5 | 36  6 | 37  7 |
| 38  8 | 39  9 | 3a  : | 3b  ; | 3c  < | 3d  = | 3e  > | 3f  ? |
| 40  @ | 41  A | 42  B | 43  C | 44  D | 45  E | 46  F | 47  G |
| 48  H | 49  I | 4a  J | 4b  K | 4c  L | 4d  M | 4e  N | 4f  O |
| 50  P | 51  Q | 52  R | 53  S | 54  T | 55  U | 56  V | 57  W |
| 58  X | 59  Y | 5a  Z | 5b  [ | 5c  \ | 5d  ] | 5e  ^ | 5f  _ |
| 60  ` | 61  a | 62  b | 63  c | 64  d | 65  e | 66  f | 67  g |
| 68  h | 69  i | 6a  j | 6b  k | 6c  l | 6d  m | 6e  n | 6f  o |
| 70  p | 71  q | 72  r | 73  s | 74  t | 75  u | 76  v | 77  w |
| 78  x | 79  y | 7a  z | 7b  { | 7c  | | 7d  } | 7e  ~ | 7f del|
```

FILES

/usr/pub/ascii

NAME

environ – user environment

DESCRIPTION

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

PATH	The sequence of directory prefixes that *sh*(1), *time*(1), *nice*(1), *nohup*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (**:**). *Login*(1) sets PATH=**:/bin:/usr/bin**.
HOME	Name of the user's login directory, set by *login*(1) from the password file *passwd*(4).
TERM	The kind of terminal for which output is to be prepared. This information is used by commands, such as *mm*(1) or *tplot*(1G), which may exploit special capabilities of that terminal.
TZ	Time zone information. The format is **xxx*n*zzz** where **xxx** is standard local time zone abbreviation, *n* is the difference in hours from GMT, and **zzz** is the abbreviation for the daylight-saving local time zone, if any; for example, **EST5EDT**.

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

SEE ALSO

env(1), login(1), sh(1), exec(2B), getenv(3C), profile(4), term(5).

gsize 10

NAME

fcntl – file control options

SYNTAX

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2B) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2B).

```
/* Flag values accessible to open(2B) and fcntl(2B) */
/*  (The first three can only be set by open) */
#define  O_RDONLY 0
#define  O_WRONLY 1
#define  O_RDWR   2
#define  O_NDELAY 04   /* Non-blocking I/O */
#define  O_APPEND 010  /* append (writes guaranteed at the end) */


/* Flag values accessible only to open(2B) */
#define  O_CREAT  00400  /* open with file create (uses*/
                   /* third open arg)*/
#define  O_TRUNC  01000  /* open with truncation */
#define  O_EXCL   02000  /* exclusive open */


/* fcntl(2B) requests */
#define  F_DUPFD  0  /* Duplicate fildes */
#define  F_GETFD  1  /* Get fildes flags */
#define  F_SETFD  2  /* Set fildes flags */
#define  F_GETFL  3  /* Get file flags */
#define  F_SETFL  4  /* Set file flags */
```

SEE ALSO

fcntl(2B), open(2B).

NAME

math – math functions and constants

SYNTAX

#include <math.h>

DESCRIPTION

This file contains declarations of all the functions in the Math Library (described in Section 3M), as well as various functions in the C Library (Section 3C) that return floating-point values.

It defines the structure and constants used by the *matherr*(3M) error-handling mechanisms, including the following constant used as an error-return value:

HUGE	The maximum value of a single-precision floating-point number.

The following mathematical constants are defined for user convenience:

M_E	The base of natural logarithms (*e*).
M_LOG2E	The base-2 logarithm of *e*.
M_LOG10E	The base-10 logarithm of *e*.
M_LN2	The natural logarithm of 2.
M_LN10	The natural logarithm of 10.
M_PI	The ratio of the circumference of a circle to its diameter. (There are also several fractions of its reciprocal and its square root.)
M_SQRT2	The positive square root of 2.
M_SQRT1_2	The positive square root of 1/2.

For the definitions of various machine-dependent "constants," see the description of the <*values.h*> header file.

FILES

/usr/include/math.h

SEE ALSO

intro(3), matherr(3M), values(5).

NAME

rc – command script for demons

SYNTAX

/etc/rc

DESCRIPTION

Rc is a script of commands executed when the system is booted or rebooted. It can start all system demons, preserve editor files, and/or clear the scratch directory **/tmp.** On a reboot, **rc** executes before login procedures take place.

The first command in the **/etc/rc** file often defines the machine's name, using hostname(1). Other commands typically invoke the cron demon, then the network demons like ftpd(1).

NAME

regexp – regular expression compile and match routines

SYNTAX

#define INIT <declarations>
#define GETC( ) <getc code>
#define PEEKC( ) <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile(instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;

int step(string, expbuf)
char *string, *expbuf;

DESCRIPTION

This page describes general purpose regular expression matching routines in the form of *ed*(1), defined in **/usr/include/regexp.h**. Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), *expr*(1), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC( )	Return the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.
PEEKC( )	Return the next character in the regular expression. Successive calls to PEEKC( ) should return the same character (which should also be the next character returned by GETC( )).
UNGETC(*c*)	Cause the argument *c* to be returned by the next call to GETC( ) (and PEEKC( )). No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC( ). The value of the macro UNGETC(*c*) is always ignored.
RETURN(*pointer*)	This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.
ERROR(*val*)	This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf*– *expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each program that includes this file must have a **#define** statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for GETC( ), PEEKC( ) and UNGETC( ). Otherwise it can be used to declare external variables that might be used by GETC( ), PEEKC( ) and UNGETC( ). See the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

```
step(string, expbuf)
```

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns one, if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, loc1 will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

Step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* will only try to match the regular expression to the beginning of the string. If more than one regular expression is to be compiled before the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns a one indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called, simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, *advance* will break out of the loop that backs up and will return zero. This is used be *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like **s/y*//g** do not loop forever.

The routines *ecmp* and *getrange* are trivial and are called by the routines previously mentioned.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT            register char *sp = instring;
#define GETC( )         (*sp++)
#define PEEKC( )        (*sp)
#define UNGETC(c)       (--sp)
#define RETURN(c)       return;
#define ERROR(c)        regerr( )

#include <regexp.h>
...
                compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
                if(step(linebuf, expbuf))
                                succeed( );
```

FILES

/usr/include/regexp.h

SEE ALSO

ed(1), grep(1), sed(1).

BUGS

The handling of *circf* is kludgy.
The routine *ecmp* is equivalent to the Standard I/O routine *strncmp* and should be replaced by that routine.
The actual code is probably easier to understand than this manual page.

NAME

stat – data returned by stat system call

SYNTAX

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

DESCRIPTION

The system calls *stat* and *fstat* return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

```
/* Structure of the result of stat */

struct  stat
{
        dev_t   st_dev;
      vert_t st_vers;
        ino_t   st_ino;
        ushort  st_mode;
        short   st_nlink;
        ushort  st_uid;
        ushort  st_gid;
        dev_t   st_rdev;
        off_t   st_size;
        time_t  st_atime;
        time_t  st_mtime;
        time_t  st_ctime;
};


#define S_IFMT     0170000      /* type of file */
#define S_IFDIR    0040000      /* directory */
#define S_IFCHR    0020000      /* character special */
#define S_IFBLK    0060000      /* block special */
#define S_IFREG    0100000      /* regular */
#define S_IFIFO    0010000      /* fifo */
#define S_ISUID    04000   /* set user id on execution */
#define S_ISGID    02000   /* set group id on execution */
#define S_ISVTX    01000   /* save swapped text even after use */
#define S_IREAD    00400   /* read permission, owner */
#define S_IWRITE   00200   /* write permission, owner */
#define S_IEXEC    00100   /* execute/search permission, owner */
```

FILES

/usr/include/sys/types.h
/usr/include/sys/stat.h

SEE ALSO

stat(2B), types(5), fs(4).

NAME

sysrc – command script for drivers

SYNTAX

/etc/sysrc

DESCRIPTION

Sysrc is a script of commands executed when the system is booted or rebooted. It can start all system drivers; it is not intended to start demons.

SEE ALSO

rc(5)

NAME

term – conventional names for terminals

DESCRIPTION

These names are used by certain commands (e.g., *nroff*, *mm*(1), *man*(1), *tabs*(1)) and are maintained as part of the shell environment (see *sh*(1), *profile*(4), and *environ*(5)) in the variable $TERM:

1520	Datamedia 1520
1620	Diablo 1620 and others using the HyType II printer
1620–12	same, in 12-pitch mode
2621	Hewlett-Packard HP2621 series
2631	Hewlett-Packard 2631 line printer
2631–c	Hewlett-Packard 2631 line printer - compressed mode
2631–e	Hewlett-Packard 2631 line printer - expanded mode
2640	Hewlett-Packard HP2640 series
2645	Hewlett-Packard HP264n series (other than the 2640 series)
300	DASI/DTC/GSI 300 and others using the HyType I printer
300–12	same, in 12-pitch mode
300s	DASI/DTC/GSI 300s
382	DTC 382
300s–12	same, in 12-pitch mode
3045	Datamedia 3045
33	TELETYPE Terminal Model 33 KSR
37	TELETYPE Terminal Model 37 KSR
40–2	TELETYPE Terminal Model 40/2
40–4	TELETYPE Terminal Model 40/4
4540	TELETYPE Terminal Model 4540
3270	IBM Model 3270
4000a	Trendata 4000a
4014	Tektronix 4014
43	TELETYPE Model 43 KSR
450	DASI 450 (same as Diablo 1620)
450–12	same, in 12-pitch mode
735	Texas Instruments TI735 and TI725
745	Texas Instruments TI745
dumb	generic name for terminals that lack reverse line-feed and other special escape sequences
sync	generic name for synchronous TELETYPE 4540-compatible terminals
hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer
tn1200	General Electric TermiNet 1200
tn300	General Electric TermiNet 300
ridge–10	Ridge Monochrome Display in portrait orientation with 10– point font
ridge–12	same, with 12– point font
ridge–14	same, with 14– point font
ridge–16	same, with 16– point font
ridge–18	same, with 18– point font
ridge–20	same, with 20– point font
ridge–24	same, with 24– point font

Up to 8 characters, chosen from [- a- z0- 9], make up a basic terminal name. Terminal sub-models and operational modes are distinguished by suffixes beginning with a – . Names should generally be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name.

Commands whose behavior depends on the type of terminal should accept arguments of the form – T*term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable $TERM, which, in turn, should contain *term*.

SEE ALSO

mm(1), nroff(1), tplot(1G), sh(1), stty(1), tabs(1), profile(4), environ(5).

BUGS

This is a small candle trying to illuminate a large, dark problem. Programs that ought to adhere to this nomenclature do so somewhat fitfully.

