ConvexOS Man Pages for Programmers

Second Edition

CONVEX COMPUTER CORPORATION

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

ConvexOS Man Pages for Programmers

Order No. DSW-332

Second Edition
December 1991

CONVEX Press
Richardson, Texas USA

ConvexOS Man Pages for Programmers

Order No. DSW-332

Printed in the United States of America

Revision information for

ConvexOS Man Pages for Programmers

Edition	Document No.	Description
Second	710-015830-001	Released with ConvexOS V10.0, December 1991.
First	710-015830-000	Initial release, April 1991.

Contents

Section 3 Library functions

Section 4 General utilities/driver functions/networking support

Section 5 File formats

Index

Using ConvexOS man pages

What are man pages?

The ConvexOS man pages, online and hardcopy, are a reference for all of the commands available to ConvexOS users. Each command has its own "man page," which can be one or more pages in length.

The ConvexOS man pages are organized into 7 sections; commands for similar functions are grouped together. This book contains four of those sections, Section 2, Section 3, Section 4, and Section 5. (Traditionally, Section 6 of the man pages is allotted to games. Because CONVEX does not sell games, the ConvexOS man pages do not contain Section 6.)

How is this book organized?

This book is divided into the following sections:

- Section 2 man pages—Describe system calls and error numbers
- Section 3 man pages—Describe various library functions
- Section 4 man pages—Describe special files, related driver functions, and networking support
- Section 5 man pages—Describe the format of various files

Where to find other ConvexOS man pages

ConvexOS Man Pages for Users contains Section 1 and Section 7 of the ConvexOS man pages. These sections describe:

- Section 1 man pages—Commands of general utility and commands for communication with other systems.
- Section 7 man pages—Miscellaneous commands, primarily in the areas of text processing and terminal environments.

Section 8 man pages are found in *ConvexOS Man Pages for System Managers*. The section 8 man pages describe various system management tools.

How to use ConvexOS man pages

The ConvexOS man pages are available in two formats: the hardcopy man pages contained in this book and the online man pages you can access by using the ConvexOS `man` command.

Using hard copy man pages

The ConvexOS man pages contained in this book are divided into Section 3, Section 4, Section 5, and Section 6, and are ordered alphabetically within each section.

Using online man pages

`man` is a program that formats and displays information from the hard copy man pages. When invoked in the simplest way, without any options and without a topic, it displays the corresponding manual page formatted with `nroff`. Access online man pages by entering

```
% man entry-name
```

where *entryname* is the name of the man page to be displayed. You can print hard copies of online man pages by entering

```
% man -t
```

For more information on `man` options, refer to the man(1) man page.

Format of man pages

The contents of each man page is divided into subsections (not all of which are relevant to each entry):

NAME	Lists exact name of command or subroutine and describes its purpose
SYNOPSIS	Summarizes use of command or subroutine being described
DESCRIPTION	Discusses use of command or subroutine
FILES	Names files built into the program
SEE ALSO/REFERENCES	Points to related information
DIAGNOSTICS/ERRORS	Discusses diagnostic messages the system produces
BUGS/NOTES	Explains known bugs or deficiencies and any known solutions

When you can't find the man page you need

One man page can document several commands. For instance, if you enter

```
% man moncontrol
```

the moncontrol(3) man page appears on your screen. However, there is no separate hard copy man page for `moncontrol`. If you look it up in the index, you are referred to the monitor(3) man page, which describes both the `moncontrol` and `monitor` commands, among others.

If you cannot find a hard copy man page for a specific command, look up that command in the index—it will refer you to the man page you need.

Technical assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC).

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384.
- All other locations, contact the local CONVEX office.

Section 2
System calls/error numbers

NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible return value. This is almost always **–1**; the individual descriptions specify the details.

As with normal arguments, all return codes and values from functions are of type integer unless otherwise noted. An error number is also made available in the external variable *errno*, which is not cleared on successful calls. Thus, *errno* should be tested only after an error has occurred.

The subroutine *perror*(3) produces a short error message on the standard error output describing the type of error that has occurred. We strongly recommend comprehensive usage of this routine upon error detection in system call returns.

The following is a complete list of the errors and their names as given in *<errno.h>*.

0 Errorno is zero
Unused.

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

2 ENOENT No such file or directory
This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a pathname does not exist. For programs built in the default (POSIX) environment, it also occurs when a pointer to an empty string is passed to system calls which expect a pathname argument.

3 ESRCH No such process
The process whose number was given to *kill*(2) or *pattach*(2) does not exist, or is already dead.

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

5 EIO I/O error
Some physical I/O error occurred during a *read* or *write*. This error may, in some cases, occur on a call following the one to which it actually applies.

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

7 E2BIG Arg list too long
An argument list longer than 10240 bytes is presented to *execve*.

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number (see *a.out*(5)).

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

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

11 EAGAIN Resources temporarily unavailable
In a *fork*, the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Insufficient free swap space
During an *execve* or *break*, a program asks for more core or swap space than the system is able to supply. A lack of swap space is normally a temporary condition, however a lack of core is not a temporary condition; the maximum size of the text, data, and stack segments is a system parameter.

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 an invalid address in attempting to access the arguments of a system call.

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

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

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

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

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

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

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

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

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

24 EMFILE Too many open files
Customary configuration limit is 256 per process.

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

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

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

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

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

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

31 EMLINK Too many links
An attempt to make more than 32767 hard links to a file.

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

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

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

35 EWOULDBLOCK Operation would block
An operation which would cause a process to block was attempted on a object in non-blocking mode (see *ioctl*(2)). Note that many system calls which formerly returned this error now return EAGAIN when called from a POSIX mode program.

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

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

38 ENOTSOCK Socket operation on non-socket
Self-explanatory.

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

40 EMSGSIZE Message too long
A message sent on a socket was larger than the internal message buffer.

41 EPROTOTYPE Protocol wrong type for socket
A protocol was specified which does not support the semantics of the socket type requested. For example you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

42 ENOPROTOOPT Protocol not available
A bad option was specified in a *getsockopt*(2) or *setsockopt*(2) call.

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

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

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

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

47 EAFNOSUPPORT Address family not supported by protocol family
An address incompatible with the requested protocol was used. For example, you should not necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

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

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

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

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

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

53 ECONNABORTED Software caused connection abort
A connection abort was caused internal to your host machine.

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

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

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

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

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

59 ETOOMANYREFS Too many references: can't splice

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

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

62 ELOOP Too many levels of symbolic links
A pathname lookup involved more than 8 symbolic links.

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

64 EHOSTDOWN Host is down

65 EHOSTUNREACH Host is unreachable

66 ENOTEMPTY Directory not empty
A directory with entries other than . and .. was supplied to a remove directory or rename call.

67 EPROCLIM Too many processes

68 EUSERS Too many users

69 EDQUOT Disc quota exceeded

70 ESTALE Stale NFS file handle
A client referenced an open file, when the file has been deleted.

71 EREMOTE Too many levels of remote in path
An unsupported operation was attempted on a file within an NFS filesystem, such as attempting to get the file's "file handle" (see *getfh*(2)).

72 EDEADLK Deadlock situation detected/avoided
The *lockf*(3) library frontend routine returns this if the *fcntl*(2) system call returns the ENOLCK error for compatibility with */usr/group* standards.

73 ENOLCK No record locks available
The record lock daemon process *lockd*(8C) has not been started or all record lock resources have been consumed.

74 ENOSYS Function not implemented

75 ETPTRUNC Logical tape record would be truncated
A particular *write*(2) to a labeled tape would result in a logical record being truncated because its length is greater than the maximum record size.

76 ETPBADFORM Labeled tape not in proper format
A label or tapemark was not found where it was expected. Continuing reading after receiving this error may give unpredictable results.

77 ETPNOSUPPORT Tape feature not supported
A perhaps valid tape feature (such as a binary-coded variable record format (V)) was discovered which is not supported on this system.

114 ENOLINK The link has been severed
One or more of the Streams drivers in a stream have been unlinked.

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 30000.

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 is the process ID of the group leader. This grouping permits the signalling of related processes (see *killpg*(2)) and the job control mechanisms of *csh*(1).

Sessions
Each process group is a member of a session; a process is considered a member of the session of which its process group is a member. A new process (see *fork*(2)) joins the session of its parent. A process can use *setsid*(2)) to alter its session membership; it thus becomes a session leader, and as such, may cause a terminal to become its controlling terminal (see *tty*(4)).

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 arbitrate between multiple jobs contending for the same terminal (see *csh*(1) and *tty*(4)).

Real User ID and Real Group ID
Each user on the system is identified by a positive integer termed the real user ID.

Each user is also a member of one or more groups. One of these groups is distinguished from others and used in implementing accounting facilities. The positive integer corresponding to this distinguished group is termed the real group ID.

All processes have a real user ID and a real group ID. These are initialized from the equivalent attributes of the process which created it.

Effective User ID, Effective Group ID, and Access Groups
Access to system resources is governed by three values: the effective user ID, the effective group ID, and the group access list.

The effective user ID and effective group ID are initially the process' real user ID and real group ID, respectively. Either may be modified through execution of a set-user-ID or set-group-ID file, possibly by one of its ancestors (see *execve*(2)).

The group access list is an additional set of group ID's used only in determining resource accessibility. Access checks are performed as described below in **File Access Permissions**

Activity ID
: Each process has an activity ID associated with it. Upon login, activity ID is initialized to 0; subsequently, each process inherits the activity ID of its parent. The activity ID provides a means by which system managers may design customized billing categories for use in conjunction with groups. Various user programs record activity ID in their log files, resulting in a global, yet flexible, resource accounting system. The activity ID is set using the system call *setaid*(2), which is available only to the superuser. (See *bill*(1) for information about how a user may set his activity ID without being the superuser; see also *activities*(5)).

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

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

Descriptor
: An integer assigned by the system when a file is referenced by *open*(2), *dup*(2), or *pipe*(2) or a socket is referenced by *socket*(2) or *socketpair*(2), which uniquely identifies an access path to that file or socket from a given process or any of its children.

Filename
: Names consisting of up to {FILENAME_MAX} characters may be used to name an ordinary file, special file, or directory.

  These characters may be selected from the set of all ASCII character excluding **0** (null) and the ASCII code for /(slash). (The parity bit, bit 8, must be 0.)

  Note that it is generally unwise to use *, !, ?, [, or] as part of filenames because of the special meaning attached to these characters by the shell.

Path Name
: 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. The total length of a path name must be less than {PATHNAME_MAX} characters.

  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. For POSIX mode programs, an empty pathname is an error; in backward compatible (*-pcc*) mode, an empty pathname refers to the current directory.

Directory
: A directory is a special type of file that contains entries that are references to other files. 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' root directory need not be the root directory of the root file system.

File Access Permissions
: Every file in the file system has a set of access permissions. These permissions are used in determining whether a process may perform a requested operation on the file (such as opening a file for writing). Access permissions are established when a file is created. They may be changed at some later time through the *chmod*(2) call.

  File access is broken down according to whether a file may be: read, written, or executed. Directory files use the execute permission to control if the directory may be searched.

  File access permissions are interpreted by the system as they apply to three different classes of users: the owner of the file, users in the file's group, and anyone else. Every file has an

independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if the process' effective user ID is that of the superuser. For non-superusers, additional access checks are made. If the process' effective user ID matches that of the owner of the file, permission is denied unless the owner permissions allow the access. If the process' effective group ID matches the group ID of the file, permission is denied unless the group permissions allow the access. If the group ID of any group to which the process belongs (see *getgroups(2)*) matches the group ID of the file, permission is denied unless the group permissions allow the access. If neither the user ID nor any of the group IDs matches, permission is denied unless the permissions for "other users" allow access.

Sockets and Address Families

A socket is an endpoint for communication between processes. Each socket has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format used in naming message recipients, etc.

Each instance of the system supports some collection of socket types (see *socket*(2) for more information about the types available and their properties).

Each instance of the system supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

SEE ALSO

intro(3), perror(3)

NAME

accept – accept a connection on a socket

SYNOPSIS

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

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

DESCRIPTION

The argument *s* is a socket that has been created with *socket*(2), bound to an address with *bind*(2), and is listening for connections after a *listen*(2). *Accept* extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *accept* blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, *accept* returns an error as described below. The accepted socket, *ns*, may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to *select*(2) a socket for the purposes of doing an *accept* by selecting it for read.

RETURN VALUE

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

ERRORS

The *accept* will fail if:

[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked non-blocking and no connections are present to be accepted.

SEE ALSO

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

NAME

access – determine accessibility of file

SYNOPSIS

```
#include <unistd.h>

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

DESCRIPTION

access() checks the given file *path* for accessibility according to *mode*, which is an inclusive or of the bits R_OK, W_OK and X_OK. Specifying *mode* as F_OK (i.e. 0) tests whether the directories leading to the file can be searched and the file exists.

The real user ID and the group access list (including the real group ID) are used in verifying permission, so this call is useful to set-UID programs.

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

RETURN VALUE

If *path* cannot be found or if any of the desired access modes would not be granted, then a -1 value is returned; otherwise a 0 value is returned.

ERRORS

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

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	The argument path name was too long.
[ENOENT]	The *path* argument points to an empty string.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name or the named file does not exist.
[EINVAL]	The *path* argument contains a byte with the high-order bit set.
[EINVAL]	The *mode* argument contains a bit other than a combination of F_OK, R_OK, W_OK, and X_OK.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	Write access is requested for a file on a read-only file system.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file that is being executed.
[EACCES]	Permission bits of the file mode do not permit the requested access; or search permission is denied on a component of the path prefix. The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits, members of the file's group other than the owner have permission checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.
[EFAULT]	*path* points outside the process's allocated address space.

NOTES

Remote accesses to local files using the *access* system call cannot be detected by the *faillogon* utility.

BACKWARD COMPATIBILITY

Previous versions of the operating system accepted an empty *path* string as a synonym for the current directory.

SEE ALSO

chmod(2), stat(2), faillogon(8)

NAME

acct – turn accounting on or off

SYNOPSIS

acct(file)
char *file;

DESCRIPTION

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

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

If accounting is already on when accounting is turned on, the accounting log will be atomically switched to the new file.

This call is permitted only to the super-user.

NOTES

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

RETURN VALUE

On error –1 is returned. The file must exist and the call may be exercised only by the super-user.

ERRORS

Acct will fail if one of the following is true:

[EPERM]	The caller is not the super-user.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*File* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EACCES]	The file is a character or block special file.

SEE ALSO

acct(5), sa(8)

BUGS

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

NAME

adjtime – adjust time

SYNOPSIS

#include <sys/time.h>

adjtime(tp, otp)
struct timeval *tp, *otp;

DESCRIPTION

adjtime adjusts the system's notion of the current time. The time is adjusted by the amount of time in *tp. The old adjustment value is returned in *otp.

The adjustment is effected by speeding up or slowing down the system's clock by a small amount. This amount is contained in the tunable variable *tickadj*. The default value for *tickadj* is 5; the minimum is 1 and the maximum is 1000. Changing *tickadj* by 1 corresponds to a 0.01% change in clock speed. For example, a value of 100 would cause the clock to run 1% fast or slow.

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

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

If *otp* is a zero pointer, the corresponding information will not be returned.

Only the superuser may adjust the time of day.

The adjustment value will be silently rounded to the resolution of the system clock.

RETURN

A 0 return value indicates that the call succeeded. A –1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] An argument address referenced invalid memory.

[EPERM] A user other than the superuser attempted to set the time.

SEE ALSO

settimeofday(2), date(1)

NAME

asiostat – wait, then return asynchronous I/O byte count

SYNOPSIS

retval = asiostat(*fd*)
int retval, *fd*;

DESCRIPTION

Asiostat first waits until any outstanding asynchronous I/O on file descriptor *fd* has been completed, and then returns the sum of the number of bytes asynchronously transferred to or from the file by the requesting process since the last call to *asiostat*.

RETURN VALUE

If errors were detected during any of the previous asynchronous I/O operations since the last *asiostat* call, a -1 is returned and the global variable *errno* will contain the first error recorded. If no errors were detected, the sum of the number of bytes asynchronously transferred since the last *asiostat* is returned.

ERRORS

For the interpretation of errors, see *read* or *write* as appropriate. *Asiostat* will itself return EBADF for nonvalid file descriptors.

SEE ALSO

fcntl(2), read(2), write(2)

NAME

bind – bind a name to a socket

SYNOPSIS

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

bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

Bind assigns a name to an unnamed socket. When a socket is created with *socket*(2) it exists in a name space (address family) but has no name assigned. *Bind* requests that *name* be assigned to the socket.

NOTES

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using *unlink*(2)). (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

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

RETURN VALUE

If the bind is successful, a 0 value is returned. A return value of –1 indicates an error, which is further specified in the global *errno*.

ERRORS

The *bind* call will fail if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCES]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The *name* parameter is not in a valid part of the user address space.

The following errors are specific to binding names in the UNIX domain.

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	A prefix component of the path name does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EROFS]	The name would reside on a read-only file system.
[EISDIR]	A null pathname was specified.

SEE ALSO

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

NAME

brk, sbrk – change data segment size

SYNOPSIS

caddr_t brk(addr)
caddr_t addr;

caddr_t sbrk(incr)
int incr;

DESCRIPTION

Brk sets the system's idea of the lowest data segment location not used by the program (called the break) to *addr* (rounded up to the next multiple of the system's page size). Locations greater than *addr* rounded up to the next multiple of the system's page size and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

In the alternate function *sbrk*, *incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned. The returned pointer is not page aligned.

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

The *getrlimit*(2) system call may be used to determine the maximum permissible size of the *data* segment; it will not be possible to set the break beyond the *rlim_max* value returned from a call to *getrlimit*, e.g. "etext + rlp→rlim_max." (See *end*(3) for the definition of *etext*.)

RETURN VALUE

Zero is returned if the *brk* could be set; –1 if the program requests more memory than the system limit. *Sbrk* returns –1 if the break could not be set.

ERRORS

Sbrk will fail and no additional memory will be allocated if one of the following are true:

[ENOMEM] The limit, as set by *setrlimit*(2), was exceeded.

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

[ENOMEM] Insufficient space existed in the swap area to support the expansion.

SEE ALSO

execve(2), getrlimit(2), malloc(3), end(3)

BUGS

Setting the break may fail due to a temporary lack of swap space. It is not possible to distinguish this from a failure caused by exceeding the maximum size of the data segment without consulting *getrlimit*.

NAME

chdir – change current working directory

SYNOPSIS

chdir(path)
char *path;

DESCRIPTION

path is the pathname of a directory. *chdir()* causes this directory to become the current working directory, the starting point for path names not beginning with "/."

In order for a directory to become the current directory, a process must have execute (search) access to the directory.

If *chdir()* fails, the current directory is unchanged.

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

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

[ENOTDIR]	A component of the pathname is not a directory.
[ENOENT]	The named directory does not exist, or *path* is an empty string.
[ENAMETOOLONG]	The argument path name was too long.
[EINVAL]	The argument contains a byte with the high-order bit set.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*Path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

BACKWARD COMPATIBILITY

Previous versions of the operating system allowed an empty pathname string as a synonym for the current directory.

SEE ALSO

chroot(2)

NAME

chmod – change mode of file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
chmod(path, mode)
char *path;
mode_t mode;
```