NAME

termcap – terminal capability data base

SYNTAX

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, *e.g.*, by *vi*(1). 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 character 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 horizontal 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		Safe to move while in standout and underline mode
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,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		Terminal underlines but does not 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 are destructive, magic so char (Teleray 1061)

A Sample Entry

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

```
c1 |c100 |concept100:is=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200:\
  :al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :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|si 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	Reverse 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 **x***x*.

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

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since *termlib* routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be cancelled with **xx@** where xx is the capability. For example, the entry

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

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

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), termcap(3), vi(1) , Term (4), Terminfo (4)

AUTHOR

University of California at Berkeley

BUGS

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

NAME

types – primitive system data types

SYNTAX

#include <sys/types.h>

DESCRIPTION

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

```
typedef  struct { int r[1]; } *        physadr;
typedef  long                          daddr_t;
typedef  char *                        caddr_t;
typedef  unsigned int                  uint;
typedef  unsigned short                ushort;
typedef unsigned int                   vers_t;
typedef  unsigned int                  ino_t;
typedef short                          cnt_t;
typedef  long                          time_t;
typedef  int                           label_t[10];
typedef  unsigned int                  dev_t;
typedef  long                          off_t;
typedef  long                          paddr_t;
typedef  long                          key_t;
```

The form *daddr_t* is used for disk addresses. Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

SEE ALSO

fs(4).

NAME

values – machine-dependent values

SYNTAX

#include <values.h>

DESCRIPTION

This file contains a set of manifest constants, conditionally defined for particular processor architectures.

The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit.

BITS(*type*)	The number of bits in a specified type (e.g., int).
HIBITS	The value of a short integer with only the high-order bit set (0x8000).
HIBITL	The value of a long integer with only the high-order bit set (0x80000000).
HIBITI	The value of a regular integer with only the high-order bit set (the same as HIBITS or HIBITL).
MAXSHORT	The maximum value of a signed short integer (0x7FFF ≡ 32767).
MAXLONG	The maximum value of a signed long integer (0x7FFFFFFF ≡ 2147483647).
MAXINT	The maximum value of a signed regular integer (the same as MAXSHORT or MAXLONG).
MAXFLOAT, LN_MAXFLOAT	The maximum value of a single-precision floating-point number, and its natural logarithm (6.805646932770577e30, and 89.3).
MAXDOUBLE, LN_MAXDOUBLE	The maximum value of a double-precision floating-point number, and its natural logarithm (3.595386269724629e308, and 709).
MINFLOAT, LN_MINFLOAT	The minimum positive value of a single-precision floating-point number, and its natural logarithm (5.877472454760670 9e-39, and -88).
MINDOUBLE, LN_MINDOUBLE	The minimum positive value of a double-precision floating-point number, and its natural logarithm (1.1125369292536015e-308, and -709).
FSIGNIF	The number of significant bits in the mantissa of a single-precision floating-point number (24).
DSIGNIF	The number of significant bits in the mantissa of a double-precision floating-point number (56).

FILES

/usr/include/values.h

SEE ALSO

intro(3), math(5).

NAME

varargs – handle variable argument list

SYNTAX

#include <varargs.h>

va_alist

va_dcl

void va_start(pvar)
va_list pvar;

type **va_arg(pvar,** *type***)**
va_list pvar;

void va_end(pvar)
va_list pvar;

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists (such as *printf*(3S)) but do not use *varargs* are inherently nonportable, as different machines use different argument-passing conventions.

va_alist is used as the parameter list in a function header.

va_dcl is a declaration for *va_alist.* No semicolon should follow *va_dcl.*

va_list is a type defined for the variable used to traverse the list.

va_start is called to initialize *pvar* to the beginning of the list.

va_arg will return the next argument in the list pointed to by *pvar*. *Type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

va_end is used to clean up.

Multiple traversals, each bracketed by *va_start* ... *va_end,* are possible.

EXAMPLE

This example is a possible implementation of *execl*(2).

```
#include <varargs.h>
#define MAXARGS     100

/*      execl is called by
                execl(file, arg1, arg2, ..., (char *)0);
*/
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[MAXARGS];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while ((args[argno++] = va_arg(ap, char *)) != (char *)0)
                ;
        va_end(ap);
        return execv(file, args);
}
```

SEE ALSO

exec(2), printf(3S).

BUGS

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this from the stack frame. For example, *execl* is passed a zero pointer to signal the end of the list. *Printf* can tell how many arguments are there by the format.
It is non-portable to specify a second argument of *char, short,* or *float* to *va_arg,* since arguments seen by the called function are not *char, short,* or *float.* C converts *char* and *short* arguments to *int* and converts *float* arguments to *double* before passing them to a function.

BLANK

NAME

intro – introduction to games

DESCRIPTION

Section six describes the recreational and educational programs found in the **/usr/games** directory. To run any game, type **/usr/games/***gamename.*

NAME

arithmetic – provide drill in number facts

SYNTAX

/usr/games/arithmetic [**+– x/**] [range]

DESCRIPTION

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

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

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

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

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

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

NAME

backgammon – another game of backgammon

SYNTAX

/usr/games/backgammon

DESCRIPTION

Backgammon asks whether you need instructions, then plays backgammon.

NAME

balls – demonstrate rubber balls

SYNTAX

/usr/games/balls

DESCRIPTION

Balls demonstrates five bouncing rubber balls. It must be run on the Ridge graphics display.

NAME

bcd – convert to antique media

SYNTAX

/usr/games/bcd text

DESCRIPTION

Bcd converts the literal *text* into a form familiar to old-timers.

SEE ALSO

dd(1)

NAME

bj – the game of black jack

SYNTAX

/usr/games/bj

DESCRIPTION

Bj simulates the dealer in the game of black jack (or twenty-one). The following rules apply:

The bet is $2 every hand.

A player "natural" (black jack) pays $3. A dealer natural loses $2. Both dealer and player naturals is a "push" (no money exchange).

If the dealer has an ace up, the player is allowed to make an "insurance" bet against the chance of a dealer natural. If this bet is not taken, play resumes as normal. If the bet is taken, it is a side bet where the player wins $2 if the dealer has a natural and loses $1 if the dealer does not.

If the player is dealt two cards of the same value, he is allowed to "double". He is allowed to play two hands, each with one of these cards. (The bet is doubled also; $2 on each hand.)

If a dealt hand has a total of ten or eleven, the player may "double down". He may double the bet ($2 to $4) and receive exactly one more card on that hand.

Under normal play, the player may "hit" (draw a card) as long as his total is not over twenty-one. If the player "busts" (goes over twenty-one), the dealer wins the bet.

When the player "stands" (decides not to hit), the dealer hits until he attains a total of seventeen or more. If the dealer busts, the player wins the bet.

If both player and dealer stand, the one with the largest total wins. A tie is a push.

The machine deals and keeps score. The following questions will be asked at appropriate times. Each question is answered by **y** followed by a new-line for "yes", or just new-line for "no".

?	(means, "do you want a hit?")
Insurance?	
Double down?	

Every time the deck is shuffled, the dealer so states and the "action" (total bet) and "standing" (total won or lost) is printed. To exit, hit the interrupt key (DEL) and the action and standing will be printed.

NAME

boggle – play the game of boggle

SYNTAX

/usr/games/boggle [+] [++]

DESCRIPTION

This program is intended for people wishing to sharpen their skills at Boggle (TM Parker Bros.). If you invoke the program with 4 arguments of 4 letters each, (*e.g.* boggle appl epie moth erhd) the program forms the obvious Boggle grid and lists all the words from **/usr/dict/words** found therein. If you invoke the program without arguments, it will generate a board for you, let you enter words for 3 minutes, and then tell you how well you did relative to **/usr/dict/words**.

The object of Boggle is to find, within 3 minutes, as many words as possible in a 4 by 4 grid of letters. Words may be formed from any sequence of 3 or more adjacent letters in the grid. The letters may join horizontally, vertically, or diagonally. However, no position in the grid may be used more than once within any one word. In competitive play amongst humans, each player is given credit for those of his words which no other player has found.

In interactive play, enter your words separated by spaces, tabs, or newlines. A bell will ring when there is 2:00, 1:00, 0:10, 0:02, 0:01, and 0:00 time left. You may complete any word started before the expiration of time. You can surrender before time is up by hitting 'break'. While entering words, your erase character is only effective within the current word and your line kill character is ignored.

Advanced players may wish to invoke the program with 1 or 2 + 's as the first argument. The first + removes the restriction that positions can only be used once in each word. The second + causes a position to be considered adjacent to itself as well as its (up to) 8 neighbors.

NAME

BTLGAMMON – the game of backgammon

SYNTAX

/usr/games/btlgammon

DESCRIPTION

Btlgammon is a program which provides a partner for the game of backgammon. It is designed to play at three different levels of skill, one of which you must select. In addition to selecting the opponent's level, you may also indicate that you would like to roll your own dice during your turns (for the superstitious players). You will also be given the opportunity to move first. The practice of each player rolling one die for the first move is not incorporated.

The points are numbered 1– 24, with 1 being white's extreme inner table, 24 being brown's inner table, 0 being the bar for removed white pieces and 25 the bar for brown. For details on how moves are expressed, type **y** when *btlgammon* asks ''Instructions?'' at the beginning of the game. When *btlgammon* first asks ''Move?'', type **?** to see a list of move options other than entering your numerical move.

When the game is finished, *btlgammon* will ask you if you want the log. If you respond with **y**, *btlgammon* will attempt to append to or create a file **back.log** in the current directory.

FILES

/usr/games/lib/backrules	rules file
/tmp/b*	log temp file
back.log	log file

BUGS

The only level really worth playing is ''expert'', and it only plays the forward game.
Btlgammon will complain loudly if you attempt to make too *many* moves in a turn, but will become very silent if you make too *few*.
Doubling is not implemented.

NAME

canfield, cfscores – the solitaire card game canfield

SYNTAX

/usr/games/canfield
/usr/games/cfscores

DESCRIPTION

If you have never played solitaire before, it is recommended that you consult a solitaire instruction book. In Canfield, tableau cards may be built on each other downward in alternate colors. An entire pile must be moved as a unit in building. Top cards of the piles are available to be able to be played on foundations, but never into empty spaces.

Spaces must be filled from the stock. The top card of the stock also is available to be played on foundations or built on tableau piles. After the stock is exhausted, tableau spaces may be filled from the talon and the player may keep them open until he wishes to use them.

Cards are dealt from the hand to the talon by threes and this repeats until there are no more cards in the hand or the player quits. To have cards dealt onto the talon the player types 'ht' for his move. Foundation base cards are also automatically moved to the foundation when they become available.

The command 'c' causes *canfield* to maintain card counting statistics on the bottom of the screen. When properly used this can greatly increase ones chances of winning.

The rules for betting are somewhat less strict than those used in the official version of the game. The initial deal costs $13. You may quit at this point or inspect the game. Inspection costs $13 and allows you to make as many moves as is possible without moving any cards from your hand to the talon. (the initial deal places three cards on the talon; if all these cards are used, three more are made available.) Finally, if the game seems interesting, you must pay the final installment of $26. At this point you are credited at the rate of $5 for each card on the foundation; as the game progresses you are credited with $5 for each card that is moved to the foundation. Each run through the hand after the first costs $5. The card counting feature costs $1 for each unknown card that is identified. If the information is toggled on, you are only charged for cards that became visible since it was last turned on. Thus the maximum cost of information is $34. Playing time is charged at a rate of $1 per minute.

With no arguments, the program *cfscores* prints out the current status of your canfield account. If a user name is specified, it prints out the status of their canfield account. If the – **a** flag is specified, it prints out the canfield accounts for all users that have played the game since the database was set up.

FILES

/usr/games/canfield	the game itself
/usr/games/cfscores	the database printer
/usr/games/lib/cfscores	the database of scores

BUGS

It is impossible to cheat.

NAME

craps – the game of craps

SYNTAX

/usr/games/craps

DESCRIPTION

Craps is a form of the game of craps that is played in Las Vegas. The program simulates the *roller*, while the user (the *player*) places bets. The player may choose, at any time, to bet with the roller or with the *House*. A bet of a negative amount is taken as a bet with the House, any other bet is a bet with the roller.

The player starts off with a "bankroll" of $2,000.

The program prompts with:

```
bet?
```

The bet can be all or part of the player's bankroll. Any bet over the total bankroll is rejected and the program prompts with **bet?** until a proper bet is made.

Once the bet is accepted, the roller throws the dice. The following rules apply (the player wins or loses depending on whether the bet is placed with the roller or with the House; the odds are even). The *first* roll is the roll immediately following a bet:

1. On the first roll:

7 or 11	wins for the roller;
2, 3, or 12	wins for the House;
any other number	is the *point*, roll again (Rule 2 applies).

2. On subsequent rolls:

point	roller wins;
7	House wins;
any other number	roll again.

If a player loses the entire bankroll, the House will offer to lend the player an additional $2,000. The program will prompt:

```
marker?
```

A **yes** (or **y**) consummates the loan. Any other reply terminates the game.

If a player owes the House money, the House reminds the player, before a bet is placed, how many markers are outstanding.

If, at any time, the bankroll of a player who has outstanding markers exceeds $2,000, the House asks:

```
Repay marker?
```

A reply of **yes** (or **y**) indicates the player's willingness to repay the loan. If only 1 marker is outstanding, it is immediately repaid. However, if more than 1 marker are outstanding, the House asks:

```
How many?
```

markers the player would like to repay. If an invalid number is entered (or just a carriage return), an appropriate message is printed and the program will prompt with **How many?** until a valid number is entered.

If a player accumulates 10 markers (a total of $20,000 borrowed from the House), the program informs the player of the situation and exits.

Should the bankroll of a player who has outstanding markers exceed $50,000, the *total* amount of money borrowed will be *automatically* repaid to the House.

Any player who accumulates $100,000 or more breaks the bank. The program then prompts:

    New game?

to give the House a chance to win back its money.

Any reply other than **yes** is considered to be a **no** (except in the case of **bet?** or **How many?**). To exit, send an interrupt (break), DEL, or control-D. The program will indicate whether the player won, lost, or broke even.

MISCELLANEOUS

The random number generator for the die numbers uses the seconds from the time of day. Depending on system usage, these numbers, at times, may seem strange but occurrences of this type in a real dice situation are not uncommon.

NAME

cribbage – the card game cribbage

SYNTAX

/usr/games/cribbage [**– req**] *name ...*

DESCRIPTION

Cribbage plays the card game cribbage, with the program playing one hand and the user the other. The program will initially ask the user if the rules of the game are needed – if so, it will print out the appropriate section from *According to Hoyle* with *more (I)*.

Cribbage options include:

– e When the player makes a mistakes scoring his hand or crib, provide an explanation of the correct score. (This is especially useful for beginning players.)

– q Print a shorter form of all messages – this is only recommended for users who have played the game without specifying this option.

– r Instead of asking the player to cut the deck, the program will randomly cut the deck.

Cribbage first asks the player whether he wishes to play a short game (once around, to 61) or a long game (twice around, to 121). A response of 's' will result in a short game, any other response will play a long game.

At the start of the first game, the program asks the player to cut the deck to determine who gets the first crib. The user should respond with a number between 0 and 51, indicating how many cards down the deck is to be cut. The player who cuts the lower ranked card gets the first crib. If more than one game is played, the loser of the previous game gets the first crib in the current game.

For each hand, the program first prints the player's hand, whose crib it is, and then asks the player to discard two cards into the crib. The cards are prompted for one per line, and are typed as explained below.

After discarding, the program cuts the deck (if it is the player's crib) or asks the player to cut the deck (if it's its crib); in the later case, the appropriate response is a number from 0 to 39 indicating how far down the remaining 40 cards are to be cut.

After cutting the deck, play starts with the non-dealer (the person who doesn't have the crib) leading the first card. Play continues, as per cribbage, until all cards are exhausted. The program keeps track of the scoring of all points and the total of the cards on the table.

After play, the hands are scored. The program requests the player to score his hand (and the crib, if it is his) by printing out the appropriate cards (and the cut card enclosed in brackets). Play continues until one player reaches the game limit (61 or 121).

A carriage return when a numeric input is expected is equivalent to typing the lowest legal value; when cutting the deck this is equivalent to choosing the top card.

Cards are specified as rank followed by suit. The ranks may be specified as one of: 'a', '2', '3', '4', '5', '6', '7', '8', '9', 't', 'j', 'q', and 'k', or alternatively, one of: ace, two, three, four, five, six, seven, eight, nine, ten, jack, queen, and king. Suits may be specified as: 's', 'h', 'd', and 'c', or alternatively as: spades, hearts, diamonds, and clubs. A card may be specified as: <rank> <suit>, or: <rank> of <suit>. If the single letter rank and suit designations are used, the space separating the suit and rank may be left out. Also, if only one card of the desired rank is playable, typing the rank is sufficient. For example, if your hand was 2H, 4D, 5C, 6H, JC, KD and it was desired to discard the king of diamonds, any of the following could be typed: k, king, kd, k d, k of d, king d, king of d, k diamonds, k of diamonds, king diamonds, or king of diamonds.

NAME

fish – play "Go Fish"

SYNTAX

/usr/games/fish

DESCRIPTION

Fish plays the game of Go Fish, a children's card game. The object is to accumulate 'books' of 4 cards with the same face value. The players alternate turns; each turn begins with one player selecting a card from his hand, and asking the other player for all cards of that face value. If the other player has one or more cards of that face value in his hand, he gives them to the first player, and the first player makes another request. Eventually, the first player asks for a card which is not in the second player's hand: he replies 'GO FISH!' The first player then draws a card from the 'pool' of undealt cards. If this is the card he had last requested, he draws again. When a book is made, either through drawing or requesting, the cards are laid down and no further action takes place with that face value.

To play the computer, simply make guesses by typing a, 2, 3, 4, 5, 6, 7, 8, 9, 10, j, q, or k when asked. Hitting return gives you information about the size of my hand and the pool, and tells you about my books. Saying 'p' as a first guess puts you into 'pro' level; the default is quite unsophisticated.

NAME

fortune – print a random, hopefully interesting, adage

SYNTYAX

/usr/games/fortune [–] [– **wslao**]

DESCRIPTION

Fortune with no arguments prints out a random adage. The flags mean:

– **w** Waits before termination for an amount of time calculated from the number of characters in the message. This is useful if it is executed as part of the logout procedure to guarantee that the message can be read before the screen is cleared.

– **s** Short messages only.

– **l** Long messages only.

– **o** Choose from an alternate list of adages, often used for potentially offensive ones.

– **a** Choose from either list of adages.

FILES

/usr/games/lib/fortunes.dat

NAME

hangman – Computer version of the game hangman

SYNTAX

/usr/games/hangman

DESCRIPTION

In *hangman,* the computer picks a word from the on-line word list and you must try to guess it. The computer keeps track of which letters have been guessed and how many wrong guesses you have made on the screen in a graphic fashion.

FILES

/usr/dict/words	On-line word list

NAME

life – the John Conway game of life

SYNTAX

/usr/games/life

DESCRIPTION

Life demonstrates the birth and death of individuals in a population. As areas of the screen get overcrowded, individuals die. If the population is suitable for growth, individuals multiply.

Life requires a Ridge graphics display.

Life runs until the user presses the **DEL** key.

NAME

master – the game of mastermind

SYNTAX

/usr/games/master

DESCRIPTION

The playing field is a number of slots, in which a number of colored pegs can be placed. I will start by placing a peg in each slot, and you will then have to guess what color peg I have in each slot. You will then place a peg in each slot, and I will have to guess what color peg is in each slot. You get one point for each guess I have to make, and I get one point for each guess you have to make.

A guess consists of a possible sequence of colored pegs. The guesser's opponent then answers with two numbers: the number of pegs in the guess that exactly match the corresponding pegs in the configuration, and the number of pegs in the guess that match in color but not in position. For example, let's say we are playing with five slots, and the following situation occurs:

my configuration:	red	red	yellow	blue	brown
your guess:	blue	red	green	red	red

The two numbers would then be 1 and 2. The 1 is because we each have a red peg in the second slot. In addition, your blue matches my blue, though the position is wrong, and one of your reds matches my red in the first slot. Only two of your reds match because I only have two reds in my configuration.

You will have a chance to decide before we start on how many slots and how many colors you want to use.

When you enter a guess, type the names of the colors, separated by spaces. When I make a guess, answer me with two digits, possibly separated by spaces.

Any time it is your turn to enter a guess, you can ask me what happened by typing "review" instead of your guess.

NAME

maze – generate a maze

SYNTAX

/usr/games/maze

DESCRIPTION

Maze asks a few questions and then prints a maze.

BUGS

Some mazes (especially small ones) have no solutions.

NAME

mille – play Mille Bournes

SYNTAX

/usr/games/mille [file]

DESCRIPTION

Mille plays a two-handed game like the Parker Brother's game of Mille Bournes. The rules are described below. If a file name is given on the command line, the game saved in that file is started.

When a game is started up, the bottom of the score window will contain a list of commands. They are:

P Pick a card from the deck. This card is placed in the 'P' slot in your hand.

D Discard a card from your hand. To indicate which card, type the number of the card in the hand (or P for the just-picked card) followed by a <RETURN> or <SPACE>. The <RETURN or <SPACE> is required to allow recovery from typos which can be very expensive, like discarding safeties.

U Use a card. The card is again indicated by its number, followed by a <RETURN> or <SPACE>.

O Toggle ordering the hand. By default off, if turned on it will sort the cards in your hand appropriately. This is not recommended for the impatient on slow terminals.

Q Quit the game. This will ask for confirmation, just to be sure. Hitting <DELETE> (or <RUBOUT>) is equivalent.

S Save the game in a file. If the game was started from a file, you will be given an opportunity to save it on the same file. If you don't wish to, or you did not start from a file, you will be asked for the file name. If you type a <RETURN> without a name, the save will be terminated and the game resumed.

R Redraw the screen from scratch. The command ^L (control 'L') will also work.

W Toggle window type. This switches the score window between the startup window (with all the command names) and the end-of-game window. Using the end-of-game window saves time by eliminating the switch at the end of the game to show the final score. Recommended for hackers and other miscreants.

If you make a mistake, an error message will be printed on the last line of the score window, and a bell will beep.

At the end of each hand or game, you will be asked if you wish to play another. If not, it will ask you if you want to save the game. If you do, and the save is unsuccessful, play will be resumed as if you had said you wanted to play another hand/game. This allows you to use the S command to reattempt the save.

AUTHOR

Ken Arnold
(The game itself is a product of Parker Brothers, Inc.)

SEE ALSO

curses(3X), *Screen Updating and Cursor Movement Optimization: A Library Package*, Ken Arnold

CARDS

Here is some useful information. The number in parentheses after the card name is the number of that card in the deck:

Hazard	Repair	Safety
Out of Gas (2)	Gasoline (6)	Extra Tank (1)
Flat Tire (2)	Spare Tire (6)	Puncture Proof (1)
Accident (2)	Repairs (6)	Driving Ace (1)
Stop (4)	Go (14)	Right of Way (1)
Speed Limit (3)	End of Limit (6)	

25 – (10), 50 – (10), 75 – (10), 100 – (12), 200 – (4)

RULES

Object: The point of game is to get a total of 5000 points in several hands. Each hand is a race to put down exactly 700 miles before your opponent does. Beyond the points gained by putting down milestones, there are several other ways of making points.

Overview: The game is played with a deck of 101 cards. *Distance* cards represent a number of miles traveled. They come in denominations of 25, 50, 75, 100, and 200. When one is played, it adds that many miles to the player's trip so far this hand. *Hazard* cards are used to prevent your opponent from putting down Distance cards. They can only be played if your opponent has a *Go* card on top of the Battle pile. The cards are *Out of Gas, Accident, Flat Tire, Speed Limit,* and *Stop*. *Remedy* cards fix problems caused by Hazard cards played on you by your opponent. The cards are *Gasoline, Repairs, Spare Tire, End of Limit,* and *Go*. *Safety* cards prevent your opponent from putting specific Hazard cards on you in the first place. They are *Extra Tank, Driving Ace, Puncture Proof,* and *Right of Way,* and there are only one of each in the deck.

Board Layout: The board is split into several areas. From top to bottom, they are: **SAFETY AREA** (unlabeled): This is where the safeties will be placed as they are played. **HAND**: These are the cards in your hand. **BATTLE**: This is the Battle pile. All the Hazard and Remedy Cards are played here, except the *Speed Limit* and *End of Limit* cards. Only the top card is displayed, as it is the only effective one. **SPEED**: The Speed pile. The *Speed Limit* and *End of Limit* cards are played here to control the speed at which the player is allowed to put down miles. **MILEAGE**: Miles are placed here. The total of the numbers shown here is the distance traveled so far.

Play: The first pick alternates between the two players. Each turn usually starts with a pick from the deck. The player then plays a card, or if this is not possible or desirable, discards one. Normally, a play or discard of a single card constitutes a turn. If the card played is a safety, however, the same player takes another turn immediately.

This repeats until one of the players reaches 700 points or the deck runs out. If someone reaces 700, they have the option of going for an *Extension*, which means that the play continues until someone reaches 1000 miles.

Hazard and Remedy Cards: Hazard Cards are played on your opponent's Battle and Speed piles. Remedy Cards are used for undoing the effects of your opponent's nastyness.

Go (Green Light) must be the top card on your Battle pile for you to play any mileage, unless you have played the *Right of Way* card (see below).

Stop is played on your opponent's *Go* card to prevent them from playing mileage until they play a *Go* card.

Speed Limit is played on your opponent's Speed pile. Until they play an *End of Limit* they can only play 25 or 50 mile cards, presuming their *Go* card allows them to do even that.

End of Limit is played on your Speed pile to nullify a *Speed Limit* played by your opponent.

Out of Gas is played on your opponent's *Go* card. They must then play a *Gasoline* card, and then a *Go* card before they can play any more mileage.

Flat Tire is played on your opponent's *Go* card. They must then play a *Spare Tire* card, and then a *Go* card before they can play any more mileage.