DESCRIPTION

The file whose name is given by *path* has its mode changed to *mode*. Modes are constructed by *or*ing together some combination of the following:

S_ISUID	set effective user ID on execution
S_ISGID	set effective group ID on execution
S_IRWXU	read, write, execute (search) by owner
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute (search on directory) by owner
S_IRWXG	read, write, execute (search) by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute (search on directory) by group
S_IRWXO	read, write, execute (search) by others
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute (search on directory) by others

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

Writing or changing the owner of a file turns off the set-user-ID and set-group-ID bits unless the real or effective user-ID is that of the super-user. Additionally, shell scripts which have either the set-user-ID bit or set-group-ID bit set will not execute if the caller's user/group-ID does not match that of the script. This adds some degree of security at the expense of some compatibility.

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

chmod() will fail and the file mode will be unchanged if:

[EINVAL]	The *path* argument contains a byte with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	The pathname was too long.
[ENOENT]	The named file does not exist, or the *path* argument points to an empty string.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

CONVEX EXTENSIONS

Convex adds the following mode bit.

_S_ISVTX see *sticky*(8)

SEE ALSO

fchmod(2), open(2), chown(2)

NAME

chown – change owner and group of a file

SYNOPSIS

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

chown(path, owner, group)
char *path;
uid_t owner;
gid_t group;
```

DESCRIPTION

The file that is named by *path* has its *owner* and *group* changed as specified. Only the super-user may change the owner of the file, because if users were able to give files away, they could defeat the file-space accounting procedures. The owner of the file may change the group to a group of which he is a member.

On some systems, *chown()* clears the set-user-ID and set-group-ID bits on the file to prevent accidental creation of set-user-ID and set-group-ID programs owned by the super-user.

RETURN VALUE

Zero is returned if the operation was successful; –1 is returned if an error occurs, with a more specific error code being placed in the global variable *errno*.

ERRORS

chown() will fail and the file will be unchanged if:

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EPERM]	The effective user ID is not the super-user and either the file owner was being changed or the effective user ID does not match the file owner or the new group is not a member of the processes group list.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.

CONVEX EXTENSIONS

A value of _SAME_UID passed for owner or _SAME_GID for group results in the current owner or group remaining unchanged.

If the final component of *path* is a symbolic link, the ownership and group of the symbolic link is changed, not the ownership and group of the file or directory to which it points.

BACKWARD COMPATIBILITY

The *owner* and *group* parameters were formerly declared to be type int, although the high word was unusable.

SEE ALSO

fchown(2), chown(8), chgrp(1), chmod(2), flock(2)

NAME

chroot – change root directory

SYNOPSIS

chroot(dirname)
char *dirname;

DESCRIPTION

Dirname is the address of the pathname of a directory, terminated by a null byte. *Chroot* causes this directory to become the root directory, the starting point for path names beginning with "/".

In order for a directory to become the root directory a process must have execute (search) access to the directory.

This call is restricted to the super-user.

RETURN VALUE

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

ERRORS

Chroot will fail and the root directory will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path name is not a directory.
[ENOENT]	The pathname was too long.
[EINVAL]	The argument contains a byte with the high-order bit set.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*Path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

chdir(2)

NAME

close – delete a descriptor

SYNOPSIS

int close(d)
int d;

DESCRIPTION

The *close()* call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, then it will be deactivated. For example, on the last close of a file the current *lseek()* pointer associated with the file is lost; on the last close of a *socket*(2) associated naming information and queued data are discarded; on the last close of a file holding an advisory lock the lock is released.

A close of all of a process's descriptors is automatic on *exit*, but since there is a limit on the number of active descriptors per process, *close* is necessary for programs which deal with many descriptors.

When a process *fork()*'s, all descriptors for the new child process reference the same objects as they did in the parent before the fork. If a new process is then to be run using *execve*(2), the process would normally inherit these descriptors. Most of the descriptors can be rearranged with *dup2*(2) or deleted with *close()* before the *execve()* is attempted, but if some of these descriptors will still be needed if the *execve()* fails, it is necessary to arrange for them to be closed if the *execve()* succeeds. For this reason, the call "fcntl(d, F_SETFD, FD_CLOEXEC)" is provided which arranges that a descriptor will be closed after a successful execve; the call "fcntl(d, F_SETFD, 0)" restores the default, which is to not close the descriptor.

RETURN VALUE

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

ERRORS

close() will fail if:

[EBADF]	*d* is not an active descriptor.
[EINTR]	The close function was interrupted by a signal.

BACKWARD COMPATIBILITY

The EINTR return value was formerly difficult to observe (see *sigvec*(2)). It is now the default, and the use of non-POSIX extensions is required to avoid it (see *sigaction*(2)).

SEE ALSO

accept(2), flock(2), open(2), pipe(2), socket(2), socketpair(2), execve(2), fcntl(2), fork(2)

NAME

connect – initiate a connection on a socket

SYNOPSIS

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

connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;

DESCRIPTION

The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name,* which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way. Generally, stream sockets may successfully *connect* only once; datagram sockets may use *connect* multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUE

If the connection or binding succeeds, then 0 is returned. Otherwise a –1 is returned, and a more specific error code is stored in *errno.*

ERRORS

The call fails if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network isn't reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the process address space.
[EINPROGRESS]	The socket is non-blocking and the connection cannot be completed immediately. It is possible to *select*(2) for completion by selecting the socket for writing.
[EALREADY]	The socket is non-blocking and a previous connection attempt has not yet been completed.

The following errors are specific to connecting names in the UNIX domain. (UNIX is a registered trademark of UNIX System Laboratories, Inc.) These errors may not apply in future versions of the UNIX IPC domain.

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	

A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named socket does not exist.

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

[EACCES] Write access to the named socket is denied.

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

[ENETDOWN] A network on the path to the local host is down.

[EHOSTDOWN] The remote host or a gateway to the remote host is down.

[EHOSTUNREACH]
The remote host is not reachable from this host.

SEE ALSO

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

NAME

creat – create a new file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
creat(name, mode)
char *name;
mode_t mode;
```

DESCRIPTION

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

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

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

RETURN VALUE

The value –1 is returned if an error occurs. Otherwise, the call returns a non-negative descriptor that only permits writing.

ERRORS

creat() will fail and the file will not be created or truncated if one of the following occur:

[EACCES]	A needed directory does not have search permission.
[EACCES]	The file does not exist and the directory in which it is to be created is not writable.
[EACCES]	The file exists, but it is unwritable.
[EEXIST]	O_CREAT and O_EXCL are set, and the file exists.
[EINTR]	The *creat()* operation was interrupted by a signal.
[EISDIR]	The file is a directory.
[EINVAL]	The *path* argument contains a byte with the high-order bit set.
[EMFILE]	There are already too many files open.
[ENAMETOOLONG]	The length of the *path* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.
[ENFILE]	Too many files are currently open on the system.
[ENOENT]	A directory specified in the pathname does not exist.
[ENOENT]	The *path* argument points to the empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The named file resides on a read-only file system.
[ENXIO]	The file is a character special or block special file, and the associated device does not exist.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EFAULT]	*name* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

[EOPNOTSUPP]
The file was a socket (not currently implemented).

BACKWARD COMPATIBILITY

In previous releases of the operating system, the EINTR errno would not occur by default (see *sigvec(2)*).

In previous releases of the operating system, the empty pathname was a synonym for the current directory.

NOTES

The *mode* given is arbitrary; it need not allow writing. This feature has been used in the past by programs to construct a simple exclusive locking mechanism. It is replaced by the O_EXCL open mode, or *flock*(2) facility.

creat() is equivalent to

open(*path*,O_WRONLY|O_CREAT|O_TRUNC,*mode*);

SEE ALSO

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

NAME

cvxprusage – get information about parallel resource utilization

SYNOPSIS

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

cvxprusage(prusage)
struct cvxprusage *prusage;
```

DESCRIPTION

Cvxprusage returns information describing the resources utilized by the current process, with an emphasis on parallel resource utilization. The buffer that *prusage* points to will be filled in with the following structure:

```
struct cvxprusage {
  struct timeval pru_stime; /* system time, in secs and usecs */
  struct timeval pru_utime; /* user time,   in secs and usecs */
    /* user PC samples */
  unsigned long long pru_utotal;/* cumulative parallelism counts */
  unsigned long long pru_usamples;/* number parallelism samples */
    /* system PC samples */
  unsigned long long pru_stotal;/* cumulative parallelism counts */
  unsigned long long pru_ssamples;/* number parallelism samples */
  unsigned long long pru_pvtime;/* process virtual time */
    /* room for expansion */
  unsigned long long pru_filler[11];/* reserved for growth */
};
```

The fields are interpreted as follows:

pru_utime the total amount of CPU time spent executing in user mode. This is identical to the value returned in the ru_exutime field of the structure returned by *getrusage(2)*.

pru_stime the total amount of CPU time spent in the system executing on behalf of the process. This is identical to the value returned in the ru_stime field of the structure returned by *getrusage(2)*.

pru_usamples the number of times the system sampled the number of threads running in this process and found its program counter in user space.

pru_utotal the cumulative number of threads the system found for all samples taken while the program's program counter was in user space.

pru_ssamples the number of times the system sampled the number of threads running in this process and found its program counter in system space.

pru_stotal the cumulative number of threads the system found for all samples taken while the program's program counter was in system space.

pru_pvtime the process virtual time. This is the total wall time the process spent executing in user mode.

The system samples the number of threads running in a process at a fixed rate, currently 100 times per second. An estimate of user level process parallelism can be made by dividing *pru_utotal* by *pru_usamples*. User level parallelism within particular regions can be estimated by using the deltas of *pru_utotal* and *pru_usamples* from two *cvxprusage* calls. A similar estimate of user level process parallelism can be obtained by dividing *pru_utime* and *pru_pvtime*.

SEE ALSO

getrusage(2), wait(2)

"Accounting Reports" chapter in the *Managing ConvexOS:Operations Guide.*

NAME

cvxstat, cvxlstat, cvxfstat – get file status

SYNOPSIS

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

cvxstat(path, buf, len)
char *path;
struct cvxstat *buf;
int len;

cvxlstat(path, buf, len)
char *path;
struct cvxstat *buf;
int len;

cvxfstat(fd, buf, len)
int fd;
struct cvxstat *buf;
int len;
```

DESCRIPTION

cvxstat obtains information about the file named by *path*. 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.

cvxlstat is like *cvxstat* unless the named file is a symbolic link, in which case *cvxlstat* returns information about the link, while *cvxstat* returns information about the file that the link references.

cvxfstat obtains the same information about an open file referenced by the argument descriptor, such as would be obtained by an *open* call.

len is the length of the current *cvxstat* buffer. To allow for future expansion, only *len* bytes are copied from the current stat structure to the buffer *buf*. If *len* is greater than the size of the cvxstat structure, only *sizeof(struct cvxstat)* bytes are copied.

buf is a pointer to a *cvxstat* structure into which information is placed concerning the file. The contents of the structure pointed to by *buf* include the following members:

```
struct cvxstat {
    mode_t st_mode;     /* protection */
    nlink_t st_nlink;   /* number of hard links to the file */
    uid_t   st_uid;     /* user-id of owner */
    gid_t   st_gid;     /* group-id of owner */
    ino64_t st_ino;     /* this inode's number */
    off64_t st_size;    /* total size of file */
    time_t  st_atime;   /* file last access time */
    int     st_spare1;
    time_t  st_mtime;   /* file last modify time */
    int     st_spare2;
    time_t  st_ctime;   /* file last status change time */
    int     st_spare3;
    time_t  st_btime;   /* file creation (birth) time */
    int     st_spare4;
    long    st_blksize; /* optimal blocksize for file system i/o ops */
    s64_t   st_blocks;  /* actual number of blocks allocated */
    dev_t   st_dev;     /* device inode resides on */
    dev_t   st_rdev;    /* the device type, for inode that is device */
    s64_t   st_gen;     /* file generation number */
```

```
        u_int   st_dmonflags;  /* daemon control flags */
};
```

st_atime — Time when file data was last read or modified. Changed by the following system calls: *mknod*(2), *utimes*(2), *read*(2) and *write*(2). For reasons of efficiency, st_atime is not set when a directory is searched, although this would be more logical.

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

st_ctime — Time when file status was last changed. It is set both by writing and changing the i-node. Changed by the following system calls: *chmod*(2) *chown*(2), *link*(2), *mknod*(2), *rename*(2), *unlink*(2), *utimes*(2), *write*(2) and *truncate*(2).

The status information word *st_mode* has bits:

```
#define S_IFMT      0170000   /* type of file */
#define S_IFIFO     0010000   /* fifo special */
#define S_IFCHR     0020000   /* character special */
#define S_IFDIR     0040000   /* directory */
#define S_IFBLK     0060000   /* block special */
#define S_IFREG     0100000   /* regular */
#define S_IFLNK     0120000   /* symbolic link */
#define S_IFSOCK    0140000   /* socket */
#define S_ISUID     0004000   /* set user id on execution */
#define S_ISGID     0002000   /* set group id on execution */
#define S_ISVTX     0001000   /* see sticky(8) */
#define S_IREAD     0000400   /* read permission, owner */
#define S_IWRITE    0000200   /* write permission, owner */
#define S_IEXEC     0000100   /* execute/search permission, owner */
```

Mode bits 0000070 and 0000007 encode group and others permissions (see *chmod*(2)).

Device type *dev_t* is a 32-bit number that should be manipulated using only the *major()* and *minor()* macros (see *types.h).*

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

cvxstat and *cvxlstat* fail if one or more of the following are true:

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

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG] The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

[ENOENT] The file referred to by *path* does not exist.

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

[ELOOP] Too many symbolic links were encountered in translating *path*.

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

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

[EINVAL] *len* is less than or equal to zero.

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

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

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

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

NOTES

The file creation time *sb_btime* is valid only for files created under ConvexOS V9.0 and later versions. For files created prior to this time, the field is set to zero. The generation number *st_gen*, birth time *st_btime*, and daemon flags *st_dmonflags* are valid only for local files, and is set to zero for remote-mounted (NFS) files.

CAVEAT

The fields in the cvxstat structure currently marked *st_spare1*, *st_spare2*, and *st_spare3* are present in preparation for inode time stamps expanding to 64 bits. This, however, can break certain programs that depend on the time stamps being contiguous (in calls to *utimes*(2)).

SEE ALSO

chmod(2), chown(2), readlink(2), utimes(2)

BUGS

Generation numbers are not available for remote mounted files over NFS.

NAME

cvxtruncate, cvxftruncate, cvxfhtruncate – truncate arbitrary blocks of a file.

SYNOPSIS

```
cvxtruncate(path, startoffset, count, migflag)
char *path;
long long startoffset;
long long count;
int migflag;

cvxftruncate(fd, startoffset, count, migflag)
int fd;
long long startoffset;
long long count;
int migflag;

#include <sys/dmonfh.h>

cvxfhtruncate(fh, startoffset, count, migflag)
struct dmon_fhandle *fh;
long long startoffset;
long long count;
int migflag;
```

DESCRIPTION

cvxtruncate truncates arbitrary blocks of a file. The starting offset *startoffset* and the number of bytes to truncate *count* must be file system block aligned.

Holes within a file can be truncated. Blocks beyond the end of the file (*startoffset* is greater than the last byte in the file) cannot be truncated. If *startoffset* + *count* is greater than the size of the file, the last block in the file is used.

If truncating the file (*migflag* is set to 0), the size of the file is updated only if the last block of the file is included in *startoffset* + *count*. The size of the file is not changed when migrating a file to secondary storage (*migflag* set to 1). Only the superuser or a daemon can truncate a migrated block.

The *cvxftruncate* variant accepts a file descriptor *fd* to a file. The *cvxfhtruncate* variant accepts a file handle to a file.

All variants of *cvxtruncate* are valid only for local files.

RETURN VALUE

The value –1 is returned if an error occurs, and external variable *errno* is set to indicate the cause of the error. Otherwise, the number of blocks actually held by the file is updated.

ERRORS

All variants of *cvxtruncate* succeed unless:

[EINVAL]	Starting offset *startoffset* is not file system block aligned.
[EINVAL]	The number of bytes to truncate *count* is not file system block aligned.
[EINVAL]	The starting offset *startoffset* is beyond the end of the file.
[EACCESS]	Write access to the file is denied.
[EISSOCK]	The target file is a socket.
[EISDEV]	The target file is a device special file.
[EREMOTE]	The target file is not a local file.
[ETXBSY]	The file is a pure procedure (shared text) file that is being executed.

[EIO] An IO error while trying to allocate an indirect block pointer.

[EISDIR] The file is a directory.

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

[EPERM] An attempt was made to truncate a migrated block and the user is not the superuser or a daemon.

cvxtruncate fails if:

[ENOENT] The named file does not exist.

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG] The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

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

[ELOOP] Too many symbolic links were encountered in translating *path*.

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

[EACCESS] The caller does not have write permission on the file.

cvxftruncate fails if:

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

cvxfhtruncate fails if:

[EPERM] The caller is not the superuser or a daemon.

NOTES

Bytes and blocks within a file are zero-relative. For example, on an 8k/1k filesystem, the first block in a file is block zero. The first byte in block zero is byte zero, and the last byte in block zero is byte 8191. Byte 8192 is the first byte of the second block. A file of size 8192 bytes takes up only one block.

These system calls are not supported on NFS files.

EXAMPLE

To truncate the second and third blocks of a file on an 8k/1k filesystem, the following call could be used. Assume that *fd* points to an open file.

```
cvxftruncate(fd, 8192, 16384, 0);
```

This starts at byte 8192 (the first byte in the second block of the file), and truncates 16384 bytes (two full blocks worth of data).

SEE ALSO

truncate(2)

NAME

dmon_fcntl – Daemon operations on files.

SYNOPSIS

```
#include <sys/dmon.h>
#include <sys/dmonfh.h>
#include <sys/dmonargs.h>

res = dmon_fcntl(descrip, cmd, arg)
caddr_t descrip;
int     cmd;
u_int   arg;
```

DESCRIPTION

dmon_fcntl is used to perform miscellaneous operations on files by daemons. Operations are performed on behalf of the dmon specified by *arg*, and are specified by the command *cmd*. The target file can be identified by either a file descriptor or a file handle in *descrip*.

The specified daemon *arg* must be in the range of 0-3, or ENODMON is returned.

The following commands are recognized.

DMON_FHSUSPEND
: Suspend callouts for the file referenced by *descrip* for the daemon specified by *arg*. Callouts may be suspended for only 32 files at any one time. EMAXSUSP is returned if an attempt is made to suspend callouts for more than 32 files. Only the superuser or daemon may execute this command.

DMON_FHRESUME
: Resume callous for the file referenced by *descrip* for the daemon specified by *arg*. Only the superuser or daemon may execute this command.

DMON_FHSETASSOC
: Permanently associate the dmon specified by *arg* with the file reference by *descrip*. Only the superuser or daemon may execute this command.

DMON_FHCLEARASSOC
: Clear the permanent association of the daemon specified by *arg* with the file reference by *descrip*. Only the superuser or daemon may execute this command.

DMON_FHGETASSOC
: Return a bit vector in *res* represents the daemons permanently associated with the file referenced by *descrip*.

DMON_FHSETFLAGS
: Sets the eight bits for a file that are under control of the daemon to the value as indicated by the low-order byte (bits 0-7) of *arg*. Only the superuser or daemon may execute this command. The value of the bitfield can be retrieved via *cvxstat*(2).

ERRORS

If an error has occurred, a value of –1 is returned in *res* and *errno* is set to indicate the error. *dmon_fcnt* fails if one or more of the following are true:

[EINVAL] *cmd* is invalid.

[EPERM] Only the superuser or daemon may perform the specified operation.

[ENODMON] The daemon specified by *dmon* is not in the range 0-3.

[ENODMON] Daemons are not enabled on this system.

[EBADF] For the file descriptor variants, the file descriptor referenced by *descrip* was

invalid.

[EIO] For the file handle variants, the file handle specified by *descrip* referenced a file on a file system that is not mounted.

[ENOENT] For the file handle variants, the file handle specified by *descrip* referenced a file that does not exist, or the generation number in the file is incorrect.

[EDMONSUSP] *cmd* is DMON_FDSUSPEND or DMON_FHSUSPEND and the daemon specified by *dmon* already has callouts suspended for this file.

[EMAXSUSP] *cmd* is DMON_FDSUSPEND or DMON_FHSUSPEND and the daemon specified by *dmon* has the maximum number of callouts suspended.

[EAGAIN] *cmd* is DMON_FDSUSPEND or DMON_FHSUSPEND and a slot was not found to save the file. The call should be retried.

[ESUSPCLEAR] *cmd* is DMON_FDRESUME or DMON_FHRESUME and the dmon did not have callouts suspended for the file.

[EACCES] *cmd* is DMON_GETASSOC and the file does not have read permission for the caller.

[EREMOTE] The file referenced by *descrip* is an NFS file.

SEE ALSO

cvxstat(2)

NAME

dmon_ioctl – control daemons

SYNOPSIS

```
#include <sys/file.h>
#include <sys/dmon.h>
#include <sys/dmonfh.h>
#include <sys/dmonargs.h>

res = dmon_ioctl(dmon, cmd, argp)
int dmon;
int cmd;
char *argp;
```

DESCRIPTION

dmon_ioctl is used to perform miscellaneous operations for daemons. Operations are performed on the daemon specified by *dmon;* operations can take different structures in the argument *argp*. This command is restricted to superusers and daemons.

The following commands are recognized.

DMON_INIT A daemon must initialize itself before it can receive events from the kernel. There can be a maximum of four daemons, numbered zero through three, that can be initialized at any given time. The structure from *sys/dmon.h* must be used.

```
struct dmon_info {
        int     dmon_chanfd;                    /* fd of krpc channel */
        int     dmon_type;                      /* type of dmon */
        u_int   dmon_flag;                      /* control flags */
        int     dmon_event[MAX_EVENTS];         /* events daemon wants */
};
```

The daemon must specify the krpc channel to use to receive event notifications in *dmon_chanfd*. This is the file descriptor of a previously allocated channel (see DMON_CHANALLOC). The daemon must also specify its type in *dmon_type*; only daemon numbers of a certain type are allowed to occupy the available slots. The valid types are as follows:

```
#define DMON_SYSHIGH    0       /* highest priority dmon */
#define DMON_SYSMIG     1       /* system migration */
#define DMON_USERMIG    2       /* user migration */
#define DMON_USER       3       /* user defined */
```

For example, the system migration daemon is daemon number one, and must specify the DMON_SYSMIG type in *dmon_type* when initializing itself.

DMON_STATUS Obtain the status of the daemon specified by *dmon*. The *dmon_info* structure is passed as the *argp*. Events that the daemon is handling are indicated by a nonzero value at the particular offset in the event array *dmon_event* that corresponds to the event. If the daemon specified by *dmon* is not initialized, *EDMONCLEAR* is returned.

DMON_CALLOUTSTATUS

Obtain the file handle for the file that currently has callouts suspended. If more than one file has callouts suspended, then only the first one found is returned. If the daemon specified by *dmon* does not have any callouts currently suspended, then *ESUSPCLEAR* is returned. The daemon file handle

structure, *dmon_fhandle* from *sys/dmonfh.h* must be passed as the argument *argp*.

```
struct dmon_fhandle    {
        long            fh_dev;                 /* device mounted on */
        u_int           fh_flags;               /* control flags */
        s64_t           fh_ino;                 /* inode number */
        union {
                s64_t   fh_ugen;                /* generation number */
                char    fh_unfs[NFS_FHSIZE];    /* handle for remote files */
        } fh_un;
};
```

DMON_FHREMOVESUSP

Clear the suspended callouts for the file referenced by *argp*, and remove all references the daemon may still have to it from the daemon's internal tables. *argp* points to a *struct dmon_fhandle* that should have been obtained from a previous DMON_CALLOUTSTATUS command.

DMON_CHANALLOC

Allocate a krpc channel for use in communicating between the daemon and the kernel (see *krpc(2)*). A valid file descriptor for the channel is returned in *res*.

DMON_FDSETLOCALCHAN

Set up a temporary krpc channel for a specific file. All callouts use the temporary channel instead of the default channel. DMON_FDSETLOCALCHAN accepts a file descriptor for the target file. The variant DMON_FHSETLOCALCHAN accepts a file handle. (see *sys/dmonfh.h*). A daemon can set up a maximum of 32 local channels. Each file may have only one local channel associated with it. Each local channel can intercept specified events by using the DMON_FDSETLOCALCHANEVNT command. By default, the local channel will receive all events which are enabled on the master channel. The *dmon_localchan* structure from *sys/dmonargs.h* must be used. The target channel is specified in *loc_chanfd* and the target file is specified in the *loc_descriptor* union.

```
struct dmon_localchan {
   int              loc_chanfd;  /* fd of new channel */
   union {
         int        fd;          /* descrip for file */
         dmonfh_t   fh;          /* handle for file  */
   } loc_descriptor;
};
```

DMON_FDSETLOCALCHANEVNT

Specify the events which are to be received on the local channel. The variant DMON_FHSETLOCALCHANEVNT accepts a file handle. The *dmon_localchanevent* structure from *sys/dmonargs.h* must be used. The bitvector of events to receive is specified in *events*, and the target file is specified in the *loc_descriptor* union.

```
struct dmon_localchanevent {
   u64_t            events;      /* bitvector of events to receive */
   union {
         int        fd;          /* descrip for file */
```

```
        dmonfh_t   fh;          /* handle for file */
    } loc_descriptor;
};
```

DMON_FDCLEARLOCALCHAN

Clears a temporary krpc channel from a file. All callouts revert to using the default krpc channel specified when the daemon initialized itself. DMON_FDCLEARLOCALCHAN accepts a file descriptor for the target file, while the variant DMON_FHCLEARCLOCALCHAN accepts a file handle. The *dmon_localchan* structure from *sys/dmonargs.h* must be used. Only the descriptor of the file *loc_descriptor* is needed.

ERRORS

If an error has occurred, a value of –1 is returned in *res* and *errno* is set to indicate the error. *dmon_ioctl* fails if one or more of the following are true:

[EINVAL] *cmd* is invalid

[EPERM] Only the superuser or daemon may perform the specified operation.

[ENODMON] *dmon* is not in the range 0-3.

[ENODMON] Daemons are not enabled on this system.

[ENFILE] *cmd* is DMON_FHSETLOCALCHAN or DMON_FDSETLOCALCHAN and the kernel file table is full.

[EDMONINUSE] *cmd* is DMON_INIT and the daemon is already initialized.

[EAGAIN] *cmd* is DMON_INIT and the daemon is in the process of closing.

[EDMONBADTYPE] *cmd* is DMON_INIT and the daemon is not of the correct type for the slot for which it is initializing.

[EBADCHAN] *cmd* is DMON_INIT, DMON_FDSETLOCALCHAN, or DMON_FHSETLOCALCHAN and the krpc channel is invalid.

[EBADF] *cmd* is DMON_FDSETLOCALCHAN or DMON_FDCLEARLOCALCHAN and the file descriptor for the target file is invalid.

[EFAULT] The buffer for the command argument *argp* is invalid.

[ECHANASSOC] *cmd* is DMON_FDSETLOCALCHAN or DMON_FHSETLOCALCHAN and there is already a local channel associated for the file.

[EMAXLOCALCHAN] *cmd* is DMON_FDSETLOCALCHAN or DMON_FHSETLOCALCHAN and the daemon has the maximum number of local channels already in use.

[EAGAIN] *cmd* is DMON_FDSETLOCALCHAN or DMON_FHSETLOCALCHAN and a slot was not found to save the file. The call should be retried.

[ECHANCLEAR] *cmd* is DMON_FDCLEARLOCALCHAN or DMON_FHCLEARLOCALCHAN and there was no local channel set for the target file.

[EDMONCLEAR] *cmd* is DMON_STATUS and the dmon is not initialized.

[ESUSPCLEAR] *cmd* is DMON_CALLOUTSTATUS and no files currently have callouts suspended.

[EDMONSUSP] *cmd* is DMON_INIT and a file(s) still have callouts suspended.

NAME

dup, dup2 – duplicate a descriptor

SYNOPSIS

newd = dup(oldd)
int newd, oldd;

dup2(oldd, newd)
int oldd, newd;

DESCRIPTION

dup() duplicates an existing object descriptor. The argument *oldd* is a small, non-negative integer index in the per-process descriptor table. The value must be less than the size of the table, which is returned by *getdtablesize*(2). The new descriptor *newd* returned by the call is the lowest numbered descriptor that is not currently in use by the process.

The object referenced by the descriptor does not distinguish between references using *oldd* and *newd* in any way. Thus if *newd* and *oldd* are duplicate references to an open file, *read*(2), *write*(2), and *lseek*(2) calls all move a single pointer into the file. If a separate pointer into the file is desired, a different object reference to the file must be obtained by issuing an additional *open*(2) call.

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

RETURN VALUE

The value –1 is returned if an error occurs in either call. The external variable *errno* indicates the cause of the error.