Accident is played on your opponent's *Go* card. They must then play a *Repairs* card, and then a *Go* card before they can play any more mileage.

Safety Cards: Safety cards prevent your opponent from playing the corresponding Hazard cards on you for the rest of the hand. It cancels an attack in progress, and *always entitles the player to an extra turn.*

Right of Way prevents your opponent from playing both *Stop* and *Speed Limit* cards on you. It also acts as a permanent *Go* card for the rest of the hand, so you can play mileage as long as there is not a Hazard card on top of your Battle pile. In this case only, your opponent can play Hazard cards directly on a Remedy card besides a Go card.

Extra Tank When played, your opponent cannot play an *Out of Gas* on your Battle Pile.

Puncture Proof When played, your opponent cannot play a *Flat Tire* on your Battle Pile.

Driving Ace When played, your opponent cannot play an *Accident* on your Battle Pile.

Distance Cards: Distance cards are played when you have a *Go* card on your Battle pile, or a Right of Way in your Safety area and are not stopped by a Hazard Card. They can be played in any combination that totals exactly 700 miles, except that *you cannot play more than two 200 mile cards in one hand.* A hand ends whenever one player gets exactly 700 miles or the deck runs out. In that case, play continues until neither someone reaches 700, or neither player can use any cards in their hand. If the trip is completed after the deck runs out, this is called *Delayed Action.*

Coup Fourré: This is a French fencing term for a counter-thrust move as part of a parry to an opponents attack. In Mille Bournes, it is used as follows: If an opponent plays a Hazard card, and you have the corresponding Safety in your hand, you play it immediately, even *before* you draw. This immediately removes the Hazard card from your Battle pile, and protects you from that card for the rest of the game. This gives you more points (see Scoring below).

Scoring: Scores are totaled at the end of each hand, whether or not anyone completed the trip. The terms used in the Score window have the following meanings:

Milestones Played: Each player scores as many miles as they played before the trip ended.
Each Safety: 100 points for each safety in the Safety area.
All 4 Safeties: 300 points if all four safeties are played.
Each Coup Fouré: 300 points for each Coup Fouré accomplished.

The following bonus scores can apply only to the winning player.

Trip Completed: 400 points bonus for completing the trip to 700 or 1000.
Safe Trip: 300 points bonus for completing the trip without using any 200 mile cards.
Delayed Action: 300 points bonus for finishing after the deck was exhausted.
Extension: 200 points bonus for completing a 1000 mile trip.
Shut-Out: 500 points bonus for completing the trip before your opponent played any mileage cards.

Running totals are also kept for the current score for each player for the hand (**Hand Total**), the game (**Overall Total**), and number of games won (**Games**).

NAME

monop – Monopoly game

SYNTAX

/usr/games/monop [file]

DESCRIPTION

Monop is like the Parker Brother's game Monopoly, and monitors a game between 1 to 9 users. It is assumed that the rules of Monopoly are known. The game follows the standard rules, with the exception that, if a property would go up for auction and there are only two solvent players, no auction is held and the property remains unowned.

The game, in effect, lends the player money, so it is possible to buy something which you cannot afford. However, as soon as a person goes into debt, he must fix the problem, *i.e.*, make himself solvent, before play can continue. If this is not possible, the player's property reverts to his debtee, either a player or the bank. A player can resign at any time to any person or the bank, which puts the property back on the board, unowned.

Any time that the response to a question is a *string*, e.g., a name, place or person, you can type '?' to get a list of valid answers. It is not possible to input a negative number, nor is it ever necessary.

A Summary of Commands:

quit: quit game: This allows you to quit the game. It asks you if you're sure.

print: print board: This prints out the current board. The columns have the following meanings (column headings are the same for the **where**, **own holdings**, and **holdings** commands):

Name	The first ten characters of the name of the square
Own	The *number* of the owner of the property.
Price	The cost of the property (if any)
Mg	This field has a '*' in it if the property is mortgaged
#	If the property is a Utility or Railroad, this is the number of such owned by the owner. If the property is land, this is the number of houses on it.
Rent	Current rent on the property. If it is not owned, there is no rent.

where: where players are: Tells you where all the players are. A '*' indicates the current player.

own holdings:
List your own holdings, *i.e.*, money, get-out-of-jail-free cards, and property.

holdings: holdings list: Look at anyone's holdings. It will ask you whose holdings you wish to look at. When you are finished, type done.

shell: shell escape: Escape to a shell. When the shell dies, the program continues where you left off.

mortgage: mortgage property: Sets up a list of mortgageable property, and asks which you wish to mortgage.

unmortgage:
unmortgage property: Unmortgage mortgaged property.

buy:	buy houses: Sets up a list of monopolies on which you can buy houses. If there is more than one, it asks you which you want to buy for. It then asks you how many for each piece of property, giving the current amount in parentheses after the property name. If you build in an unbalanced manner (a disparity of more than one house within the same monopoly), it asks you to re-input things.
sell:	sell houses: Sets up a list of monopolies from which you can sell houses. it operates in an analogous manner to *buy*
card:	card for jail: Use a get-out-of-jail-free card to get out of jail. If you're not in jail, or you don't have one, it tells you so.
pay:	pay for jail: Pay $50 to get out of jail, from whence you are put on Just Visiting. Difficult to do if you're not there.
trade:	This allows you to trade with another player. It asks you whom you wish to trade with, and then asks you what each wishes to give up. You can get a summary at the end, and, in all cases, it asks for confirmation of the trade before doing it.
resign:	Resign to another player or the bank. If you resign to the bank, all property reverts to its virgin state, and get-out-of-jail free cards revert to the deck.
save:	save game: Save the current game in a file for later play. You can continue play after saving, either by adding the file in which you saved the game after the *monop* command, or by using the *restore* command (see below). It will ask you which file you wish to save it in, and, if the file exists, confirm that you wish to overwrite it.
restore:	restore game: Read in a previously saved game from a file. It leaves the file intact.
roll:	Roll the dice and move forward to your new location. If you simply hit the <RETURN> key instead of a command, it is the same as typing *roll*.

FILES

/usr/games/lib/cards.pck	Chance and Community Chest cards

BUGS

No command can be given an argument instead of a response to a query.

NAME

moo – guessing game

SYNTAX

/usr/games/moo

DESCRIPTION

Moo is a guessing game. The computer picks a number consisting of four distinct decimal digits. The player guesses four distinct digits, and the computer gives a bovine kind of answer for each guess:

cow is a correct digit in an incorrect position.

bull is a correct digit in a correct position.

The game continues until the player guesses the number (a score of four bulls).

NAME

number – convert Arabic numerals to English

SYNTAX

/usr/games/number

DESCRIPTION

Number copies the standard input to the standard output, changing each decimal number to fully spelled-out words.

NAME

psych – draw lines on Tektronix terminal

SYNTAX

/usr/games/space

DESCRIPTION

Psych draws some mildly psychotic lines on the screen. To run **psych** on the Ridge graphics display, first use the **settek(1)** command.

NAME

quiz – test your knowledge

SYNTAX

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

DESCRIPTION

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

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

The – **t** flag specifies 'tutorial' mode, where missed questions are repeated later, and material is gradually introduced as you learn.

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

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

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

FILES

/usr/games/quiz.k/*

BUGS

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

NAME
rain – animated raindrops

SYNTAX
/usr/games/rain

DESCRIPTION
Rain's display is modeled after the VAX/VMS program of the same name. The terminal has to be set for 9600 baud to obtain the proper effect.

As with all programs that use *termcap*, the TERM environment variable must be set (and exported) to the type of the terminal being used.

FILES
/etc/termcap

NAME

snake, snscore – display chase game

SYNTAX

/usr/games/snake [– **w***n*] [– **l***n*]
/usr/games/snscore

DESCRIPTION

The object of the game is to make as much money as possible without getting eaten by the snake. Snake is a display-based game which must be played on terminal that supports vi(1). The – **l** and – **w** options allow you to specify the length and width of the field. By default the entire screen (except for the last column) is used.

You are represented on the screen by an I. The snake is 6 squares long and is represented by S's. The money is $, and an exit is #. Your score is posted in the upper left hand corner.

You can move around using the same conventions as vi(1), the h, j, k, and l keys work, as do the arrow keys. Other possibilities include:

sefc	These keys are like hjkl but form a directed pad around the d key.
HJKL	These keys move you all the way in the indicated direction to the same row or column as the money. This does *not* let you jump away from the snake, but rather saves you from having to type a key repeatedly. The snake still gets all his turns.
SEFC	Likewise for the upper case versions on the left.
ATPB	These keys move you to the four edges of the screen. Their position on the keyboard is the mnemonic, e.g. P is at the far right of the keyboard.
x	This lets you quit the game at any time.
p	Points in a direction you might want to go.
w	Space warp to get out of tight squeezes, at a price.
!	Shell escape
^Z	Suspend the snake game, on systems which support it. Otherwise an interactive shell is started up.

To earn money, move to the same square the money is on. A new $ will appear when you earn the current one. As you get richer, the snake gets hungrier. To leave the game, move to the exit (#).

A record is kept of the personal best score of each player. Scores are only counted if you leave at the exit, getting eaten by the snake is worth nothing.

As in pinball, matching the last digit of your score to the number which appears after the game is worth a bonus.

To see who wastes time playing snake, run */usr/games/snscore* .

FILES

/usr/games/lib/snakerawscores	database of personal bests
/usr/games/lib/snake.log	log of games played
/usr/games/busy	program to determine if system too busy

BUGS

When playing on a small screen, it's hard to tell when you hit the edge of the screen.

The scoring function takes into account the size of the screen. A perfect function to do this equitably has not been devised.

NAME

space – intra-celestial missile wars with gravitational effects

SYNTAX

/usr/games/space

DESCRIPTION

Space is two-person game of firing missiles at each other. In turn, each player enters a missile launch angle (in the range 0 to 360), and a launch velocity (in the range 0 to 10).

Each trajectory is plotted on the display. With each turn, the players fine-tune their angles and velocities until one player blows up the other's spaceship.

With each prompt for a trajectory or velocity, the system prints the player's previous input as a reminder. To re-confirm that previous value, without re-entering the number by hand, just press [RETURN].

Velocities and angles may contain decimal fractions, such as **34.6**. Both spaceships use the same method of calculating angles. Zero degrees is the the right, 90 degrees is straight up, 180 degrees is to the left, and 270 degrees is straight down.

NAME

trk – star trek

SYNTAX

/usr/games/trk [+– **x/**] [range]

DESCRIPTION

The Federation of Planets is at war with the Klingon Empire, and as captain of the USS Enterprise, you must destroy the invasion fleet. In the game, the galaxy is divided into 64 quadrants on an eight-by-eight grid, with quadrant 0,0 in the upper left hand corner. Each quadrant is divided into 100 sectors on a ten-by-ten grid. Each sector contains one object (e.g., the Enterprise, a Klingon, or a star). Navigation is handled in degrees, with zero being straight up and ninty being to the right. Distances are measured in quadrants. One tenth quadrant is one sector. The galaxy contains starbases, at which you can dock to refuel, repair damages, etc. The galaxy also contains stars. Stars usually have a knack for getting in your way, but they can be triggered into going nova by shooting a photon torpedo at one, thereby (hopefully) destroying any adjacent Klingons. This is not a good practice however, because you are penalized for destroying stars. Also, a star will sometimes go supernova, which obliterates an entire quadrant. You must never stop in a supernova quadrant, although you may "jump over" one. Some starsystems have inhabited planets. Klingons can attack inhabited planets and enslave the populace, which they then put to work building more Klingon battle cruisers.

STARTING THE GAME

To start the game:

$ /usr/games/trk [– a] *filename*

If a *filename* is stated, a log of the game is written onto that file. If omitted, the file is not written. **-a** specifies the log file is to be appended to, not created anew. The game will ask you what length game you would like. Valid responses are "short", "medium", and "long". Ideally, the length of the game does not affect the difficulty, but currently the shorter games tend to be harder than the longer ones. You may also type "restart", which restarts a previously saved game. You will then be prompted for the skill, to which you must respond "novice", "fair", "good", "expert", "commadore", or "impossible". You should start out with a novice and work up, but if you really want to see how fast you can be slaughtered, start out with an impossible game. In general, throughout the game, if you forget what is appropriate the game will tell you what it expects if you just enter a question mark.

ISSUING COMMANDS

If the game expects you to enter a command, it will say "Command: " and wait for your response. Most commands can be abbreviated. At almost any time you can type more than one thing on a line. For example, to move straight up one quadrant, you can type