ERRORS

dup() and *dup2()* fail if:

[EBADF]	*oldd* or *newd* is not a valid active descriptor
[EMFILE]	Too many descriptors are active.

NOTES

These functions are an alternate interface to the *fcntl()* function using the F_DUPFD command. The call

dup(fd)

is equivalent to

fcntl(fd,F_DUPFD,0);

The call

dup2(fd,fd2)

is equivalent to

fcntl(fd,F_DUPFD,fd2)

except that

(1) If fd2 is less than zero or greater than OPEN_MAX, *dup2* fails with errno EBADF.

(2) If fd is valid, and equal to fd2, *dup2()* does not have the effect of closing and reopening the file.

(3) If fd is invalid, *dup2()* does not close fd2.

SEE ALSO

accept(2), open(2), close(2), fcntl(2), pipe(2), socket(2), socketpair(2), getdtablesize(2)

NAME

execve – execute a file

SYNOPSIS

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

DESCRIPTION

execve() transforms the calling process into a new process. The new process is constructed from an ordinary file called the *new process file*. This file is either an executable object file, or a file of data for an interpreter. An executable object file consists of an identifying header, followed by pages of data representing the initial program (text) and initialized data pages. Additional pages may be specified by the header to be initialize with zero data.

An interpreter file begins with a line of the form "*#! interpreter.*" When an interpreter file is *execve()*'d, the system *execve()*'s the specified *interpreter*, giving it the name of the originally exec'd file as an argument, and shifting over the rest of the original arguments.

There can be no return from a successful *execve* because the calling core image is lost. This is the mechanism whereby different process images become active.

The argument *argv* is an array of character pointers to null-terminated character strings. These strings constitute the argument list to be made available to the new process. By convention, at least one argument must be present in this array, and the first element of this array should be the name of the executed program (i.e. the last component of *name*).

The argument *envp* is also an array of character pointers to null-terminated strings. These strings pass information to the new process that are not directly arguments to the command.

Descriptors open in the calling process remain open in the new process, except for those that have their close-on-exec flag is set. Descriptors which remain open are unaffected by *execve()*.

Ignored signals remain ignored across an *execve()*, but signals that are caught are reset to their default values. The signal stack is reset to an undefined state.

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

The *saved-set-user-ID* and *saved-set-group-ID* are saved for use by the *setuid*(2) and *setgid*(2) functions.

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

process ID	see *getpid*(2)
parent process ID	see *getppid*(2)
process group ID	see *getpgrp*(2)
session membership	see *setsid*
real user id	see *setuid*
real group id	see *setgid*
access groups	see *getgroups*(2)
working directory	see *chdir*(2)
root directory	see *chroot*(2)
control terminal	see *tty*(4)
resource usages	see *getrusage*(2)
interval timers	see *getitimer*(2)
resource limits	see *getrlimit*(2)
file mode mask	see *umask*(2)
signal mask	see *sigprocmask*(3)
pending signals	see *signal*(2)

When the executed program begins, it is called as follows:

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

where *argc* is the number of elements in *argv* (the "arg count") and *argv* is the array of character pointers to the arguments themselves.

envp is a pointer to an array of strings that constitute the *environment* of the process. A pointer to this array is also stored in the global variable "environ." Each string consists of a name, an "=," and a null-terminated value. The array of pointers is terminated by a null pointer. The shell *sh*(1) passes an environment entry for each global shell variable defined when the program is called. See *environ*(7) for some conventionally used names.

RETURN VALUE

If *execve()* returns to the calling process, an error has occurred; the return value will be –1 and the global variable *errno* will contain an error code.

ERRORS

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

[E2BIG] The number of bytes in the argument list and environment is greater than ARG_MAX.

[ENAMETOOLONG] The *path* argument is longer than PATH_MAX.

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

[ENOTDIR] A component of the new process file is not a directory.

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

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

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

[ENOEXEC] The new process file has the appropriate access permission, but has an invalid magic number in its header.

[EPERM] The file is on a file system mounted with the "-o nosuid" option (see *mount(8)*) and the file has the setuid or setgid permission bits on.

[EPERM] The new process file file has the non-swapped bit set (see OF_NON_SWAP in *a.out(5)*) and the caller is not root.

[EPERM] The new process file is a shell script with the setuid and/or setgid bits set, and the value of the kernel tunable *suid_shell_script* is zero.

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

[ENOMEM] The new process requires more virtual memory than is allowed by the imposed maximum (*getrlimit*(2)).

[EFAULT] 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.

BACKWARD COMPATIBILITY

In previous versions of the operating system, a non-root owned, *setuid* program retained all the additional privileges of "root" when executed by "root."

SEE ALSO

exit(2), fork(2), close(2), sigvec(2), execl(3), a.out(5), environ(7)

NAME

_exit – terminate a process

SYNOPSIS

_exit(status)
int status;

DESCRIPTION

_exit() terminates a process with the following consequences:

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

If the parent process of the calling process is executing a *wait()*, or is interested in the SIGCHLD signal, then it is notified of the calling process's termination, and the low-order eight bits of *status* are made available to it.

The parent process ID of all of the calling process's existing child processes are also set to 1. This means that the initialization process inherits each of these processes as well.

Most C programs call the library routine *exit*(3) which performs cleanup actions in the standard I/O library before calling *_exit()*.

RETURN VALUE

This call never returns.

SEE ALSO

fork(2), wait(2), exit(3)

NAME

faillog – enable or disable logging of failed file accesses

SYNOPSIS

faillog(file)
char *file;

DESCRIPTION

The system is prepared to write a record in a log file each time a process attempts to access a file for which it does not have permission. This call, with a null-terminated string naming the existing file */usr/adm/failure_log* as the argument, turns on logging; it appends a record to */usr/adm/failure_log* for each failed access. The *faillog* system call with an argument of 0 turns off failure logging.

The log file format is shown in *failure_log*(5).

If *faillog* is invoked when logging is already turned on, the failure log switches to the new file. There is no interruption in logging when this occurs. All failures are recorded in one of the log files.

This call is permitted only to the superuser.

NOTES

Failed file access logging is automatically disabled when the file system the log file resides on becomes 98% full; it is enabled again when the file system becomes less than 96% full. The percentages may be changed using the *sysgen* tunable parameters LOG_SUSPEND and LOG_RESUME.

When the Network File System (NFS) is used, failed accesses to files residing on the NFS server are logged on the server, not the client. See the "System Generation" chapter in the *CONVEX System Manager's Guide* for more information.

RETURN VALUE

On error, –1 is returned. The file must exist and the call may be exercised only by the superuser.

ERRORS

faillog will fail if one of the following is true:

[EPERM]	The caller is not the super-user.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*File* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EACCES]	The file is a character or block special file.

SEE ALSO

failure_log(5), faillogon(8), faillogpr(8).
"System Protection" and "System Generation" chapters in the *CONVEX System Manager's Guide*

NAME

fchmod – change mode of file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
fchmod(fd, mode)
int fd;
mode_t mode;
```

DESCRIPTION

The file referenced by the descriptor *fd* has its mode changed to *mode*. Modes are constructed by *or*ing together some combination of the following:

S_ISUID	04000	set effective user ID on execution
S_ISGID	02000	set effective group ID on execution
_S_ISVTX	01000	see *sticky*(8)
S_IRUSR	00400	read by owner
S_IWUSR	00200	write by owner
S_IXUSR	00100	execute (search on directory) by owner
S_IRWXG	00070	read, write, execute (search) by group
S_IRWXO	00007	read, write, execute (search) by others

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

Writing or changing the owner of a file turns off the set-user-ID and set-group-ID bits. This makes the system somewhat more secure by protecting set-user-ID (set-group-ID) files from remaining set-user-ID (set-group-ID) if they are modified, at the expense of a degree of compatibility.

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

fchmod() will fail if:

[EBADF]	The descriptor is not valid.
[EINVAL]	*fd* refers to a socket, not to a file.
[EROFS]	The file resides on a read-only file system.

SEE ALSO

chmod(2), open(2), chown(2)

NAME

fchown – change owner and group of file by descriptor

SYNOPSIS

#include <sys/types.h>

fchown(fd, owner, group)
int fd;
uid_t owner;
gid_t group;

DESCRIPTION

The file which is referenced by *fd* has its *owner* and *group* changed as specified. Only the super-user may change the owner of the file, because if users were able to give files away, they could defeat the file-space accounting procedures. The owner of the file may change the group to a group of which he is a member.

On some systems, *chown()* clears the set-user-ID and set-group-ID bits on the file to prevent accidental creation of set-user-ID and set-group-ID programs owned by the super-user.

fchown() is particularly useful when used in conjunction with the file locking primitives.

RETURN VALUE

Zero is returned if the operation was successful; –1 is returned if an error occurs, with a more specific error code being placed in the global variable *errno*.

ERRORS

fchown will fail if:

[EBADF]	*fd* does not refer to a valid descriptor.
[EINVAL]	*fd* refers to a socket, not a file.
[EPERM]	The effective user ID is not the super-user and either the file owner was being changed or the effective user ID does not match the file owner or the new group is not a member of the processes group list.
[EROFS]	The named file resides on a read-only file system.
[EIO]	An I/O error occurred while reading from or writing to the file system.

CONVEX EXTENSIONS

A value of _SAME_UID passed for owner or _SAME_GID for group results in the current owner or group remaining unchanged.

BACKWARD COMPATIBILITY

The *owner* and *group* parameters were formerly declared to be type int, although the high word was unusable.

SEE ALSO

chown(2), chown(8), chgrp(1), chmod(2), flock(2)

NAME

fcntl – file control

SYNOPSIS

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

res = fcntl(fd, cmd, arg)
int res;
int fd, cmd, arg;
```

DESCRIPTION

fcntl() provides for control over descriptors. The argument *fd* is a descriptor to be operated on by *cmd* as follows:

F_DUPFD Return a new descriptor as follows:

Lowest numbered available descriptor greater than or equal to *arg*.
Same object references as the original descriptor.
New descriptor shares the same file pointer if the object was a file.
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 *execv*(2) system calls.

F_GETFD Get the flags associated with the file descriptor *fd*. Currently, FD_CLOEXEC is the only flag. If FD_CLOEXEC is not set, then the file will remain open across *exec*; if FD_CLOEXEC is set, then the file will be closed upon execution of *exec*.

F_SETFD Set the flags associated with *fd* to the argument *arg* .

F_GETFL Get descriptor status flags, as described below.

F_SETFL Set descriptor status flags.

F_GETLK Get a description of the first lock which would block the lock specified in the *flock* structure pointed to by *arg*. The information retrieved overwrites the information in the *flock* structure. The starting offset in the *flock* structure, *l_whence*, will always be SEEK_SET, with the *l_start* and *l_len* fields set accordingly. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

F_SETLK Set or clear an advisory record lock according to the *flock* structure pointed to by *arg*. F_SETLK is used to establish shared (F_RDLCK) and exclusive (F_WRLCK) locks, or to remove either type of lock (F_UNLCK). If the specified lock cannot be applied, *fcntl* will return with an error value of -1.

F_SETLKW This *cmd* is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the requesting process will sleep until the lock may be applied.

The flags for the F_GETFL and F_SETFL flags are:

O_NONBLOCK Nonblocking I/O; if no data is available to a *read()* call or if a write operation would block, the call returns -1 with the error EAGAIN. *CONVEX EXTENSION:* For files and filesystems under control of a

migration daemon, if this flag is set the process will receive an *ENOSPC* error instead of blocking if there are not enough free blocks or fragments to statisfy a filesystem allocation request. Setting O_NONBLOCK on a file for migration purposes is only valid for directories and regular files; all other file types (named pipes, devices, sockets, etc) will be silently ignored.

O_APPEND Force each write to append at the end of file; corresponds to the O_APPEND flag of *open*(2).

CONVEX adds the following file control flag as extension:

O_LARGEFILE Allow reading and writing regular files at offsets greater than 2^31 - 1; corresponds with the O_LARGEFILE flag of *open*(2).

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 FD_CLOEXEC is defined)
F_GETFL	Value of flags and access modes
other	Value other than –1.

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

ERRORS

fcntl() will fail if one or more of the following are true:

[EACCES] The *cmd* argument is F_SETLK, the type of lock (*l_type*) is a shared lock (*F_RDLCK*) or exclusive lock (*F_WRLCK*), and the segment to be locked is already exclusive-locked by some other process, or the type is exclusive and some portion of the segment is already shared-locked or exclusive-locked by some other process.

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

[EBADF] *cmd* is F_RDLCK and *fildes* is not open for reading.

[EBADF] *cmd* is F_SETLK or F_SETLKW, the type is F_WRLCK, and *fildes* is not open for writing.

[EMFILE] *cmd* is F_DUPFD and the maximum allowed number of file descriptors is currently open.

[EINVAL] *cmd* is F_DUPFD and *arg* is negative or greater than the maximum allowable number (see *getdtablesize*(2)).

[EINVAL] FASIO is specified on a F_SETFL (see EXTENSIONS below) *cmd* and *fd* is DTYPE_SOCKET.

[EFAULT] *cmd* is F_GETLK, F_SETLK, or F_SETLKW and *arg* points to an invalid address.

[EINVAL] *cmd* is F_GETLK, F_SETLK, or F_SETLKW and the data *arg* points to is not valid.

[EINTR] *cmd* is F_SETLKW and a signal interrupted the process while it was waiting for the lock to be granted.

[ENOLCK] *cmd* is F_SETLK or F_SETLKW and there are no more file lock entries available.

[ENETUNREACH] *cmd* is F_GETLK, F_SETLK, or F_SETLKW and there is no loopback interface configured for the INET address family.

[EALREADY] *cmd* is _F_CACHEBUFS and *arg* is zero and buffer pointer caching is not currently enabled.

CONVEX EXTENSIONS

CONVEX adds the following *cmd* values as extensions:

_F_GETOWN Get the process ID or process group currently receiving SIGIO and SIGURG signals; process groups are returned as negative values. The value returned by the _F_GETOWN function of file descriptor owner.

_F_SETOWN Set the process or process group to receive _SIGIO and _SIGURG signals; process groups are specified by supplying *arg* as negative; otherwise, *arg* is interpreted as a process ID.

_F_SETBLKSIZE
Set the physical record size for block tape operations to the value of *arg*. This value defaults to 65536 when the file is opened.

_F_CACHEBUFS
Tell the filesystem to cache *arg* number of buffer pointers in the in-core inode. Doing this dramatically speeds up the *fsync()* system call for large files undergoing small writes. If *arg* is zero, buffer pointer caching is turned off. *Arg* may not be more than 64.
This command is only available for local ufs files.

CONVEX adds the following file status flags as extensions:

_FASIO Use daemon processes to accomplish asynchronous I/O. Subsequent operations on this *fd* cause the user process to proceed in parallel with the I/O transfers. Several I/O transfers may be proceeding in parallel to or from the same file (*fd*), each under the control of a separate daemon process, limited only by a system wide configuration maximum ("ps -axl" displays daemon processes as "asiodaemon".) Synchronization may be accomplished with the *asiostat*(), *select*(), or *fstat()* system calls, or by the use of _FASYNC below. Note that it is innocently assumed that all requested I/O will be performed; the *read()* or *write()* system calls will return as if all the data had been transferred, even though the transfer may be impossible. To determine the actual results, the *asiostat()* system call will return the sum of the number of bytes transferred on all asynchronous I/O requests on a given file descriptor since the last *asiostat*() call. Note that the FASIO property may be inherited by forked children, and that it is illegal to set FASIO on socket descriptors. Asynchronous I/O seems to help tape performance, but due to the existence of the buffer cache, read ahead, and write behind, sequential file transfer is already quite asynchronous, and no performance improvement may be noted. The *brk*(), *exit*(), and *fork*() system calls will wait for all asynchronous I/O invoked by the calling process to be completed before being processed. The *asiostat*(), *close*(), *fcntl*(), *flock*(), *fstat*(), *fsync*(), *ftruncate*(), and *ioctl* system calls will wait for all asynchronous I/O on the specified file descriptor (*fd*) to be completed before being processed.

_FASYNC If the _FASIO bit is set (see above), a _SIGIO signal will be sent to a process when all its outstanding asynchronous I/O (using daemon processes) has been completed. If the _FASIO bit is not set, _FASYNC enables the _SIGIO signal to be sent to the process group when I/O is possible, e.g. upon availability of data to be read. (This latter facility is available only on tty operations.)

_FNCACHE This bit requests the kernel to bypass the incore buffer cache and perform I/O transfers directly to/from user address space. As a side effect, file system read ahead and write behind are disabled. For files that are being sequentially accessed, the net result is usually lost performance. _FNCACHE might be helpful in situations where the file in question is being randomly accessed, but you should test the results with and without this option set before using it. The operating system cannot bypass the buffer cache if the transfers are not aligned on the file's blocksize boundaries because disk controllers can't start transferring in the middle of sectors. The *fstat()* system call returns the proper blocksize as *st_blksize*. Transfers should start (by seeking if necessary) at integral multiples of *st_blksize*.

NOTES

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee exclusive access (i.e., processes may still access files without using advisory locks, possibly resulting in inconsistencies).

The record locking mechanism allows two types of locks: shared locks (F_RDLCK) and exclusive locks (F_WRLCK). More than one process may hold a shared lock for a particular segment of a file at any given time, neither multiple exclusive locks nor mixed shared and exclusive locks may exist simultaneously on any segment.

In order to claim a shared lock, the descriptor must have been opened with read access. The descriptor on which an exclusive lock is being placed must have been opened with write access.

A shared lock may be *upgraded* to an exclusive lock, and vice versa, simply by specifying the appropriate lock type with a *cmd* of F_SETLK or F_SETLKW; the previous lock will be released and the new lock applied (possibly after other processes have gained and released the lock).

If the *cmd* is F_SETLKW and the requested lock cannot be claimed immediately (e.g., another process holds an exclusive lock that partially or completely overlaps the current request) then the calling process will block until the lock may be acquired. Processes blocked awaiting a lock may be awakened by signals.

Care should be taken to avoid deadlock situations in applications in which multiple processes perform blocking locks on a set of common records.

The record that is to be locked or unlocked is described by the *flock* structure, which is defined in <fcntl.h> as follows:

```
struct flock {
        short   l_type;         /* F_RDLCK, F_WRLCK, or F_UNLCK */
        short   l_whence;       /* flag to choose starting offset */
        off_t   l_start;        /* relative offset, in bytes */
        off_t   l_len;          /* length, in bytes; 0 means lock to EOF */
        pid_t   l_pid;          /* returned with F_GETLK */
};
```

The *flock* structure describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), and size (*l_len*) of the segment of the file to be affected. *l_whence* must be set to SEEK_SET, SEEK_CUR, or SEEK_END to indicate that the relative offset will be measured from the start of the file, current position, or end-of-file, respectively. The process ID field (*l_pid*) is only used with the F_GETLK *cmd* to return the description of a lock held by another process.

Locks may start and extend beyond the current end-of-file, but must not be negative relative to the beginning of the file. A lock may be set to always extend to the end-of-file by setting *l_len* to zero. If such a lock also has *l_whence* and *l_start* set to zero, the entire file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller

segments at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take affect. All locks associated with a file for a given process are removed when the file is closed or the process terminates. Locks are not inherited by the child process in a *fork*(2) system call.

It is not possible to lock a file beyond the two gigabyte boundary, do to restrictions in the implementation of large file access.

In order to maintain consistency in the network case, data must not be cached on client machines. For this reason, file buffering for an NFS file is turned off when the first lock is attempted on the file. Buffering will remain off as long as the file is open. Programs that do I/O buffering in the user address space, however, may have inconsistent results (the standard I/O package, for instance, is a common source of unexpected buffering).

The advisory record locking capabilities of *fcntl* are implemented throughout the network by the network lock daemon; see *lockd*(8C). If the file server crashes and is rebooted, the lock daemon will attempt to recover all locks that were associated with that server. If a lock cannot be reclaimed, the process that held the lock will be issued a SIGLOST signal.

BUGS

The asynchronous I/O facilities of FNDELAY and FASYNC are currently available only for tty and socket operations.

File locks obtained through the *fcntl* mechanism do not interact in any way with those acquired via *flock*(2). They do, however, work correctly with the exclusive locks claimed by *lockf*(3).

F_GETLK returns F_UNLCK if the requesting process holds the specified lock. Thus, there is no way for a process to determine if it is still holding a specific lock after catching a SIGLOST signal.

In a network environment, the value of *l_pid* returned by F_GETLK is next to useless.

Setting FNDELAY for regular files (for migration purposes) affects all open instances of the file. FNDELAY should only be used by cooperating processes so that they do not block when accessing a migrated file.

SEE ALSO

close(2), dup(2), execve(2), getdtablesize(2), open(2), sigvec(2), lockf(3), lockd(8C), ifconfig(8), getpattr(2)

NAME

fdpath – return a path to a file from a file descriptor

SYNOPSIS

```
#include <sys/param.h>
#include <sys/file.h>

fdpath(fd, pathbuf, buflen)
int fd;
char *pathbuf;
int *buflen;
```

DESCRIPTION

fdpath returns a path for a file referenced by the file descriptor *fd*. The buffer *pathbuf* should be large enough to hold the maximum path length, *MAXPATHLEN*, plus a terminating NULL byte. The length of the buffer must be specified by *buflen*.

buflen is updated to the actual length of the pathname (not including the terminating NULL byte), even in case of error.

RETURN VALUE

The value –1 is returned if an error occurs, and external variable *errno* is set to indicate the cause of the error. Otherwise, the pathname is copied to *pathbuf*.

ERRORS

fdpath fails if:

[ENOENT]	No name for this file descriptor
[ENOENT]	No saved file handles for this file descriptor
[EBADF]	Invalid file descriptor
[EACCES]	Execute access denied for a directory component.
[ENAMETOOLONG]	The name of the path does not fit in the user buffer.
[EIO]	Couldn't get status of a directory component.
[EIO]	IO error while reading a directory component.
[EINTR]	IO interrupted while reading directory component.

SEE ALSO

open(2), stat(2)

NAME

fhopen – open a file for reading or writing via a file handle

SYNOPSIS

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

fhopen(fh, flags)
struct dmon_fhandle  *fh;
int flags;
```

DESCRIPTION

fhopen opens the file referenced by the dmon file handle *fh* for reading and/or writing, as specified by the *flags* argument, and returns a descriptor for that file. The file must already exist; it is an error to attempt to create a file from a file handle. *flags* values are constructed by ORing flags from the following list (only one of the first three flags below may be used):

This call is restricted to the superuser or a daemon.

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY When opening a file on a filesystem under the control of a migration daemon:

If O_NDELAY is set:

The process wants to receive an *ENOSPC* error instead of blocking.

If O_NDELAY is clear:

The process blocks until sufficient blocks or fragments become available to satisfy the request.

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

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

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

The new descriptor is set to remain open across *execve*(2) system calls; see *close*(2) and *fcntl*(2).

There is a system enforced limit on the number of open file descriptors per process, whose value is returned by the *getdtablesize*(2) call.

RETURN VALUE

The value –1 is returned if an error occurs, and external variable *errno* is set to indicate the cause of the error. Otherwise a non-negative numbered file descriptor for the new open file is returned.

ERRORS

fhopen fails if:

[EPERM] The user is not the superuser or a daemon.

[EACCES] The required permissions (for reading and/or writing) are denied for the file referenced by *fh*.

[EACCES] The file referred to by *fh* does not exist.

[EMFILE] The system limit for open file descriptors per process has already been reached.

[ENFILE] The system file table is full.

[EROFS] The named file resides on a read-only file system, and the file is to be opened for writing.

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

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

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and the *open* call requests write access.

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

[EIO] The file corresponds to a tape device that is offline, or one that was opened for write access without having a write enable ring present on the tape.

[EOPNOTSUPP] An attempt was made to open a socket (not currently implemented).

NOTES

If a file is opened via *fhopen*, then no naming information is stored in the kernel. If a subsequent *fdpath(2)* is performed on the file in an attempt to obtain the name the file was open with, then the operation will fail since the file was not opened via a pathname.

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), read(2), write(2), umask(2), getpattr(2), fdpath(2)

NAME

fhpath – return the path a file was opened with

SYNOPSIS

```
#include <sys/param.h>
#include <sys/dmonfh.h>

fhpath(dfh, fh, pathbuf, buflen)
struct dmon_fhandle  *dfh, *fh;
char *pathbuf;
int  *buflen;
```

DESCRIPTION

fhpath returns the path for a file referenced by *fh* that is in the directory referenced by *dfh*. The buffer *pathbuf* should be large enough to hold the maximum path length, *MAXPATHLEN*, plus a terminating NULL byte. The length of the buffer must be specified by *buflen*.

buflen is updated to the actual length of the pathname(not including the terminating NULL byte), even in case of error.

RETURN VALUE

The value –1 is returned if an error occurs, and external variable *errno* is set to indicate the cause of the error. Otherwise, the pathname is copied to *pathbuf*.

ERRORS

fhpath fails if:

[ENOENT] No file exists for either the directory file handle *dfh* or the target file handle *fh*.

[EACCES] Execute access denied for a directory component.

[ENAMETOOLONG] The name of the path does not fit in the user buffer.

[EFAULT] The file handle supplied was invalid.

SEE ALSO

open(2), stat(2)

NAME

flock – apply or remove an advisory lock on an open file

SYNOPSIS

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

#define     LOCK_SH     1     /* shared lock */
#define     LOCK_EX     2     /* exclusive lock */
#define     LOCK_NB     4     /* don't block when locking */
#define     LOCK_UN     8     /* unlock */

flock(fd, operation)
int fd, operation;
```

DESCRIPTION

Flock applies or removes an *advisory* lock on the file associated with the file descriptor *fd*. A lock is applied by specifying an *operation* parameter which is the inclusive or of either LOCK_SH or LOCK_EX and, possibly, LOCK_NB. To unlock an existing lock *operation* should be LOCK_UN.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency (i.e. processes may still access files without using advisory locks possibly resulting in inconsistencies).

The locking mechanism allows two types of locks: *shared* locks and *exclusive* locks. At any time multiple shared locks may be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file.

A shared lock may be *upgraded* to an exclusive lock, and vice versa, simply by specifying the appropriate lock type; this results in the previous lock being released and the new lock applied (possibly after other processes have gained and released the lock).

Requesting a lock on an object which is already locked normally causes the caller to be blocked until the lock may be acquired. If LOCK_NB is included in *operation*, then this will not happen; instead the call will fail and the error EWOULDBLOCK will be returned.

NOTES

Locks are on files, not file descriptors. That is, file descriptors duplicated through *dup*(2) or *fork*(2) do not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent will lose its lock.

Processes blocked awaiting a lock may be awakened by signals.

BUGS

File locks obtained through the *flock*(2) mechanism do not interact in any way with those acquired via *fcntl*(2) or *lockf*(3).

RETURN VALUE

Zero is returned if the operation was successful; on an error a –1 is returned and an error code is left in the global location *errno*.

ERRORS

The *flock* call fails if:

[EWOULDBLOCK]	The file is locked and the LOCK_NB option was specified.
[EBADF]	The argument *fd* is an invalid descriptor.
[EINVAL]	The argument *fd* refers to an object other than a file.

SEE ALSO

open(2), close(2), dup(2), execve(2), fork(2)

NAME

fork – create a new process

SYNOPSIS

```
#include <sys/types.h>
        pid_t pid;
        pid = fork();
```

DESCRIPTION

fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process except for the following:

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 descriptors. These descriptors reference the same underlying objects. For instance, file pointers in file objects are shared between the child and the parent so that a *lseek*(2) on a descriptor in the child process can affect a subsequent *read* or *write* by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.

File locks set by the parent are not inherited by the child.

The resource utilization of a child process is set to 0; see *setrlimit*(2). (This includes *tms_utime, tms_stime, tms_cutime,* and *tms_cstime.*)

Pending alarms are cleared for the child process.

The set of signals pending for the child process is initialized to the empty set.

RETURN VALUE

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

ERRORS

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

[EAGAIN] The system-imposed limit (based on the "maxusers" boot-time parameter) on the total number of processes under execution would be exceeded.

[EAGAIN] The system-imposed limit (based on the "max_user_processes" boot-time parameter) on the total number of processes under execution by a single user would be exceeded.

[EDEADLK] The calling process is multi-threaded.

SEE ALSO

execve(2), wait(2)

NAME

fsync – synchronize a file's in-memory state with that on disk

SYNOPSIS

fsync(fd)
int fd;

DESCRIPTION

Fsync causes all modified data and attributes of *fd* to be moved to a permanent storage device. This normally results in all in-memory modified copies of buffers for the associated file to be written to a disk.

Fsync should be used by programs which require a file to be in a known state; for example in building a simple transaction facility.

RETURN VALUE

A 0 value is returned on success. A –1 value indicates an error.

ERRORS

The *fsync* fails if:

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

[EINVAL] *Fd* refers to a socket, not to a file.

SEE ALSO

sync(2), sync(8), update(8)

BUGS

The current implementation of this call is expensive for large files.

NAME

getaid – get a process' activity ID

SYNOPSIS

aid = getaid(pid)
int aid, pid;

DESCRIPTION

Getaid returns the activity ID of the process whose ID is *pid*. If *getaid* fails, the value **–1** is returned and *errno* is set to indicate the error. **–1** is not a legitimate value for a process' activity ID; any other 32-bit value is.

ERRORS

[ESRCH] The process specified does not exist.

SEE ALSO

setaid(2),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

NAME

getdirentries – gets directory entries in a filesystem independent format

SYNOPSIS

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

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

DESCRIPTION

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

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

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

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

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

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

RETURN VALUE

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

SEE ALSO

open(2), lseek(2)

ERRORS

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

[EBADF]	*fd* is not a valid file descriptor open for reading.
[EFAULT]	Either *buf* or *basep* point outside the allocated address space.
[EINTR]	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
[EIO]	An I/O error occurred while reading from or writing to the file system.

NAME

getdomainname, setdomainname – get/set name of current yellow pages domain

SYNOPSIS

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

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

DESCRIPTION

getdomainname returns the name of the yellow pages (YP) domain for the current processor, as previously set by *setdomainname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

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

The purpose of YP domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different YP domain name. At the current time, only the yellow pages service makes use of domains.

RETURN VALUE

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

ERRORS

The following errors may be returned by these calls:

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

[EINVAL] The *namelen* parameter is unreasonable. Currently, there is a limit of 63 characters for the domainname.

[EPERM] The caller was not the superuser. This error only applies to *setdomainname*.

BUGS

Domain names are limited to 63 characters.

NAME

getdtablesize – get descriptor table size

SYNOPSIS

nds = getdtablesize()
int nds;

DESCRIPTION

Each process has a fixed size descriptor table which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The call *getdtablesize* returns the size of this table.

SEE ALSO

close(2), dup(2), open(2)

NAME

getgid, getegid – get group identity

SYNOPSIS

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

gid_t gid;
gid = getgid()
#include <sys/types.h>

gid_t egid;
egid = getegid()
```

DESCRIPTION

getgid() returns the real group ID of the current process, *getegid* the effective group ID.

The real group ID is specified at login time.

The effective group ID is more transient and determines additional access permission during execution of a "set-group-ID" process. *getgid()* is most useful for such processes.

RETURNS

These functions are always successful; there is no return value which indicates an error.

SEE ALSO

getuid(2), setregid(2), setuid(3)

NAME

getgroups – get group access list

SYNOPSIS

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

ngrps = getgroups(gsize, gidset)
int ngrps, gsize;
gid_t *gidset;
```

DESCRIPTION

getgroups() gets the current group access list of the user process and stores it in the array *gidset*. The parameter *gsize* indicates the number of entries that may be placed in *gidset* . The return value is the number of entries in *gidset* that were filled.

As a special case, if *gsize* is zero, the number of groups which would be placed in *gidset* is returned without assigning into *gidset.*

No more than NGROUPS_MAX, as defined in <limits.h>, will ever be returned.

RETURN VALUE

A value of –1 indicates that an error occurred, and the error code is stored in the global variable *errno* .

BACKWARD COMPATIBILITY

In previous versions of the operating system, the array filled in by *getgroups()* was of type *int*; it is now of type *gid_t* .