```
move 0 1
```

or you could just type

```
move
```

and the game would prompt you with

```
Course:
```

to which you could type

```
0 1
```

The "1" is the distance, which could be put on still another line. Also, the "move" command could have been abbreviated "mov", "mo", or just "m". If you are partway through a command and you change your mind, you can usually type "-1" to cancel the command. Klingons generally cannot hit you if you don't consume anything (e.g., time or energy), so some commands are considered "free". As soon as you consume anything though -- POW!

THE COMMANDS

====== Short Range Scan ======

Mnemonic: srscan
Shortest Appreviation: s
Full Commands: srscan
srscan yes/no
Consumes: nothing

The short range scan gives you a picture of the quadrant you are in, and (if you say "yes") a status report which tells you a whole bunch of interesting stuff. You can get a status report alone by using the *status* command. An example follows:

```
Short range sensor scan
  0 1 2 3 4 5 6 7 8 9
0 . . . . . . . * . * 0  stardate  3702.16
1 . . E . . . . . . . 1  condition RED
2 . . . . . . . . . * 2  position  0,3/1,2
3 * . . . . # . . . . 3  warp factor    5.0
4 . . . . . . . . . . 4  total energy   4376
5 . . * . * . . . . . 5  torpedoes 9
6 . . . @ . .   . . . 6  shields        down, 78%
7 . . . . . . . . . . 7  Klingons left  3
8 . . . K . . . . . . 8  time left 6.43
9 . . . . . . * . . . 9  life support   damaged, reserves = 2.4
  0 1 2 3 4 5 6 7 8 9
Distressed Starsystem Marcus VII
```

The cast of characters is as follows:

E	the hero
K	the villain
#	the starbase
*	stars
@	inhabited starsystem
	empty space
	a black hole

The name of the starsystem is listed underneath the short range scan. The word "distressed", if present, means that the starsystem is under attack. Short range scans are absolutely free. They use no time, no energy, and they don't give the Klingons another chance to hit you.

====== Status Report ======

Mnemonic: status
Shortest Abbreviation: st
Consumes: nothing

This command gives you information about the current status of the game and your ship, as follows:

Stardate
The current stardate.

Condition
: as follows:
  - RED -- in battle
  - YELLOW -- low on energy
  - GREEN -- normal state
  - DOCKED -- docked at starbase
  - CLOAKED -- the cloaking device is activated

Position
: Your current quadrant and sector.

Warp Factor
: The speed you will move at when you move under warp power (with the *move* command).

Total Energy
: Your energy reserves. If they drop to zero, you die. Energy regenerates, but the higher the skill of the game, the slower it regenerates.

Torpedoes
: How many photon torpedoes you have left.

Shields Whether your shields are up or down, and how effective they are if up (what percentage of a hit they will absorb).

Klingons Left
: Number of Klingons left.

Time Left
: How long the Federation can hold out if you sit on your tush and do nothing. If you kill Klingons quickly, this number goes up, otherwise, it goes down. If it hits zero, the Federation is conquered.

Life Support
: If "active", everything is fine. If "damaged", your reserves tell you how long you have to repair your life support or get to a starbase before you starve, suffocate, or something equally unpleasant.

Current Crew
: The number of crew members left. This figures does not include officers.

Brig Space
: The space left in your brig for Klingon captives.

Klingon Power
: The number of units needed to kill a Klingon. Remember, as Klingons fire at you they use up their own energy, so you probably need somewhat less than this.

Skill, Length
: The skill and length of the game you are playing.

Status information is absolutely free.

====== Long Range Scan ======
Mnemonic: lrscan
Shortest Abbreviation: l
Consumes: nothing

Long range scan gives you information about the eight quadrants that surround the quadrant you're in. A sample long range scan follows:

```
Long range scan for quadrant 0,3

       2     3     4
    -------------------
    !  *  !  *  !  *  !
    -------------------
  0 ! 108 !   6 !  19 !
    -------------------
  1 !   9 ! /// !   8 !
    -------------------
```

The three digit numbers tell the number of objects in the quadrants. The units digit tells the number of stars, the tens digit the number of starbases, and the hundreds digit is the number of Klingons. "*" indicates the negative energy barrier at the edge of the galaxy, which you cannot enter. "///" means that that is a supernova quadrant and must not be entered.

====== Damage Report ======
Mnemonic: damages
Shortest Abbreviation: da
Consumes: nothing

A damage report tells you what devices are damaged and how long it will take to repair them. Repairs proceed faster when you are docked at a starbase.

====== Set Warp Factor ======
Mnemonic: warp
Shortest Abbreviation: w
Full Command: warp factor
Consumes: nothing

The warp factor tells the speed of your starship when you move under warp power (with the *move* command). The higher the warp factor, the faster you go, and the more energy you use. The minimum warp factor is 1.0 and the maximum is 10.0. At speeds above warp 6 there is danger of the warp engines being damaged. The probability of this increases at higher warp speeds. Above warp 9.0 there is a chance of entering a time warp.

====== Move Under Warp Power ======
Mnemonic: move
Shortest Abbreviation: m
Full Command: move course distance
Consumes: time and energy

This is the usual way of moving. The course is in degrees and the distance is in quadrants. To move one sector specify a distance of 0.1. Time is consumed proportionately to the inverse of the warp factor squared, and directly to the distance. Energy is consumed as the warp factor cubed, and directly to the distance. If you move with your shields up it doubles the amount of energy consumed. When you move in a quadrant containing Klingons, they get a chance to attack you. The computer detects navigation errors. If the computer is out, you run the risk of running into things. The course is determined by the Space Inertial Navigation System [SINS]. As described in Star Fleet Technical Order TO:02:06:12, the SINS is calibrated, after which it becomes the base for navigation. If damaged, navigation becomes inaccurate. When it is fixed, Spock recalibrates it, however, it cannot be calibrated extremely accurately until you dock at starbase.

====== **Move Under Impulse Power** ======

Mnemonic: impulse
Shortest Abbreviation: i
Full Command: impulse course distance
Consumes: time and energy

The impulse engines give you a chance to maneuver when your warp engines are damaged; however, they are incredibly slow (0.095 quadrants/stardate). They require 20 units of energy to engage, and ten units per sector to move. The same comments about the computer and the SINS apply as above. There is no penalty to move under impulse power with shields up.

====== **Deflector Shields** ======

Mnemonic: shields
Shortest Abbreviation: sh
Full Command: shields up/down
Consumes: energy

Shields protect you from Klingon attack and nearby novas. As they protect you, they weaken. A shield which is 78% effective will absorb 78% of a hit and let 22% in to hurt you. The Klingons have a chance to attack you every time you raise or lower shields. Shields do not rise and lower instantaneously, so the hit you receive will be computed with the shields at an intermediate effectiveness. It takes energy to raise shields, but not to drop them.

====== **Cloaking Device** ======

Mnemonic: cloak
Shortest Abbreviation: cl
Full Command: cloak up/down
Consumes: energy

When you are cloaked, Klingons cannot see you, and hence they do not fire at you. They are useful for entering a quadrant and selecting a good position, however, weapons cannot be fired through the cloak due to the huge energy drain that it requires. The cloak up command only starts the cloaking process; Klingons will continue to fire at you until you do something which consumes time.

====== **Fire Phasers** ======

Mnmemonic: phasers
Shortest Abbreviation: p
Full Commands: phasers automatic amount
phasers manual amt1 course1 spread1 ...
Consumes: energy

Phasers are energy weapons; the energy comes from your ship's reserves ("total energy" on a srscan). It takes about 250 units of hits to kill a Klingon. Hits are cumulative as long as you stay in the quadrant. Phasers become less effective the further from a Klingon you are. Adjacent Klingons receive about 90% of what you fire, at five sectors about 60%, and at ten sectors about 35%. They have no effect outside of the quadrant. Phasers cannot be fired while shields are up; to do so would fry you. They have no effect on starbases or stars. In automatic mode the computer decides how to divide up the energy among the Klingons present; in manual mode you do that yourself. In manual mode firing you specify a direction, amount (number of units to fire) and spread (0 -> 1.0) for each of the six phaser banks. A zero amount terminates the manual input.

====== **Fire Photon Torpedoes** ======

Mnemonic: torpedo
Shortest Abbreviation: t
Full Command: torpedo course [yes/no] [burst angle]
Consumes: torpedoes

Photon torpedoes are projectile weapons. Torpedoes don't cause partial damage; you either hit and destroy the target, or you miss. A hit on a Klingon (or a starbase) destroys the target. Hitting a star usually causes it to go nova, and occasionally supernova. Photon torpedoes cannot be aimed precisely. They can be fired with shields up, but they get even more random as they pass through the shields. Torpedoes may be fired in bursts of three. If this is desired, the burst angle is the angle between the three shots, which may vary from one to fifteen. The word "no" says that a burst is not wanted; the word "yes" (which may be omitted if stated on the same line as the course) says that a burst is wanted. Photon torpedoes have no effect outside the quadrant.

====== **Onboard Computer Request** ======

Mnemonic: computer
Shortest Abbreviation: c
Full Command: computer request; request;...
Consumes: nothing

The computer command gives you access to the facilities of the onboard computer, which allows you to do all sorts of fascinating stuff. Computer requests are:

score   Shows your current score.

course quad/sect
    Computes the course and distance from whereever you are to the given location. If you type "course /x,y" you will be given the course to sector x,y in the current quadrant.

move quad/sect
    Identical to the course request, except that the move is executed.

chart
: prints a chart of the known galaxy, i.e., everything that you have seen with a long range scan. The format is the same as on a long range scan, except that "..." means that you don't yet know what is there, and ".1." means that you know that a starbase exists, but you don't know anything else. "$$$" mans the quadrant that you are currently in.

trajectory
: prints the course and distance to all the Klingons in the quadrant.

warpcost dist warp_factor
: computes the cost in time and energy to move 'dist' quadrants at warp 'warp_factor'.

impcost dist
: same as warpcost for impulse engines.

pheff range
: tells how effective your phasers are at a given range.

distresslist
: gives a list of currently distressed starbases and starsystems. More than one request may be stated on a line by seperating them with semicolons.

====== **Dock at Starbase** ======

Mnemonic: dock
Shortest Abbreviation: do
Consumes: nothing

You may dock at a starbase when you are in one of the eight adjacent sectors. When you dock you are resupplied with energy, photon torpedoes, and life support reserves. Repairs are also done faster at starbase. Any prisoners you have taken are unloaded. You do not recieve points for taking prisoners until this time. Starbases have their own deflector shields, so you are safe from attack while docked.

====== **Undock from Starbase** ======

Mnemonic: undock
Shortest Abbreviation: u
Consumes: nothing

This just allows you to leave starbase so that you may proceed on your way.

====== **Rest** ======

Mnemonic: rest
Shortest Abbreviation: r
Full Command: rest time
Consumes: time

This command allows you to rest to repair damages. It is not advisable to rest while under attack.

====== **Call Starbase For Help** ======

Mnemonic: help
Shortest Abbreviation: help
Consumes: nothing

You may call Starbase for help via your subspace radio. The Starbase has long-range transporter beams to get you, but they cannot always rematerialize you. You should avoid using this command unless absolutely necessary, for the above reason and because it counts heavily against you in the scoring.

====== **Visual Scan** ======

Mnemonic: visual
Shortest Abbreviation: v
Full Command: visual course
Consumes: time

When your short range scanners are out, you can still see what is out "there" by doing a visual scan. Unfortunately, you can only see three sectors at one time, and it takes 0.005 stardates to perform. The three sectors in the general direction of the course specified are examined and displayed.

====== **Abandon Ship** ======

Mnemonic: abandon
Shortest Abbreviation: abandon
Consumes: nothing

The officers escape the Enterprise in the shuttlecraft. If the transporter is working and there is an inhabitable starsystem in the area, the crew beams down, otherwise you leave them to die. You are given an old but still usable ship, the Faire Queene.

====== **Ram** ======

Mnemonic: ram
Shortest Abbreviation: ram
Full Command: ram course distance
Consumes: time and energy

This command is identical to "move", except that the computer doesn't stop you from making navigation errors. You get very nearly slaughtered if you ram anything.

====== **Self Destruct** ======

Mnemonic: destruct
Shortest Abbreviation: destruct
Consumes: everything

Your starship is self-destructed. Chances are you will destroy any Klingons (and stars, and starbases) left in your quadrant.

====== **Terminate the Game** ======

Mnemonic: terminate
Shortest Abbreviation: terminate
Full Command: terminate yes/no

Cancels the current game. No score is computed. If you answer yes, a new game will be started, otherwise trek exits.

====== **Call the Shell** ======

Mnemonic: !

Shortest Abbreviation: !

Temporarily escapes to the shell. When you log out of the shell you will return to the game. Kirk never used the Shell, but you can do it.

SCORING

The scoring algorithm is rather complicated. Basically, you get points for each Klingon you kill, for your Klingon per stardate kill rate, and a bonus if you win the game. You lose points for the number of Klingons left in the galaxy at the end of the game, for getting killed, for each star, starbase, or inhabited starsystem you destroy, for calling for help, and for each casualty you incur. You will be promoted if you play very well. You will never get a promotion if you call for help, abandon the Enterprise, get killed, destroy a starbase or inhabited starsystem, or destroy too many stars.

REFERENCE PAGE

Command	*Uses*	*Consumes*
ABANDON	shuttlecraft, transporter	-
CLoak Up/Down	cloaking device	energy
Computer request; request;...	computer	-
DAmages	-	-
DESTRUCT	computer	-
DOck	-	-
HELP	subspace radio	-
Impulse course distance	impulse engines, computer, SINS	time, energy
Lrscan	L.R. sensors	-
Move course distance	warp engines, computer, SINS	time, energy
Phasers Automatic amount	phasers, computer	energy
Phasers Manual amt1 course1 spread1 ...	phasers	energy
Torpedo course [Yes] angle/No	torpedo tubes	torpedoes
RAM course distance	warp engines, computer, SINS	time, energy
Rest time	-	time
!	-	-
SHields Up/Down	shields	energy
Srscan [Yes/No]	S.R. sensors	-
STatus	-	-
TERMINATE Yes/No	-	-
Undock	-	-
Visual course	-	time
Warp warp_factor	-	-

NAME

ttt, cubic – tic-tac-toe

SYNTAX

/usr/games/ttt
/usr/games/cubic

DESCRIPTION

Ttt is the X and O game popular in the first grade. This is a learning program that never makes the same mistake twice.

Although it learns, it learns slowly. It must lose nearly 80 games to completely know the game.

Cubic plays three-dimensional tic-tac-toe on a 4×4×4 board. Moves are specified as a sequence of three coordinate numbers in the range 1-4.

FILES

/usr/games/ttt.k	learning file

NAME

worm – the growing worm game

SYNTAX

/usr/games/worm [*size*]

DESCRIPTION

In *worm,* you are a little worm, your body is the "o"'s on the screen and your head is the "@". You move with the hjkl keys (as in the game snake). If you don't press any keys, you continue in the direction you last moved. The upper case HJKL keys move you as if you had pressed several (9 for HL and 5 for JK) of the corresponding lower case key (unless you run into a digit, then it stops).

On the screen you will see a digit. If your worm eats the digit, it will grow longer. The amount of growth depends on the digit. The object of the game is to eat a lot a digits and make the worm really long.

The game ends when the worm runs into either the sides of the screen, or itself. The current score (how much the worm has grown) is kept in the upper left corner of the screen.

size is the optional argument that sets the initial length of the worm. *Size* must be an integer in the range 1 to 75, inclusive.

NAME

worms – animated worms

SYNTAX

/usr/games/worms [– **field**] [– **length #**] [– **number #**] [– **trail**]

DESCRIPTION

Worms shows some creepy stuff on the screen.

– **field** makes a "field" for the worm(s) to eat.

– **trail** causes each worm to leave a trail behind it.

Figure the rest out yourself.

FILES

/etc/termcap

BUGS

The lower-right-hand character position will not be updated properly on a terminal that wraps at the right margin.

Terminal initialization is not performed.

NAME

wump – hunt-the-wumpus

SYNTAX

/usr/games/wump

DESCRIPTION

A wumpus is a creature that lives in a cave with several rooms connected by tunnels. In **wump**, the player wanders among the rooms, tries to shoot the wumpus with an arrow, and tries to avoid being eaten by the wumpus or falling into a bottomless pit. While hunting wumpi, a super-bat might pick you up and drop you in another room.

Wump offers on-line instructions.

NAME

intro – introduction to special files

DESCRIPTION

This section describes various special files that refer to specific hardware peripherals and ROS System device drivers. The names of the entries are generally derived from names for the hardware, as opposed to the names of the special files themselves. Characteristics of both the hardware device and the corresponding ROS System device driver are discussed where applicable.

DEVICE CODE

Each device in section (7) has a major device number, minor device number, a block- or character-type, and a standard file name. This paragraph gives those details.

BUGS

While the names of the entries *generally* refer to vendor hardware names, in certain cases these names are seemingly arbitrary for various historical reasons.

NAME

cdisp – Ridge color display

DESCRIPTION

/dev/cdisp refers to a Ridge color display controller connected to a Ridge color display monitor.

The device file must be opened for use; only one user may access it at a time. Bytes written to the controller are passed to the display immediately. If the user writes an odd number of bytes, the driver automatically appends a pad-byte of zero.

A write that requests input from the device must precede a read.

On a read, pending output finishes, then the device returns the number of bytes requested, or the number of bytes actually read, whichever is less.

DEVICE CODE

device type	*major*	*minor*	*block-type or character-type*	*standard file name*
Ridge color display	7	0..3	char	/dev/cdisp

FILES

/dev/cdisp
/drivers/dr11m

SEE ALSO

See the Metheus Omega 400 manuals for details on the byte sequences to control the color monitor.

BUGS

Only an even number of bytes should be requested from the device. (If an odd number is requested, the device will reduce the returned bytes by one to return an even number. The odd byte in question becomes a "pending byte", and is the first byte returned on the next request.)

NAME

clp – Centronics line printer

DESCRIPTION

/dev/clp refers to a Centronics-type printer connected to the DP/Centronics connector on the Ridge 32 back panel.

When it is closed, bytes written are printed and a page is ejected.

The driver interprets carriage return (0DH), newline (0AH), tab(09H), and form-feed (0CH) characters.

Two *ioctl*(2) system calls are available:

```
#include <sys/lprio.h>
ioctl (fildes, command, arg)
struct lprio *arg;
```

The *commands* are:

LPRGET	Get the current printer parameters and store in the *lprio* structure referenced by **arg**.
LPRSET	Set the current printer parameters from the structure referenced by **arg**.

This allows an external program to control tab expansion, tab size, carriage return expansion, and newline expansion.

DEVICE CODE

			block-type or	*standard*
device type	*major*	*minor*	*character-type*	*file name*
Centronics lp	4	16	char	/dev/clp

FILES

/dev/clp
/drivers/fdlp

NAME

disc – disc device

DESCRIPTION

/dev/disc refers to the boot device. The system automatically creates an entry called **/dev/disc** for the system boot device.

disc*cu* refers to a Ridge disc device. By convention, *c* is a controller number in the range 0..3, and *u* is a unit number on the controller in the range 0..3.

DEVICE CODE

device type	*major*	*minor*	*block-type or character-type*	*standard file name*
boot disc (system makes this)	8	0	block	/dev/disc
Disc device (mknod(1) makes this)	8	*U*	block	/dev/disc*U*

The minor number *U* is derived from a combination of the binary bits that represent the controller number and the unit number. It is calculated by:

*controller**	*unit*	*device-name*	(binary *minor)*	*minor*
0	0	/dev/disc00	00-00	0
0	1	/dev/disc01	00-01	1
0	2	/dev/disc02	00-10	2
0	3	/dev/disc03	00-11	3
1	0	/dev/disc10	01-00	4
1	1	/dev/disc11	01-01	5
1	2	/dev/disc12	01-10	6
1	3	/dev/disc13	01-11	7

By convention, controller 0 is the device with "2" encoded on its address DIP-switches.

The boot disc can be determined by checking the major and minor numbers of **/dev/disc**.

FILES

/dev/disc

/dev/disc00 /dev/disc01 /dev/disc02 /dev/disc03

/dev/disc10 /dev/disc11 /dev/disc12 /dev/disc13

NAME

disp – Ridge Monochrome Display

DESCRIPTION

/dev/disp0 through **/dev/disp3** refer to Ridge Monochrome displays attached to "Display" ports 1 through 4 on the Ridge 32 back panel.

Input/output control is described in termio(7). There are two ioctl(2) calls that apply to the Ridge Monochrome Display only:

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

If *arg* is **0**, the cursor will not be shown. If *arg* is **1**, the cursor will be shown when a read request is outstanding.

```
ioctl (fildes, BCGETSZ, are)
struct DisplayInfo *arg;

struct DisplayInfo {
        int width;
        int height;
        int *bitmap;
}
```

Get the parameters associated with the display and store them in the DisplayInfo structure referenced by *arg.*

The width and height of the display are measured in bits. The base address in virtual memory of the display is given in **bitmap**.

DEVICE CODE

device type	*major*	*minor*	*block-type or character-type*	*standard file name*
monochrome disp	6	0..3	block	/dev/disp0..disp3

FILES

/dev/disp0, /dev/disp1, /dev/disp2, /dev/disp3,
/drivers/disp,

SEE ALSO

Ridge Multi-Window Display Management in the **ROS Utility Guide** (9053).

NAME

dr11m – Ridge color display

DESCRIPTION

/dev/dr11m refers to a Ridge color display controller connected to a Ridge color display monitor.

The device file must be opened for use; only one user may access it at a time. Bytes written are buffered by the driver and written to the device.

A write that requests input from the device must precede a read.

A read is satisfied by flushing any pending output, then by returning to the user a value that is the lesser of the amount read and the amount requested.

DEVICE CODE

device type	*major*	*minor*	*block-type or character-type*	*standard file name*
dr11	7	0	character	/dev/dr11m

FILES

/dev/dr11m

SEE ALSO

See the Metheus Omega 400 manuals for details on the byte sequences to control the color monitor.

NAME

fl – floppy disc device driver

DESCRIPTION

/dev/fl refers to the floppy disc drive in the Ridge 32 cabinet. A file on the floppy disc is referred to by the device name, plus the / character, plus its own name. **/dev/fl/dog** refers to file **dog** on the floppy disc.

The floppy disc file name consists of 1 to 15 characters from the set [**A** to **Z**, **a** to **z**, **0** to **9**, and "**.**"]. When specifying a pathname, upper- and lowercase characters are considered equivalent. The floppy disc directory stores names in uppercase only.

The floppy disc can be either single- or doubled-sided, with either single- or double-density formatting following the IBM 3740 soft-sectored standard. A floppy disc is normally formatted by ROS utilities as double-sided, double-density with 512 bytes of data per block.

The directory and file structure maintained on a floppy disc is compatible with the University of California, San Diego (UCSD) Pascal system format. This structure allows up to 77 files per volume, with a single directory for the entire volume.

Files on a floppy disc may be read or written a block of bytes at a time, with no transformation or interpretation of the data performed by the device driver. ROS utilities are available to format a new floppy disc (zero(1)), to list the directory (dir(1)), and to duplicate the contents of the floppy (copyfloppy(1)). To compact the space allocation of a floppy disc, use copyfloppy(1).

FILES

/dev/fl
/drivers/fdlp

DEVICE CODE

device type	*major*	*minor*	*block-type or character-type*	*standard file name*
floppy disc	3	0	block	/dev/fl

SEE ALSO

copyfloppy(1), dir(1), zero(1)

NAME

lp – line printer

DESCRIPTION

/dev/lp refers to a DataProducts printer on the DP/Centronics connector. When it is closed, a page eject is generated. Bytes written are printed.

The driver interprets carriage return (0DH), newline (0AH), tab(09H), and form-feed (0CH) characters.

Two *ioctl*(2) system calls are available:

```
#include <sys/lprio.h>
ioctl (fildes, command, arg)
struct lprio *arg;
```

The *commands* are:

LPRGET	Get the current printer parameters and store in the *lprio* structure referenced by **arg**.
LPRSET	Set the current printer parameters from the structure referenced by **arg**.

This allows an external program to control tab expansion, tab size, carriage return expansion, and newline expansion.

DEVICE CODE

device type	*major*	*minor*	*block-type or character-type*	*standard file name*
DataProducts lp	4	0	char	/dev/lp

FILES

/dev/lp
/drivers/fdlp

NAME

mouse – pointing device

DESCRIPTION

mouse refers to the pointing or locator device associated with a Ridge monochrome display. Each window in a multi-window environment has an event queue for locator device events (motion and/or button changes), which may be read via the **mouse** device associated with that window.

The input device data is encoded as a sequence of bytes that are placed into the mouse event queue for the currently selected active input window. Each mouse event is translated into 5 bytes, which are queued according to the selected window's flag bits, as described below. The entire event queue is flushed first if there is not enough room left for all 5 bytes, thus insuring that only the most recent events are queued.

The first byte contains the state of the buttons. Bit 0 (least significant) is the right button, bit 1 is the middle button, and bit 2 (more significant) is the left button. The other bits are normally zero, but may contain more button bits if a nonstandard locator device is being used. A bit value of 1 indicates the corresponding button is depressed, and a bit value of 0 indicates the button is released.

The second and third bytes contain the X coordinate, in two's complement representation, high-order byte first. The fourth and fifth bytes contain the Y coordinate in the same format. The coordinate system used for the X,Y position depends on the mode of the currently selected window and the event mode bits in the *wFlags* word for the window.

If the EMLocCoords bit is set, then display device coordinates are used, and the location can be anywhere on the display surface. The X coordinates range from 0 to 1023, and the Y coordinates range from 0 to 799, with the origin in the upper left corner of the Ridge monochrome display *disp*(7).

If the EMLocCoords bit is not set, then only mouse positions within the currently active window are queued. The coordinate system used in this case is translated relative to the location of the window on the display surface, and may be scaled depending on the current mode of the window.

If the window is emulating a Tektronix 4014 terminal, then the X coordinates range from 0 to 1024, and the Y coordinates range from 0 to 780, with the origin in the lower left corner of the window. The display device coordinates of the mouse are appropriately scaled based on the size of the window.

If the window is not in Tektronix 4014 mode, then display device coordinates are used, but are first translated relative to the origin of the window. The X coordinates range from 0 to one less than the width of the window, and the Y coordinates range from 0 to one less than the height of the window, with the origin in the upper left corner of the window.

The mouse input for a window is accessed using the *open*(2) system call, providing the name **mouse** appended to the window pathname, separated from it by a slash "/". For example, to open the mouse input associated with window **window2** on display device number "0", the pathname would be **/dev/disp0/window2/mouse**.

Alternatively, the special name **/dev/mouse** can be used by a process to specify the mouse input from the window which corresponds to its control terminal. This special name is mapped by ROS to the actual pathname of the window plus the string **/mouse**. Thus, processes may easily refer to the mouse input from their control window without having to know the actual window pathname.

A process uses the standard *read*(2) system call to access the mouse input from the window associated with an open file descriptor. Each read will return as many bytes from the event queue as are requested, although it is recommended that 5 bytes be read at a time, to maintain synchronization with the queuing of the mouse event bytes.

Normally, a process will block when it attempts to read from a window that has no mouse input queued for it, or if the window is not the active input window. A non-blocking *read*(2) may be used to determine if there is any mouse input queued for a particular window. If the O_NDELAY bit is set either by *open*(2) or *fcntl*(2) on a file descriptor bound to the mouse input for a window, then a read will return immediately with a value of 0 when there are no mouse input bytes queued for the window.

Several of the bits in the *wFlags* word associated with each window provide control over the queueing of input events from the mouse.

Mouse pointing device (locator) input is placed in the mouse event queue for the currently selected active input window according to the following bits, as defined in the **<sys/winctrl.h>** header file. If the bits for the currently selected window do not allow queueing of a mouse event, then the window associated with the Window Manager process is tested according to its bits. This allows a process in the currently selected window to filter some or all mouse events, passing unwanted events to the Window Manager, which may choose to handle the mouse event or have it discarded.

EMQueueLoc
: Bit that determines if mouse input is to be queued for the window. The initial value of this bit is 0, causing all mouse events for the window to be ignored. If this bit is set to 1, then mouse events which satisfy the conditions specified by the other control bits will be queued.

EMLocCoords
: Bit that determines the allowable coordinates of the mouse which will be queued for the window. The initial value of this bit is 0, allowing only mouse events whose coordinates are inside the window's boundaries to be queued. If this bit is set to 1, then mouse events located anywhere on the screen which satisfy the other conditions will be queued.

EMLocMotion
: Bit that allows queueing of mouse events resulting from motion or button changes. The initial value of this bit is 0, allowing only mouse events that indicate a button change to be queued, thus ignoring events resulting only from motion. If this bit is set to 1, then mouse events will be queued which satisfy the other conditions, regardless of button changes.

EMButtonMask
: Bit field that determines which button changes will cause a mouse input event to be queued for the window. The field is composed of the logical OR of the following four fields, and is initialized to the value 0.

EMRightButton
: Bit that corresponds to the right button on the mouse. A bit value of 1 indicates that a mouse event which has a different state for the right button from the previous event is to be queued. A bit value of 0 indicates that right button changes are to be ignored.

EMMiddleButton
: Bit that corresponds to the middle button on the mouse. A bit value of 1 indicates that a mouse event which has a different state for the middle button from the previous event is to be queued. A bit value of 0 indicates that middle button changes are to be ignored.

EMLeftButton
: Bit that corresponds to the left button on the mouse. A bit value of 1 indicates that a mouse event which has a different state for the left button from the previous event is to be queued. A bit value of 0 indicates that left button changes are to be ignored.

EMOtherButtons
: Bit that corresponds to all other buttons on a nonstandard mouse or other pointing device. A bit value of 1 indicates that a mouse event which has a different state for at least one of the other buttons from the previous event is to be queued. A bit value of 0 indicates that all other button changes are to be ignored.

EMSigIOLoc
: Bit that determines if mouse input events cause a SIGIO *signal*(2) to be sent to the process or process group waiting for mouse input from the window. The initial value of this bit is 0, causing no signals to be sent. If this bit is set to 1, then each mouse event (not each byte) for the window generates a signal to the processes associated by *fcntl*(2) with the window.

The *wFlags* bits that control mouse input events may be accessed by making an *ioctl*(2) system call using a file descriptor associated with a window.

```
#include <termio.h>

ioctl (filedesc, command, arg)
int *arg;
```

The *command* argument using this form is:

MOUSEGET
: Get the flag bits associated with the mouse input into the integer pointed to by *arg*. All other bits are cleared to zero.

```
ioctl (filedesc, command, arg)
int arg;
```

The *command* argument using this form is:

MOUSESET
: Set the flag bits associated with the mouse input from *arg*. All other bits are ignored.

FILES

/dev/mouse

SEE ALSO

Ridge Multi-Window Display Management Guide
windows(3X), wgraf(3X), disp(7)

NAME

mt – magnetic tape interface files

DESCRIPTION

Mt0 through *mt31*, and *rmt0* through *rmt31* refer to the magnetic tape drive in non-raw and raw modes, respectively.

When opened for reading or writing, the tape is assumed to be positioned as desired. When a file is closed, a double end-of-file (double tape mark) is written if the file was opened for writing. If the file was normal-rewind, the tape is rewound. If it is no-rewind and the file was open for writing, the tape is positioned before the second EOF just written. If the file was no-rewind and opened read-only, the tape is positioned after the EOF following the data just read. Once opened, reading is restricted to between the position when opened and the next EOF or the last write. The EOF is returned as a zero-length read. By judiciously choosing **mt** files, it is possible to read and write multi-file tapes.

A standard tape consists of a series of 4096-byte records terminated by an EOF. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time (although very inadvisable).

The **mt** files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the "raw" interface is appropriate. The associated files are named **rmt0, ..., rmt31**. Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, up to the buffer size specified. Seeks are ignored. An EOF is returned as a zero-length read, with the tape positioned after the EOF, so that the next read will return the next record.

The following definitions of ioctl operations are from **sys/mtio.h**:

```
/* structure for mag tape op command */

  struct  mtop{
  int      mt_op;      /* operations defined below */
  daddr_t  mt_count;   /* how many of them */
  };

/* IOCTL operations */

   #define   MTWEOF 0  /* write an end of file record */
   #define   MTFSF  1  /* forward space file */
   #define   MTBSF  2  /* backward space file */
   #define   MTFSR  3  /* forward space record */
   #define   MTBSR  4  /* backward space record */
   #define   MTREW  5  /* rewind */
   #define   MTOFFL 6  /* rewind and put the drive offline */
   #define   MTNOP  7  /* no operation, sets status only */
   #define   MTHSPD 8  /* set high speed */
   #define   MTLSPD 9  /* set low speed */

/* structure for mag tape get status command */
```

```
struct  mtget{
int      mt_type;     /* type of magtape device */
int      mt_dsreg;    /* drive status register */
int      mt_erreg;    /* error register */
int      mt_resid;    /* residual count */
daddr_t  mt_fileno;   /* file number of current position */
daddr_t  mt_blkno;    /* block number of current position */
};

/*
 * bits of the drive status register
 *
 * 0-7            - Drive Identification
 * 8-11           - (undefined)
 * 12     BCO  =    - Byte Count Overflow
 * 13     TPE  =    - Tape Parity Error
 * 14     DMAE =    - DMA Error
 * 15     OUR  =    - Over/Under Run
 * 16     CIP       - Command In Progress
 * 17     FMK  =    - Filemark
 * 18     HER  =    - Hard Error
 * 19     CER  =    - Corrected Error
 * 20     IDENT=    - PE Identification
 * 21     P4/26     - Undefined Signal Connected To P4-26
 * 22     P3/44     - Undefined Signal Connected To P3-14
 * 23     HISP      - High Speed
 * 24     RWD       - Rewind
 * 25     EOT       - End Of Tape
 * 26     LPT       - Load Point
 * 27     FPT       - File Protect
 * 28     ONL       - On Line
 * 29     RDY       - Ready
 * 30     FBSY      - Formatter Ready
 * 31     DBSY      - Data Busy
 *
 *          = Cleared with next command. */

/*
 * constants for mt_type byte */

  #define MT_ISTS    0x01    }
  #define MT_ISHT    0x02    }
  #define MT_ISTM    0x03    } for compatibility with 4.2 bsd.
  #define MT_ISMT    0x04    } only; not available on Ridge
  #define MT_ISUT    0x05    }
  #define MT_ISCPC   0x06    }
  #define MT_ISAR    0x07    }
  #define MT_ISCY    0x08  /* Cipher streaming tape drive */

/* structure of IOCTL call for magnetic tape operations */

  ioctl (fildes, opcode, opstruct)
```

```
int fildes              /* file descriptor of tape device */
int opcode               /* a tape operation defined above */
struct mtop *opstruct;  /* pointer to structure for */
                       /* operations except MTNOP */
  or
struct mtget *opstruct;  /* pointer to MTNOP structure only */
```

DEVICE CODE

64 device files per controller
major device number: 9
minor device number is calculated by:

```
(unsigned character byte)
bits 0  1  2  3 |4  5  6  7
-----------------+----------
     0  0  0  0 |       0 buffered (non-raw) device
con-  0  0  0  1 |       1 raw device
trol- 0  0  1  0 |    0   no rewind on close
er #1  0  0  1  1 |    1   rewind on close
     0  1  0  0 |0  0      density 800 bpi
     0  1  0  1 |0  1      density 1600 bpi
     0  1  1  0 |1  0      density 3200 bpi
     0  1  1  1 |1  1      density 6250 bpi (not supported)
     1  0  0  0 |       0 buffered (non-raw) device
     1  0  0  1 |       1 raw device
con-  1  0  1  0 |    0   no rewind on close
trol- 1  0  1  1 |    1   rewind on close
ler #2 1  1  0  0 |0  0      density 800 bpi
     1  1  0  1 |0  1      density 1600 bpi
     1  1  1  0 |1  0      density 3200 bpi
     1  1  1  1 |1  1      density 6250 bpi (not supported)
```

FILES

/dev/mt??
/dev/rmt??
/usr/include/sys/mtio.h

32 files for raw mode					32 files for non-raw mode				
minor no.	bytes /inch	re-wind	unit no.	name	minor no.	bytes /inch	re-wind	unit no.	name
5	1600	n	0	rmt0	4	1600	n	0	mt0
7	1600	y	0	rmt1	6	1600	y	0	mt1
9	3200	n	0	rmt16	8	3200	n	0	mt16
11	3200	y	0	rmt17	10	3200	y	0	mt17
21	1600	n	1	rmt2	20	1600	n	1	mt2
23	1600	y	1	rmt3	22	1600	y	1	mt3
25	3200	n	1	rmt18	24	3200	n	1	mt18
27	3200	y	1	rmt19	26	3200	y	1	mt19
37	1600	n	2	rmt4	36	1600	n	2	mt4
39	1600	y	2	rmt5	48	1600	y	2	mt5
41	3200	n	2	rmt20	40	3200	n	2	mt20

43	3200	y	2	rmt21	42	3200	y	2	mt21
53	1600	n	3	rmt6	52	1600	n	3	mt6
55	1600	y	3	rmt7	54	1600	y	3	mt7
57	3200	n	3	rmt22	56	3200	n	3	mt22
59	3200	y	3	rmt23	58	3200	y	3	mt23
69	1600	n	4	rmt8	68	1600	n	4	mt8
71	1600	y	4	rmt9	70	1600	y	4	mt9
73	3200	n	4	rmt24	72	3200	n	4	mt24
75	3200	y	4	rmt25	74	3200	y	4	mt25
85	1600	n	5	rmt10	84	1600	n	5	mt10
87	1600	y	5	rmt11	86	1600	y	5	mt11
89	3200	n	5	rmt26	88	3200	n	5	mt26
91	3200	y	5	rmt27	90	3200	y	5	mt27
101	1600	n	6	rmt12	100	1600	n	6	mt12
103	1600	y	6	rmt13	102	1600	y	6	mt13
105	3200	n	6	rmt28	104	3200	n	6	mt28
107	3200	y	6	rmt29	106	3200	y	6	mt29
117	1600	n	7	rmt14	116	1600	n	7	mt14
119	1600	y	7	rmt15	118	1600	y	7	mt15
121	3200	n	7	rmt30	120	3200	n	7	mt30
123	3200	y	7	rmt31	122	3200	y	7	mt31

NAME

null – the null device

DESCRIPTION

/dev/null refers to the null device. Data written to the null device is discarded. If the null device is read, end-of-file status is returned (but no data).

DEVICE CODE

device type	*major*	*minor*	*block-type or character-type*	*standard file name*
null device	1	2	char	/dev/null

FILES

/dev/null

NAME

termio – general terminal interface

TERMIO(7) IS NOT A FILE. THIS EXPLAINS TERMINAL I/O CHARACTERISTICS.

DESCRIPTION

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by **getty** and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the **control terminal** for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a fork(2). A process can break this association by changing its process group using setpgrp(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 buffer reaches 4352 characters, which is rare, or when the user has accumulated 4352 input characters that have not yet been read by some program. When the input limit is reached, the oldest character is thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (ASCII EOT) character, or a user-defineable end-of-line character. 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 necessary, however, to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, **control-h** erases the last character typed, except that it will not erase beyond the beginning of the line. By default, **control-x** kills (deletes) the entire input line. Both these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. The erase and kill characters may be changed.

Certain characters have special functions on input:

INTR	(Rubout or ASCII DEL) generates an *interrupt* signal which is sent to all processes associated with the 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)*.
QUIT	(Control-| or ASCII FS) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will be placed under control of the debugger.
ERASE	(Control-h) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.
KILL	(Control-x) deletes the entire line, as delimited by a NL, EOF, or EOL character.
EOF	(Control-d or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is processed on the next read request. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.

NL	(ASCII LF) is the normal line delimiter. It can not be changed or escaped.
EOL	(ASCII NUL) is an additional line delimiter, like NL. It is not normally used.
STOP	(Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.
START	(Control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL may be changed to suit individual tastes.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue when input is requested by a read on the terminal file. If 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.

Several *ioctl(2)* system calls apply to terminal files. The primary calls use the following structure, defined in **<termio.h>**:

```
#define  NCC  8
struct termio {
  int     c_version;  /* termio structure version */
  int     c_iflag;    /* input modes */
  int     c_oflag;    /* output modes   */
  int     c_cflag;    /* control modes */
  int     c_lflag;    /* line   discipline modes */
  int     c_line;     /* line   discipline */
  int     c_numchr;   /* number of control chars following */
  unsigned char   c_cc[NCC];  /* control chars */
};
```

The special control characters are defined by the array **c_cc**. The relative positions and initial values for each function are as follows:

0	INTR	DEL
1	QUIT	FS
2	ERASE	**control-h**
3	KILL	**control-x**
4	EOF	EOT
5	EOL	NUL
6	JOB	**control-z**
7	reserved	

The *c_iflag* field describes the basic terminal input control:

IGNBRK	0000001	Ignore break condition.
BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
PARMRK	0000010	Mark parity errors.
INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.
INLCR	0000100	Map NL to CR on input.
IGNCR	0000200	Ignore CR.

ICRNL	0000400	Map CR to NL on input.
IUCLC	0001000	Map upper-case to lower-case on input.
IXON	0002000	Enable start/stop output control.
IXANY	0004000	Enable any character to restart output.
IXOFF	0010000	Enable start/stop input control.

If IGNBRK is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if BRKINT is set, the break condition will generate an interrupt signal and flush both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

If PARMRK is set, a character with a framing or parity error which is not ignored is read as the three character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error which is not ignored is read as the character NUL (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper-case alphabetic character is translated into the corresponding lower-case character.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read.

If IXANY is set, any input character, will restart output which has been suspended. IXANY is NOT implemented.

If IXOFF is set, the system will transmit START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all bits clear.

The *c_oflag* field specifies the system treatment of output:

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lower case to upper on output.
ONLCR	0000004	Map NL to CR-NL on output.
OCRNL	0000010	Map CR to NL on output.
ONOCR	0000020	No CR output at column 0.
ONLRET	0000040	NL performs CR function.

OFILL	0000100	Use fill characters for delay.
OFDEL	0000200	Fill is DEL, else NUL.
NLDLY	0000400	Select new-line delays:
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays:
TAB0	0	
TAB1	0004000	
TAB2	0010000	
TAB3	0014000	Expand tabs to spaces.
BSDLY	0020000	Select backspace delays:
BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	
FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

ONLY THE FOLLOWING ARE IMPLEMENTED:
OPOST, OLCUC, ONLCR, OCRNL, ONLRET, ONOCR, TAB3

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position).

If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

Horizontal-tab delay type 3, the only delay bit implemented, specifies that tabs are to be expanded into spaces.

The initial output control value is all bits clear.

The *c_cflag* field (WHICH IS IGNORED BY THE RIDGE DISPLAY) describes the hardware control of the terminal:

CBAUD	0000037	Baud rate:
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud

B300	0000007	300 baud
B600	0000010	600 baud
B1200	0000011	1200 baud
B1800	0000012	1800 baud
B2400	0000013	2400 baud
B4800	0000014	4800 baud
B9600	0000015	9600 baud
EXTA	0000016	External A
EXTB	0000017	External B
B19200	0000020	19200 baud
CSIZE	0000140	Character size:
CS5	0	5 bits
CS6	0000040	6 bits
CS7	0000100	7 bits
CS8	0000140	8 bits
CSTOPB	0000100	Send two stop bits, else one.
CREAD	0000400	Enable receiver.
PARENB	0001000	Parity enable.
PARODD	0002000	Odd parity, else even.
HUPCL	0004000	Hang up on last close.
CLOCAL	0010000	Local line, else dial-up.
CAUTO	0020000	Enable RS-232 hardware flow-control.

The CBAUD bits specify the baud rate. For any particular hardware, impossible speed changes are ignored.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stops bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used.

If CREAD is set, the receiver is enabled. Otherwise no characters will be received.

If HUPCL is set, the line will be disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. Otherwise modem control is assumed.

If CAUTO is set, hardware flow-control for RS-232 terminals is enabled. This feature prevents too-rapid input/output with a peripheral device. If enabled, the RS-232 receiver and transmitter are switched on and off by the "carrier detect" (DCD) and the "clear to send" (CTS) lines, respectively. No data will be received unless DCD (pin 8) is true, and no data will be transmitted until CTS (pin 5) is true.

The initial hardware control value after open is B9600, CS7 CREAD, PARENB. CLOCAL.

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	0000001	Enable signals.
ICANON	0000002	Canonical input (erase and kill processing).

XCASE	0000004	*Canonical upper/lower presentation.
ECHO	0000010	Enable echo.
ECHOE	0000020	Echo erase character as BS-SP-BS.
ECHOK	0000040	Echo NL after kill character.
NOFLSH	0000200	Disable flush after interrupt or quit.
LINTRUP	0000400	Generate signal SIGIO when data is available.

* IGNORED BY RIDGE DISPLAY

If ISIG is set, each input character is checked against the special control characters INTR, JOB, and QUIT. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g. 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read will not be satisfied until at least MIN characters have been received. This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN value is stored in the position for the EOF character. The time value represents tenths of seconds.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which will clear the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character will be echoed after the kill character to emphasize that the line will be deleted, otherwise a series of backspace- space- backspace sequences are output to cause the line to be erased. Note that an escape character preceding the erase or kill character removes any special function. Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit and interrupt characters will not be done.

The initial line-discipline control value is all bits clear.

The primary *ioctl(2)* system calls have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETA	Get the parameters associated with the terminal and store in the *termio* structure referenced by **arg**.
TCSETA	Set the parameters associated with the terminal from the structure referenced by **arg**. The change is immediate.
TCSETAW	Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.
TCSETAF	Wait for the output to drain, then flush the input queue and set the new parameters.

Additional *ioctl(2)* calls have the form:

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

The commands using this form are:

TCXONC	Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.
TCFLSH	If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

SEE ALSO

stty(1), ioctl(2).

NAME

tty – controlling terminal interface

DESCRIPTION

/dev/tty0 through **/dev/tty7** refer to RS-232 terminals attached to the J connectors on the Ridge 32 back panel.

/dev/tty refers to the current user's terminal in use, which may be **tty0** through **tty7**, or **disp0** through **disp3**. It is useful for programs or shell sequences 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 the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

DEVICE CODE

device type	*major*	*minor*	*block-type or character-type*	*standard file name*
current user terminal	1	1	char	/dev/tty
RS-232 ports	2	0..7	char	/dev/tty0..tty7
monochrome disp	6	0..3	block	/dev/disp0..disp3

FILES

/dev/tty	this terminal
/dev/tty0	(J1 on Ridge back panel)
/dev/tty1	(J2 on Ridge back panel)
/dev/tty2	(J3 on RIdge back panel)
/dev/tty3	(J4 on Ridge back panel)
/dev/tty4	
/dev/tty5	
/dev/tty6	
/dev/tty7	

/dev/disp0
/dev/disp1
/dev/disp2
/dev/disp3

/drivers/disp	device driver for disp's
/drivers/fdlp	device driver for tty's

NAME

vp – Versatec printer/plotter in print mode

DESCRIPTION

/dev/vp refers to Versatec printer/plotter in print mode on the Versatec printer/plotter connector.

When it is closed, a page eject is generated. Bytes written are printed.

The driver interprets carriage return (0DH), newline (0AH), tab (09H), and form-feed (0CH) characters.

Two ioctl(2) system calls are available:

```
#include <sys/lprio.h>
ioctl (fildes, command, arg)
struct lprio *arg;
```

The *commands* are:

LPRGET	Get the current printer parameters and store in the *lprio* structure referenced by **arg**.
LPRSET	Set the current printer parameters from the structure referenced by **arg**.

This allows an external program to control tab expansion, tab size, carriage return expansion, and newline expansion.

DEVICE CODE

			block-type or	*standard*
device type	*major*	*minor*	*character-type*	*file name*
Versatec printer	4	32	character	/dev/vp

FILES

/dev/vp
/dev/fdlp

NAME

vplot – Versatec printer/plotter in plot mode

DESCRIPTION

/dev/vplot refers to Versatec printer/plotter in plot mode on the Versatec printer/plotter connector.

When it is closed, bits written are printed and a page-eject is generated.

Two *ioctl*(2) system calls are available:

```
#include <sys/vprio.h>
ioctl (fildes, command, arg)
struct vprio *arg;
```

The *commands* are:

VPPRINT	Put the Versatec into print mode.
VPPLOT	Put the Versatec into plot mode.

DEVICE CODE

device type	*major*	*minor*	*block-type or character-type*	*standard file name*
Versatec in plot mode	5	0	character	/dev/vplot

FILES

/dev/vplot
/drivers/fdlp

Ridge Computers
Corporate Headquarters

2451 Mission College Blvd.
Santa Clara, California 95054
Phone: (408) 986-8500
Telex: 176956