CONVEX EXTENSIONS

For backward compatibility, CONVEX adds:

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

ngrps = cvx$int_getgroups(gsize, intgidset)
int ngrps, gsize;
int *intgidset.
```

Usage is otherwise as for *getgroups().*

ERRORS

The possible errors for *getgroups()* are:

[EFAULT]	The argument *gidset* is an invalid address.
[EINVAL]	The size of *gidset* as specified by *gsize* is not zero and is too small to accommodate the entire group access list.

SEE ALSO

setgroups(2), initgroups(3x)

NAME

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

SYNOPSIS

hostid = gethostid()
int hostid;

sethostid(hostid)
int hostid;

DESCRIPTION

Sethostid establishes a 32-bit identifier for the current processor which is intended to be unique among all UNIX systems in existence. (UNIX is a registered trademark of UNIX System Laboratories, Inc.) This is normally a DARPA Internet address for the local machine. This call is allowed only to the super-user and is normally performed at boot time.

Gethostid returns the 32-bit identifier for the current processor.

SEE ALSO

hostid(1), gethostname(2)

BUGS

32 bits for the identifier is too small.

NAME

gethostname, sethostname – get/set name of current host

SYNOPSIS

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

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

DESCRIPTION

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

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

RETURN VALUE

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

ERRORS

The following errors may be returned by these calls:

[EINVAL]	The *namelen* parameter is unreasonable. Currently, there is a limit of 64 characters for the hostname.
[EFAULT]	The *name* parameter gave an invalid address.
[EPERM]	The caller was not the superuser.

SEE ALSO

gethostid(2), hostname(1)

BUGS

Host names are limited to MAXHOSTNAMELEN (from *<sys/param.h>*) characters, currently 64.

NAME

getitimer, setitimer – get/set value of interval timer

SYNOPSIS

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

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

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

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

DESCRIPTION

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

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

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

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

Time values smaller than the resolution of the system clock are rounded up to 10 milliseconds on the CONVEX-1.

The ITIMER_REAL timer decrements in real time. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

NOTES

Three macros for manipulating time values are defined in *<sys/time.h>*. *timerclear* sets a time value to zero, *timerisset* tests if a time value is non-zero, and *timercmp* compares two time values (beware that >= and <= do not work with this macro).

RETURN VALUE

If the calls succeed, a value of 0 is returned. If an error occurs, the value –1 is returned, and a more precise error code is placed in the global variable *errno*.

ERRORS

The possible errors are:

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

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

SEE ALSO

sigvec(2), gettimeofday(2)

NAME

getpagesize – get system page size

SYNOPSIS

pagesize = getpagesize()
int pagesize;

DESCRIPTION

Getpagesize returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a *system* page size and may not be the same as the underlying hardware page size.

SEE ALSO

brk(2), pagesize(1)

NAME

getpattr, setpattr – get/set process attributes

SYNOPSIS

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

getpattr ( pid, pattrib )
int pid;
struct pattributes *pattrib;

setpattr ( pid, pattrib )
int pid;
struct pattributes *pattrib;
```

DESCRIPTION

getpattr returns the process attributes associated with the process specified by *pid*.

setpattr will set the specified processes attributes as defined in the attributes structure.

There are currently four supported attributes;

pattr_pfixed
When set to one specifies that the process must always be scheduled with all the CPUs available for it's use.

pattr_login
When set to one specifies that the process is an initial login process. This attribute can only be changed by the superuser, and is silently ignored when attempted to be set by a non-superuser.

pattr_pvtime
When set to one specifies that the process has the process virtual timer enabled. The process virtual timer cannot be enabled for a process while it is multithreaded (multiple threads of execution).

pattr_dmonnoblock
When set to one specifies that a user wants an error when trying to access a migrated file. If clear, then the process wants transparent access to migrated files, and is willing to be blocked if necessary.

RETURN VALUE

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

ERRORS

The possible errors are:

[EFAULT]	The address specified for *pattrib* is invalid.
[ESRCH]	The specified *pid* is not a valid process id.
[EPERM]	The caller does not have the correct permissions to access the specified *pid*.

SEE ALSO

open(2), fcntl(2)

NAME

getpeername – get name of connected peer

SYNOPSIS

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

getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

Getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

accept(2), bind(2), socket(2), getsockname(2)

BUGS

Names bound to sockets in the UNIX domain are inaccessible; *getpeername* returns a zero length name. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

NAME

getpflags, setpflags – get/set process entry flags

SYNOPSIS

flags = getpflags(pid)
int flags;

setpflags(newflags, mask)
int newflags;
int mask;

WARNING

The *getpflags* and *setpflags* system calls have been removed from the operating system. They are replaced by *getpattr* and *setpattr*. This man page is left for reference in converting existing applications.

DESCRIPTION

getpflags returns the process flags associated with the process specified in *pid*.

setpflags sets the bits indicated by *mask* to the values given in *newflags*. The exact operation is as follows:

```
proc[pid].p_flag &= ~mask;
proc[pid].p_flag |= (newflags & mask);
```

This allows the caller to specify exactly which bits they wish to modify without the necessity to first get the flags and then make the call with the properly adjusted flags. The reason that the latter course is undesirable is that some of the process flags may have changed since the flags were obtained. Only the superuser can use this system call and extreme caution should be exercised in using it at all.

RETURN VALUE

If the call succeeds, *getpflags* returns the process flags. *setpflags* returns a value of **0**. If the call fails, *both* return a value of **–1** and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EINVAL] The *pid* is not a valid process id.

[EPERM] The caller of *setpflags* was not the superuser.

SEE ALSO

getpid(2), getppid(2)

NAME

getpgrp – get process group

SYNOPSIS

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

pid_t pgrp;
pgrp = getpgrp();
```

DESCRIPTION

The process group of the current process is returned by *getpgrp()*.

Process groups are used for distribution of signals and by the terminal driver to arbitrate requests for terminal input. Processes that are in the controlling process group are foreground and may read, while others will block with a signal if they attempt to read.

This call is used by programs such as *csh*(1) to create process groups in implementing job control. The *tcsetpgrp*(3) and *tcgetpgrp*(3) functions are used to get/set the process group of the controlling terminal.

BACKWARD COMPATIBILITY

Previous versions of the operating system specified an argument *(pid)* to *getpgrp()*. A value of zero meant the current process. This mechanism is supported only in backward compatible binary mode.

SEE ALSO

setpgrp(2), getuid(2), tty(4)

NAME

getpid, getppid – get process identification

SYNOPSIS

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

pid_t pid;
pid = getpid();

pid_t ppid;
ppid = getppid()
```

DESCRIPTION

getpid() returns the process ID of the current process. Most often it is used with the host identifier *gethostid*(2) to generate uniquely-named temporary files.

getppid() returns the process ID of the parent of the current process.

RETURNS

getpid() and *getppid()* are always successful; there is no error return value.

SEE ALSO

gethostid(2)

NAME

getpriority, setpriority – get/set program scheduling priority

SYNOPSIS

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

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

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

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

DESCRIPTION

The scheduling priority of the process, process group, or user, as indicated by *which* and *who* is obtained with the *getpriority* call and set with the *setpriority* call. *which* is one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, and *who* is interpreted relative to *which* (a process identifier for PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user ID for PRIO_USER).

prio is a value in the range –64 to 64. (Values less than –64 are interpreted as -64; likewise values greater than 64 are interpreted as 64.) The default priority is 0; lower priorities cause more favorable scheduling. Internally in the kernel the priority is mapped into one of 64 run queues. The kernel always schedules the process on the lowest queue that is ready to run. Priorities can be interpreted as being in one of three groups. *Fixed high priorities* are in the range of –64 to –33. Their run queue number is computed by 32+(0.5*priority), which results in queue values 0 to 15. Priorities in the range of –32 to 32 are *variable priority,* meaning their queue value depends on the amount of recent CPU utilization. Processes which use lots of CPU cycles find their queue values incrementing; thus scheduling favors interactive, I/O bound jobs. The equation for queue value is 32+(cpu/12)+(0.75*priority), resulting in a range of 8 to 56. *Fixed low priorities* are in priority range 33 to 64. Their run queue value is computed by 32+(0.5*priority), resulting in queue values ranging from 48 to 63. Since variable priority processes never get queue values higher than 56, it is possible to place a process on a priority such that it runs only if no other process wants to (and thus consumes only otherwise idle CPU cycles.) (Note that future releases of ConvexOS may compute internal queue priorities differently).

CAUTION – Use the extremities of the priority ranges with great care! Internal system processes such as the swapper and the page daemon run on queue values 0 to 10. The careless use of high priorities will cause unacceptable response time for other users, and may cause system deadlocks or hangs. Fixed high priorities are designed solely for real time jobs, whose response time must be deterministic. Fixed low priorities will not provide acceptable response times for interactive jobs.

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

RETURN VALUE

Since *getpriority* can legitimately return the value **–1**, it is necessary to clear the external variable *errno* prior to the call, then check it afterward to determine if a **–1** is an error or a legitimate value. The *setpriority* call returns **0** if there is no error, or **–1** if there is.

ERRORS

Getpriority and *setpriority* may return one of the following errors:

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

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

In addition to the errors indicated above, *setpriority* may fail with one of the following errors returned:

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

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

SEE ALSO

nice(1), fork(2), renice(8)

NAME

getrlimit, setrlimit – control maximum system resource consumption

SYNOPSIS

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

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

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

DESCRIPTION

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

The *resource* parameter is one of the following:

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

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

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

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

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

RLIMIT_RSS the maximum size, in bytes, a process's resident set size may grow to. This imposes a limit on the amount of physical memory to be given to a process; if memory is tight, the system will prefer to take memory from processes which are exceeding their declared resident set size.

RLIMIT_CONCUR
the maximum number of processors that may be allocated to any process.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded a process may receive a signal (for example, if the cpu time is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The *rlimit* structure is used to specify the hard and soft limits on a resource,

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

Only the super-user may raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*. Both *rlim_cur* and *rlim_max* must be greater than zero when the *resource* is RLIMIT_CONCUR.

An "infinite" value for a limit is defined as RLIM_INFINITY (0x7fffffff).

Because this information is stored in the per-process information, this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to *csh*(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way: a *break* call fails if the data space limit is reached, or the process is killed when the stack limit is reached (since the stack cannot be extended, there is no way to send a signal!).

A file i/o operation which would create a file which is too large will cause a signal SIGXFSZ to be generated; this normally terminates the process, but may be caught. When the soft cpu time limit is exceeded, a signal SIGXCPU is sent to the offending process.

RETURN VALUE

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

ERRORS

The possible errors are:

[EFAULT]	The address specified for *rlp* is invalid.
[EINVAL]	The limits specified to *setrlimit* contained an invalid value.
[EPERM]	The limit specified to *setrlimit* would have raised the maximum limit value, and the caller is not the super-user.

SEE ALSO

csh(1)

BUGS

There should be *limit* and *unlimit* commands in *sh*(1) as well as in *csh*.

NAME

getrusage – get information about resource utilization

SYNOPSIS

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

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

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

DESCRIPTION

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

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

The fields are interpreted as follows:

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

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

ru_maxrss the maximum resident set size utilized (in kilobytes).

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

ru_idrss an integral value of the amount of unshared memory residing in the data segment of a process (expressed in units of megabytes * clock-ticks-of-execution).

ru_isrss an integral value of the amount of unshared memory residing in the stack segment of a process (expressed in units of megabytes * clock-ticks-of-execution).

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

ru_majflt the number of page faults serviced which required I/0 activity.

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

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

ru_oublock the number of times the file system had to perform block output.

ru_msgsnd the number of *ipc* messages sent.

ru_msgrcv the number of *ipc* messages received.

ru_nsignals the number of signals delivered.

ru_nvcsw the number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).

ru_nivcsw the number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

ru_exutime the total amount of time spent executing in user mode with microsecond accuracy.

NOTES

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

SEE ALSO

gettimeofday(2), wait(2),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

BUGS

There is no way to obtain information about a child process which has not yet terminated.

The resident memory sizes should not depend on system clock speed.

NAME

getsockname – get socket name

SYNOPSIS

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

getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

getsockname returns the current *name* for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return, it contains the actual size of the name returned in bytes.

DIAGNOSTICS

A **0** is returned if the call succeeds, **–1** if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

bind(2), socket(2)

BUGS

Names bound to sockets in the UNIX domain are inaccessible; *getsockname* returns a zero length name. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

NAME

getsockopt, setsockopt – get and set options on sockets

SYNOPSIS

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

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

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

DESCRIPTION

Getsockopt and *setsockopt* manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see *getprotoent*(3N).

The parameters *optval* and *optlen* are used to access option values for *setsockopt*. For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is returned, *optval* may be supplied as 0.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file *<sys/socket.h>* contains definitions for "socket" level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (4P).

Most socket-level options take an *int* parameter for *optval*. For *setsockopt*, the parameter should non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a *struct linger* parameter, defined in *<sys/socket.h>*, which specifies the desired state of the option and the linger interval (see below).

The following options are recognized at the socket level. Except as noted, each may be examined with *getsockopt* and set with *setsockopt*.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind*(2) call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are

directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a *close*(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the *close* attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close* is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with *recv* or *read* calls without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values. Finally, SO_TYPE and SO_ERROR are options used only with *setsockopt*. SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUE

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	The option is unknown at the level indicated.
[EFAULT]	The address pointed to by *optval* is not in a valid part of the process address space. For *getsockopt*, this error may also be returned if *optlen* is not in a valid part of the process address space.
[EINVAL]	*optval* is not a valid pointer.
[EINVAL]	*level* is negative.
[ENOBUFS]	The option SO_SNDBUF or SO_RCVBUF was requested with *optval* pointing to an invalid buffer size (set only).

SEE ALSO

ioctl(2), socket(2), getprotoent(3N), ip(4P), tcp(4P)

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

getsysinfo – get system information

SYNOPSIS

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

getsysinfo(SYSINFO_SIZE, sysinfo)
struct        system_information  *sysinfo;
```

DESCRIPTION

Getsysinfo returns information describing the configuration of the CONVEX Computer System on which you are running. This system call is specific to the ConvexOS operating system.

SYSINFO_SIZE is defined in *<sys/sysinfo.h>* as the size of the *system_information* structure. This value is used as a protection mechanism when more information is provided in the *system_information* structure. If older software is run against a newer, larger version of *struct system_information,* the **SYSINFO_SIZE** value supplied will be the number of bytes of *system_information* placed in *sysinfo.*

The buffer *sysinfo* points to will be filled in with the following structure:

```
struct system_information {
        unsigned short  system_sn;    /* serial number of system */
        unsigned char   cpu_type;     /* type of CPUs in complex */
        unsigned char   cpu_count;    /* # CPUs in complex       */
        long long       flags;        /* flags                   */
};
```

The fields are interpreted as follows. More complete definitions of the fields may be found in */usr/include/sys/sysinfo.h.*

system_sn	serial number of the system on which the program is running.
cpu_type	indicates the CPU type of the CPU(s) in the complex
cpu_count	indicates the number of CPU(s) in the complex
flags	various flags indicating hardware and software capabilities.

RETURN VALUE

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

ERRORS

The following error code may be set in *errno* :

[EINVAL] SYSINFO_SIZE is less than or equal to zero.

SEE ALSO

gethostid(2), gethostname(2), uname(2)

NAME

gettimeofday, settimeofday – get/set date and time

SYNOPSIS

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

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

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

DESCRIPTION

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

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

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

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

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

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

RETURN

A 0 return value indicates that the call succeeded. A –1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] An argument address referenced invalid memory.

[EPERM] A user other than the super-user attempted to set the time.

SEE ALSO

date(1), ctime(3)

NAME

getuid, geteuid – get user identity

SYNOPSIS

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

uid_t uid;
uid = getuid()

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

uid_t euid;
euid = geteuid()
```

DESCRIPTION

getuid() returns the real user ID of the current process, and *geteuid()* returns the effective user ID.

The real user ID identifies the person who is logged in. The effective user ID gives the process additional permissions during execution of "set-user-ID" mode processes, that use *getuid()* to determine the real-user-id of the process which invoked them.

RETURNS

These functions are always successful; there is no return value which indicates an error.

SEE ALSO

getgid(2), setreuid(2)

NAME

ioctl – control device

SYNOPSIS

#include <sys/ioctl.h>

ioctl(d, request, argp)
int d, request;
char *argp;

DESCRIPTION

Ioctl performs a variety of functions on open descriptors. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with *ioctl* requests. The write-ups of various devices in section 4 discuss how *ioctl* applies to them.

An ioctl *request* has encoded in it whether the argument is an "in" parameter or "out" parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an ioctl *request* are located in the file *<sys/ioctl.h>*.

RETURN VALUE

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

ERRORS

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

[EBADF]	*D* is not a valid descriptor.
[ENOTTY]	*D* is not associated with a character special device.
[ENOTTY]	The specified request does not apply to the kind of object which the descriptor *d* references.
[EINVAL]	*Request* or *argp* is not valid.
[EPERM]	Only superusers may perform the specified ioctl.

SEE ALSO

execve(2), fcntl(2), ta(4), tty(4), intro(4N)

NAME

kill – send signal to a process

SYNOPSIS

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

DESCRIPTION

kill() sends the signal *sig* to a process specified by *pid*. *sig* may be either one of the signals specified in *sigvec*(2) or 0, in which case error checking is performed, but no signal is actually sent. This can be used to check the validity of *pid*.

For a process to have permission to send a signal, the real or effective user ID of the sending process must match the real or saved set-user-ID of the receiving process; otherwise, this call is restricted to the super-user. A single exception is the signal SIGCONT that may always be sent to any member of the same session as the sender.

If the process number is 0, the signal is sent to all other processes in the senders process group; this is a variant of *killpg*(2).

If the process number is –1 and the user is the super-user, the signal is broadcast universally except to system processes and the process sending the signal. If the process number is –1 and the user is not the super-user, the signal is broadcast universally to all processes with the same UID as the user except the process sending the signal. No error is returned if any process could be signaled.

For compatibility with System V, if the process number is negative but not –1, the signal is sent to all processes whose process group ID is equal to the absolute value of the process number. This is a variant of *killpg*(2).

Processes may send signals to themselves.

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

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

[EINVAL]	*sig* is not a valid signal number.
[ESRCH]	No process can be found corresponding to that specified by *pid*.
[ESRCH]	The process ID was given as 0 but the sending process does not have a process group.
[EPERM]	The sending process is not the super-user and its real or effective user ID does not match the real or save-set-user-ID of the receiving process. When signaling a process group, this error is returned if any members of the group could not be signaled.
[EACCES]	The receiving process is a system process.

BACKWARD COMPATIBILITY

Previous versions of the operating system based the permission test solely on a match of the effective UID's of the sender and receiver.

SEE ALSO

kill(1), getpid(2), getpgrp(2), killpg(2), sigvec(2)

NAME

killpg – send signal to a process group

SYNOPSIS

killpg(pgrp, sig)
int pgrp, sig;

DESCRIPTION

Killpg sends the signal *sig* to the process group *pgrp*. See *sigvec*(2) for a list of signals.

The sending process and members of the process group must have the same effective user ID, otherwise this call is restricted to the super-user. As a single special case the continue signal SIGCONT may be sent to any process which is a descendant of the current process.

RETURN VALUE

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

ERRORS

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

[EINVAL]	*Sig* is not a valid signal number.
[ESRCH]	No process can be found corresponding to that specified by *pgrp*.
[EPERM]	The sending process is not the super-user and one or more of the target processes has an effective user ID different from that of the sending process.

SEE ALSO

kill(2), getpgrp(2), sigvec(2)

NAME

krpc – daemon interface to kernel RPC facility

SYNOPSIS

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

krpc(fd, seqno, event, repargs, replen, callargs, callen)
int      fd;
caddr_t  seqno;
caddr_t  event;
caddr_t  repargs;
u_int    replen;
caddr_t  callargs;
u_int    callen;
```

DESCRIPTION

The *krpc* system call is the server interface to a local RPC facility. Only the superuser or a process in group *daemon* may use *krpc*.

The following list explains each argument:

fd	The file descriptor of a valid krpc channel
seqno	The sequence number of the transaction.
event	The event that occurred.
repargs	The address of the structure to contain reply arguments.
replen	The actual size of the reply arguments buffer.
callargs	The address of the structure containing the calling arguments.
callen	The maximum expected size of the calling arguments.

Before *krpc* can be used, servers must obtain a valid communication channel. (See *dmon_ioctl(2)* for information on obtaining a channel).

krpc services RPC requests by accepting RPC messages for the specified channel and sending back replies. A new message may be received and a reply returned to a previous message, depending on the value of *callargs* and *repargs*.

If *callargs* is not NULL, the request arguments are copied into the address given by *callargs*. If there are RPC messages waiting, the *krpc* call returns immediately. If there are no RPC messages waiting, the caller waits until an RPC request is received. When a message is received, it's sequence number is copied into the address given by *seqno*. This sequence number must be given when sending a reply to the message.

If *callargs* is NULL, no messages are received. Outstanding messages will still be available to subsequent calls.

Replies work in a similar manner, depending up the value of *repargs*. If *repargs* is NULL, no reply is sent. If *repargs* is not NULL, a reply is sent to the client that generated the message with sequence number *seqno*. The reply arguments are pointed to by *repargs*; these are copied in and sent with the reply.

The calling arguments buffer *callargs* must be large enough to hold the maximum possible message. This size is specified by *cargslen*.

If both *callargs* and *repargs* are not NULL, the reply is sent before attempting to process the call request.

If *callargs* is not NULL, the call to *krpc* blocks indefinitely until a message is received. The *select*(2) system call may be used to determine the availability of a message or whether a client is waiting for a message. A *select*(2) returns true for read queries if there is a message waiting for the server. Similarly, *select*(2) returns true for write queries if there is a client waiting on a reply from the server.

When the krpc channel referenced by *fd* is closed, no further communication with a server daemon is possible. If the channel referenced by *fd* is the primary channel that is used by the daemon to communicate with the kernel (i.e., it was the default channel specified when the daemon was

initialized), then closing the file descriptor *fd* is equivalent to shutting down the daemon.

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

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

[EBADF]	The file descriptor referenced by *fd* is invalid.
[EBADCHAN]	The channel referenced by the file descriptor *fd* is not a valid krpc channel.
[EFAULT]	One of the address arguments points outside the process's allocated address space.
[EINTR]	A signal was delivered before the service was used.
[EPERM]	The calling process is not the super-user and its effective group id is not daemon.
[ESRCH]	The sequence number for a reply was not found.
[EINVAL]	On a reply, the message that matched a sequence number was invalid.

SEE ALSO

dmon_ioctl(2)

NAME

krpc_open – open a KRPC channel with the kernel

SYNOPSIS

int krpc_open()

DESCRIPTION

The *krpc_open()* system call allocates a krpc channel in the kernel and returns a file descriptor for that channel. The channel can then be registered with certain kernel modules and used for communication and remote procedure calls.

RETURN VALUE

The value -1 is returned if an error occurs, and external variable *errno* is set to indicate the cause of the error. Otherwise a non-negative numbered file descriptor for the new open channel is returned.

ERRORS

krpc_open() fails if one or more of the following are true:

[EPERM]	The calling process is not the super-user and its effective group id is not daemon.
[EMFILE]	The system limit for open file descriptors per process has already been reached.
[ENFILE]	The system file table is full.

SEE ALSO

krpc(2)

NAME

link – make a hard link to a file

SYNOPSIS

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

DESCRIPTION

A hard link to *name1* is created; the link has the name *name2*. *name1* must exist.

With hard links, both *name1* and *name2* must be in the same file system. Unless the caller is the super-user, *name1* must not be a directory. Both the old and the new *link* share equal access and rights to the underlying object.

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

link() will fail and no link will be created if one or more of the following are true:

[EACCES] A component of either path prefix denies search permission.
The requested link requires writing in a directory with a mode that denies write permission.

[EEXIST] The link named by *name2* does exist.

[EINVAL] Either pathname contains a byte with the high-order bit set.

[EMLINK] The number of links to *name1* would exceed LINK_MAX for the file system containing *name1*.

[ENAMETOOLONG]
The length of *name1* or *name2* exceeds PATH_MAX for the file system, or a pathname component exceeds NAME_MAX for a file system with _POSIX_NO_TRUNC in effect.

[ENOENT] A component of either path prefix does not exist.
The file named by *name1* does not exist.
name1 or *name2* points to an empty string.

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

[EPERM] The file named by *name1* is a directory.

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

[EXDEV] The link named by *name2* and the file named by *name1* are on different file systems.

[EFAULT] One of the pathnames specified is outside the process's allocated address space.

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

BACKWARD COMPATIBILITY

Previous versions of the operating system allowed the empty path string as a synonym for the current directory.

Previous versions of the operating system allowed root to create hard links to directories.

SEE ALSO

symlink(2), unlink(2)

NAME

listen – listen for connections on a socket

SYNOPSIS

listen(s, backlog)
int s, backlog;

DESCRIPTION

To accept connections, a socket is first created with *socket*(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with *listen*(2), and then the connections are accepted with *accept*(2). The *listen* call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

RETURN VALUE

A 0 return value indicates success; –1 indicates an error.

ERRORS

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the operation *listen*.

SEE ALSO

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

BUGS

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

NAME

lseek, lseek64 – move read/write pointer

SYNOPSIS

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

off_t lseek(d, offset, whence)
int   d, whence;
off_t offset;

off64_t lseek64(d, offset, whence)
int d, whence;
off64_t offset;
```

DESCRIPTION

The descriptor *d* refers to a file or device open for reading and/or writing. Both *lseek*() and *lseek64*() sets the file pointer of *d* as follows:

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

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

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

Upon successful completion, the resulting pointer location as measured in bytes from beginning of the file is returned. Some devices are incapable of seeking. The value of the pointer associated with such a device is undefined.

While both *off_t* and *off64_t* are signed types, some devices (e.g. */dev/mem*) and the process interface (see *pi*()) treat *offset* as an unsigned value. When *offset* is being interpreted as a memory address, only the low order 32 bits are significant. The high order 32 bits of *off64_t* offsets are ignored in such cases.

lseek will fail when *whence* is SEEK_CUR and the current value of the offset is greater than will fit in an *off_t*.

Even though internally file offsets are 64 bits, overflow of the offset when using *lseek* produces a negative offset. This is an error in most cases. Those in which it isn't are treating *offset* as unsigned, so the offset isn't really negative (merely large) and there wasn't really any overflow in the first place.

RETURN VALUE

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

ERRORS

Both *lseek*() and *lseek64*() will fail and the file pointer will remain unchanged if:

[EBADF]	*d* is not an open file descriptor.
[EINVAL]	*whence* is not a proper value.
[EINVAL]	The resulting file pointer would be negative as expressed in the offset type (either *off_t* or *off64_t*) and *d* does not allow negative offsets
[ESPIPE]	*d* is associated with a pipe or a socket.

NOTES

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

A successful call to *lseek64*() will cause the **O_LARGEFILE** bit on the file descriptor to be set.

SEE ALSO

dup(2), fseek(3s), open(2) "ConvexOS Large Files User's Guide"

NAME

lstat – get file status

SYNOPSIS

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

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

lstat64(path, buf)
char *path;
stat64_t *buf;
```

DESCRIPTION

lstat and *lstat64* obtain information about the file named by *path*. 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.

lstat and *lstat64* operate like *stat*(2) except in the case where the named file is a symbolic link, in which case *lstat* returns information about the link while *stat* returns information about the file the link references.

buf is a pointer to a *stat* structure into which information is placed concerning the file. The contents of the structure pointed to by *buf* include the following members:

```
struct stat {
        dev_t     st_dev;     /* device inode resides on */
        ino_t     st_ino;     /* this inode's number */
        mode_t    st_mode;    /* protection */
        nlink_t   st_nlink;   /* number of hard links to the file */
        uid_t     st_uid;     /* user-id of owner */
        gid_t     st_gid;     /* group-id of owner */
        off_t     st_size;    /* total size of file */
        time_t    st_atime;   /* file last access time */
        time_t    st_mtime;   /* file last modify time */
        time_t    st_ctime;   /* file last status change time */
        /* The following are CONVEX extensions */
        dev_t     st_rdev;    /* the device type, for inode that is device */
        long      st_blksize; /* optimal blocksize for file system i/o ops */
        long      st_blocks;  /* actual number of blocks allocated */
};
```

st_atime Time when file data was last read. Changed by the following system calls: *mknod*(2), *utimes*(2), *read*(2), and *readlink*(2). When a directory is searched st_atime for the directory is set.

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

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

st_mode The attributes of st_mode may be queried with the following macros:

S_ISDIR(st.st_mode)	True if a directory
S_ISCHR(st.st_mode)	True if a character special device
S_ISBLK(st.st_mode)	True if a block special device
S_ISREG(st.st_mode)	True if a regular file

S_ISFIFO(st.st_mode) True if a pipe or fifo special file

_S_ISLNK(st.st_mode) True if a symbolic link

_S_ISSOCK(st.st_mode) True if a socket

The permissions associated with st_mode may be extracted with the S_IRWXU, S_IRWXG, and S_IRWXO masks. The set-user-ID bit is S_ISUID and the set-group-ID bit is S_ISGID. Refer to *chmod*(2) for more details. The file mode bits will contain _S_ISVTX if the sticky bit is set (see sticky(8)).

st_blksize The block size of the file system. See *df*(1).

st_blocks Number of 512 byte disk sectors allocated to the file. Note that fragments are always an integral multiple of 512 bytes, and that storage for files is allocated in multiples of the fragment size (or block size, for files sizes greater than 12 times the block size). This means more sectors may be allocated to a file than would appear necessary as determined by dividing the file size by 512.

lstat64 take a *cvxstat* structure in place of the *stat* structure. For convience, the *cvxstat* structure has been typedef'ed to *stat64_t*.

The *stat* structure and the *stat64_t* structure have identical members, with the exception that the *st_size* member is an *off64_t* instead of an *off_t* thus providing size information for files larger than two gigabytes.

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

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

[ENOENT] *path* points to an empty string.

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

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG]
The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters. The file referred to by *path* does not exist.

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

[ELOOP] Too many symbolic links were encountered in translating *path*.

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

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

BACKWARD COMPATIBILITY

In previous versions of the operating system, the empty pathname string was a synonym for the current directory.

The file status information was previously expressed as follows:

```
#define  S_IFMT     0170000   /* type of file */
#define  S_IFIFO    0010000   /* fifo special */
#define  S_IFCHR    0020000   /* character special */
#define  S_IFDIR    0040000   /* directory */
#define  S_IFBLK    0060000   /* block special */
#define  S_IFREG    0100000   /* regular */
#define  S_IFLNK    0120000   /* symbolic link */
#define  S_IFSOCK   0140000   /* socket */
#define  S_ISUID    0004000   /* set user id on execution */
#define  S_ISGID    0002000   /* set group id on execution */
#define  S_ISVTX    0001000   /* see sticky(8) */
#define  S_IREAD    0000400   /* read permission, owner */
#define  S_IWRITE   0000200   /* write permission, owner */
```

```
#define    S_IEXEC        0000100        /* execute/search permission, owner */
```

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

SEE ALSO

chmod(2), chown(2), fstat(2), readlink(2), stat(2), utimes(2)

NAME

mkdir – make a directory file

SYNOPSIS

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

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

DESCRIPTION

mkdir() creates a new directory file with name *path*. The mode of the new file is initialized from *mode*.

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.

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:

[EACCES]	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be created.
[EEXIST]	The named file exists.
[EINVAL]	The *path* argument contains a byte with the high-order bit set.
[EMLINK]	The link count of the parent directory would exceed LINK_MAX for that file system.
[ENAMETOOLONG]	The length of *path* exceeds PATH_MAX for the file system.
[ENAMETOOLONG]	The length of a pathname component exceeds NAME_MAX and the file system enforces _POSIX_NO_TRUNC.
[ENOENT]	A component of the path prefix does not exist. The *path* argument points to the empty string.
[ENOSPC]	The file system does not contain enough space to hold the contents of the new directory or to expand the parent directory.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while writing to the file system.

SEE ALSO

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

NAME

mknod – make a special (character, block, or fifo) file

SYNOPSIS

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 (including file type bits) is initialized from *mode*. The values of the file type bits which are permitted are:

```
#define   S_IFCHR     0020000     /* character special */
#define   S_IFBLK     0060000     /* block special */
#define   S_IFREG     0100000     /* regular */
#define   S_IFIFO     0010000     /* FIFO special */
```

Values of *mode* other than those above are undefined and should not be used.

The protection part of the mode is modified by the process's mode mask; (see *umask*(2)).

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the group ID of the parent directory.

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.

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

mknod fails and the file mode will be unchanged if:

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

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG] The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

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

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

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EPERM] An attempt was made to create a file of type other than FIFO special and the process's effective user ID is not super-user.

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

[EISDIR] The specified *mode* would have created a directory.

[ENOSPC] The directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] There are no free inodes on the file system on which the file is being created.

[EDQUOT] The directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the node is being created has been exhausted.

[EROFS] The file referred to by *path* resides on a read-only file system.

[EEXIST] The file referred to by *path* exists.

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

SEE ALSO

chmod(2), stat(2), umask(2), mknod(8)

NAME

mmap – map shared memory into your address space

SYNOPSIS

```
#include      <sys/types.h>
#include      <sys/mman.h>
caddr_t
mmap(addr, len, prot, share, fd, offset)
caddr_t addr; unsigned *len, prot, share;
int fd; off_t offset;
```

DESCRIPTION

The *mmap* system call maps shared memory into the calling process' address space. Shared memory segments may be mapped between the top of the data segment and the bottom of the stack segment. All processes which map the same shared memory segment into their address space are guaranteed to be accessing the same physical pages, even though at possibly different virtual addresses (if they are on the same machine). *mmap* does its memory mapping based upon the file-system block size.

The *mmap* parameters specify how the mapping is to be done. Many of them assume logical default values– see below. *addr* specifies the desired virtual address of the mapping. *len* specifies the length of the segment, which may be larger than the mapped object. The actual length of the mapped object is returned in *len*.

prot specifies the desired page protection bits in the page table entries, and is composed of the logical-OR of PROT_READ, PROT_WRITE, and PROT_EXEC.

share is composed of the logical-OR of the mapping type, and various mapping options. There are four mapping types (or domains)– MAP_FILE, MAP_ANON, MAP_THREAD, and MAP_DEVICE. The available options (described below) include MAP_FIXED, MAP_INHERIT, MAP_HASSEMAPHORE, MAP_EXTEND, MAP_SHARED, MAP_PRIVATE, MAP_FILLFD, MAP_GROWAUTO, MAP_GROWUP, MAP_GROWDOWN, and MAP_DEBUG.

fd is a file descriptor, that identifies the file or character device to be mapped, or names an area of swap space that is mapped. *offset* is the offset in the mapped object at which mapping begins. It must be a multiple of NBPG, the number of bytes per page.

In the MAP_FILE domain, the argument *fd* is the file descriptor of a regular file, which will in effect, be mapped into your virtual address space. Pages in the mapped area which are read will automatically be initialized to the contents of the file. If *share* is MAP_PRIVATE, dirty (written into) pages are replicated and made private to the writing process. Dirty, private pages are paged to the swap area. The regular file is left undisturbed. If MAP_SHARED, all processes writing into pages in the mapped area are altering the same physical pages. Dirty, shared pages are paged to the regular file irregularly, if at all.

The *msync* system call flushes dirty, shared pages to the buffer cache. Subsequent *sync* and *fsync* system calls will process the data for the file. When the last process using a shared memory segment in the MAP_FILE domain unmaps the segment, an automatic *msync* of the mapped area occurs.

It is possible to enlarge the file with write accesses (see below.) If a file is opened for writing, it can get extended to the next multiple of the file-system block size. It is not necessary that the entire file be mapped into the process' address space; the *offset* parameter specifies the byte offset in the file which corresponds to the first byte in the mapped segment.

In the MAP_ANON domain, the argument *fd* is the file descriptor of a regular file. This regular file is not disturbed. Pages are initialized to zero, and paged to a temporary file in /swap if *share* is MAP_SHARED, or paged to the swap area if *share* is MAP_PRIVATE. Cooperating processes can MAP_SHARE the same segment (and so share the same temporary file), by mapping the same regular file in the MAP_ANON domain. After the last process attached to a shared memory segment in the MAP_ANON domain unmaps the segment, the contents of the segment are lost, and, if present, the temporary file is deleted. Currently, a segment in the MAP_ANON domain cannot be enlarged after the initial mmap call. The *offset* argument should be zero for the MAP_ANON domain.

In the MAP_THREAD domain, the argument *fd* is the file descriptor of a regular file, which will in effect, be mapped into your virtual address space. Pages in the mapped area which are read will automatically be initialized to the contents of the file. If *share* is MAP_PRIVATE, dirty (written into) pages are replicated and made private for each writing thread of your process. Dirty, private pages are paged to the swap area. The regular file is left undisturbed. MAP_SHARED is not currently allowed. It is not necessary that the entire file be mapped into the process' address space; the *offset* parameter specifies the byte offset in the file which corresponds to the first byte in the mapped segment.

The MAP_DEVICE domain supports access to kernel and physical memory directly, and may in the future be used to implement device drivers with buffers mapped directly into user memory. *fd* must be a character special device that has a map entry point in its device driver. The /dev/mem and /dev/kmem devices allow the mapping of physical and kernel virtual memory into user address space. No other devices currently support the map entry point. The *offset* argument is the physical or kernel virtual address that is mapped as the first byte in the shared segment. Paging is not possible from segments in the MAP_DEVICE domain, since physical and kernel virtual memory are always "resident". Mapping with MAP_DEVICE is inherently machine dependent.

A single *fd* cannot be used to map segments in different domains simultaneously; e.g. if a process uses the file with MAP_FILE another process cannot perform a MAP_ANON on that file.

The mapping options include the type (or domain); MAP_SHARED, which specifies that segments are to be shared with other processes mapping the same segment; MAP_PRIVATE, which specifies that changes made to the segment are private to this process. MAP_EXTEND, which is used to extend or lengthen files (see below); MAP_INHERIT, which specifies that shared memory segments are to be retained across the *exec* call; MAP_HASSEMAPHORE, that specifies the shared memory segment has one or more semaphores (see *msleep*(2)); MAP_FILLFD causes anonymous memory regions to use the file descriptor passed for initialization of pages but changes are paged to swap instead of back to the file as MAP_FILE does. MAP_DEBUG, which causes the kernel to print debug information on why calls to *mmap* fail; and MAP_FIXED, that specifies the kernel may not modify *addr* as it does the mapping.

The *prot* argument is the logical-OR of PROT_READ, PROT_WRITE, and PROT_EXEC, corresponding to the ability to read, write, or execute the mapped pages. Ability to read, write, or execute the mapped segment is granted only if the file identified by *fd* grants read, write, or execute permission to the calling process.

The shared memory segment is mapped into the process' virtual address space at *addr*, or if *addr* is zero, a default address is provided by the kernel, which is the address at which the segment is mapped by other processes (so that pointers will work.) Address 0xc0000000 is provided if no other processes have *fd* mapped. *addr* cannot lie within the process' text, data, or stack segments (see below). The kernel may arbitrarily round or change the address at which the segment will be mapped unless MAP_FIXED. If MAP_FIXED is specified an error will be returned if the user's specified *addr* is unsuitable.

The *len* argument is a pointer to an argument that specifies the length of the mapped segment; it represents the largest possible file in the MAP_FILE domain, or the largest possible memory region in the MAP_ANON domain. For a memory region with the MAP_GROWAUTO attribute, len specifies the limit for the memory region to grow very much as the *stacksize* of a process is limited. A mapped file may be smaller than *len*; the actual length of a mapped file, or the size of a single page for MAP_GROWAUTO regions, is returned in *len*. If *len* is zero for a segment in the MAP_FILE domain, the length of the mapped segment will be the length of the file specified by *fd* rounded up to the nearest page boundary.

References beyond the end of file may optionally extend the file if MAP_EXTEND is set in *share* and the domain is MAP_FILE, up to the limit in the *len* argument. If MAP_EXTEND is not set, bus error signals (SIGBUS) will be generated, which normally terminate the process. MAP_EXTEND currently has no meaning in the MAP_ANON domain.

Shared segments are preserved across *fork* and *vfork*, allowing children to reference the shared memory segments owned by their parents. Shared memory segments will be inherited across *exec* if and only if the MAP_INHERIT flag is set in *share*. Shared memory segments are unmapped as a result of process termination, *exit*, or *munmap*.

NOTES

It is not possible to *mmap*() regions of a file beyond the two gigabyte address point.

BUGS

Using *read* or *write* on a file that is simultaneously mapped in the MAP_FILE domain is not prohibited; however it is not guaranteed that file system accesses and shared memory accesses will see a consistent view of the file, because of simultaneous access by the system and the mapping processes.

It is not possible to *truncate* or *ftruncate* a file while it is mapped into a shared memory segment; any attempt to do so will be return an error.

RETURN VALUE & ERRORS

mmap returns a character pointer to the first byte in the mapped segment if the mapping was successful. If unsuccessful, mmap returns a -1 with errno set as follows:

[EINVAL] *offset* is not a multiple of NBPG (the number of bytes per page), or *offset* is negative, or *fd* is not a regular file for MAP_FILE domain, or *fd* is not a character special device for MAP_DEVICE domain, or *len* is zero and the domain is not MAP_FILE, or no domain was selected in the *share* argument, or the given *fd* is already mapped under a different domain, or the shared segment overlaps with another segment (see BUGS above).

[ESPIPE] *fd* is not an inode.

[ENODEV] The character special device *fd* does not have a map entry in the MAP_DEVICE domain.

[EBADF] *fd* was not a valid file descriptor.

[EACCES] The desired page protection (PROT_READ, PROT_WRITE, and PROT_EXEC) conflict with the permissions granted by *fd*.

[ENOENT] The directory /swap does not exist.

[ENOMEM] There is insufficient swap space or memory to grant the request.

SEE ALSO

munmap(2), msleep(2), mwakeup(2), msync(2), tas(3), mremap(2)

NAME

mount – mount file system

SYNOPSIS

```
#include <sys/types.h>
#include <sys/errno.h>
#include <rpc/types.h>
#include <nfs/nfs.h>
#include <sys/mount.h>

mount(type, dir, M_NEWTYPE|flags, data)
char *type;
char *dir;
int flags;
caddr_t data;
```

DESCRIPTION

mount attaches a file system to a directory. After a successful return, references to directory *dir* will refer to the root directory on the newly mounted file system. *dir* is a pointer to a null-terminated string containing a path name. *dir* must exist already, and must be a directory. Its old contents are inaccessible while the file system is mounted.

mount may be invoked only by the super-user.

The *flags* argument is constructed by the logical OR of the following bits (defined in **<sys/mount.h>**):

M_RDONLY	Mount filesystem read-only.
M_NOSUID	Ignore set-uid bit on execution.
M_NEWTYPE	If set, the type passed is a string such as *nfs* or *4.2*.
M_GRPID	Use **BSD** file-creation semantics (see *open(2)*).
M_REMOUNT	Change options on an existing mount.
M_NOSUB	Disallow mounts beneath this filesystem.

Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

The *type* indicates the type of the filesystem. It should be either the string *nfs* or *4.2* if *M_NEWTYPE* is specified, or a type from *mount.h* such as **MOUNT_UFS** or **MOUNT_NFS**. *data* is a pointer to a structure that contains the type specific arguments to mount. Below is a list of the filesystem types supported and the type specific arguments to each:

"4.2" or MOUNT_UFS

```
struct ufs_args {
     char    *fspec; /* block special file to mount */
     int     blkhi;  /* percentage of full disk for highwater mark */
     int     blklo;  /* percentage of full disk for lowater mark */
     long    ihi;    /* High water mark for inode allocation */
     long    ilo;    /* Lo water mark for inode allocation */
     u_int   migflags;       /* migration control flags */

};
```

"nfs" or MOUNT_NFS

```
#include      <nfs/nfs.h>
#include      <netinet/in.h>
struct nfs_args {
      struct sockaddr_in  *addr; /* file server address */
```

```
        fhandle_t       *fh;/* File handle to be mounted */
        int     flags; /* flags */
        int     wsize; /* write size in bytes */
        int     rsize; /* read size in bytes */
        int     timeo; /* initial timeout in .1 secs */
        int     retrans;/* times to retry send */
        char    *hostname;/* server's hostname */
        int     acregmin;/* attr cache file min secs */
        int     acregmax;/* attr cache file max secs */
        int     acdirmin;/* attr cache dir min secs */
        int     acdirmax;/* attr cache dir max secs */
        char    *netname;/* server's netname */
};
```

RETURN VALUE

mount returns 0 if the action occurred, and −1 if one of the errors below is detected.

ERRORS

mount will fail when one of the following occurs:

[EPERM] The effective uid of the caller is not root.

[EINVAL] The pathname of *fspec* (MOUNT_UFS) or *dir* contains a character with the high-order bit set.

[ENOTDIR] A component of the path prefix in *fspec* (MOUNT_UFS) or *dir* is not a directory.

[ENOENT] *fspec* (MOUNT_UFS) or *dir* does not exist.

[ENAMETOOLONG] The pathname of *fspec* (MOUNT_UFS) or *dir* is too long.

[EACCES] Search permission is denied for a component of the path prefix of *fspec* (MOUNT_UFS) or *dir*.

[ELOOP] Too many symbolic links were encountered in translating the pathname of *fspec* (MOUNT_UFS) or *dir*.

[ENODEV] *type* is invalid, or the kernel does not support the requested type.

[ENOTBLK] *fspec* (MOUNT_UFS) is not a block device.

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

[EPFNOSUPPORT] The address family specified in *addr* (MOUNT_NFS) is not supported for NFS file systems. (Currently only AF_INET is supported).

[EINVAL] The *wsize*, *rsize*, *timeo*, *retrans* specified in the *nfs_args* structure is invalid. (MOUNT_NFS)

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

[EBUSY] *fspec* (MOUNT_UFS) is already mounted.

[EBUSY] Insufficient space exists to hold the mount table entry or the cylinder group information for the file system.

[EBUSY] The super block for the file system contains bad data; i.e. it has a bad magic number or an out of range block size.

[EFAULT] One of the arguments passed to *mount* evaluates to an address outside the process's allocated address space.

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

[EAGAIN] The dirty bit was set in the superblock of the filesystem. Run *fsck*(8) on the device, then retry the mount.

SEE ALSO

unmount(2), mountd(8C), mount(8)

NAME

mremap – change attributes of a memory region in a process' address space

SYNOPSIS

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

mremap(addr, len, prot, share)
caddr_t addr;
unsigned *len, prot, share;
```

DESCRIPTION

The *mremap* system call allows a user to change the attributes for a segment (for example: bss, stack, data) of a process' address space. This includes regions previously mapped with *mmap*(2) as well as arbitrary segments of a process' text, bss, tbss, data, tdata, stack, or tstack regions.

addr is the beginning virtual address of the memory region to which the new attributes apply. *addr* must be a multiple of NBPG.

len specifies the length of the segment. If *len* is not a multiple of NBPG then it is rounded up to a multiple of NBPG. When the MAP_GROWAUTO option is specified in the *share* field, then *len* is used to specify the limit for the size of the region and contains the current size of the segment being remapped.

The *prot* specifies the desired page protection bits in the page table entries and is composed of the logical-OR of PROT_READ, PROT_WRITE, and PROT_EXEC. The PROT_READ, PROT_WRITE, and PROT_EXEC, correspond to the ability to read, write, or execute the remapped pages.

The *share* field is the logical or of the attributes desired for the memory segments within the range specified by *addr* and *len*. Valid options are MAP_SHARED, MAP_PRIVATE, MAP_GROWAUTO, MAP_GROWDOWN, MAP_GROWUP, and MAP_INHERIT. When the MAP_GROWAUTO Option is specified, only the segment that contains the address *addr* is remapped.

RETURN VALUE & ERRORS

mremap returns 0 if the remapping was successful. If unsuccessful, mremap returns a -1 with errno set as follows:

[EINVAL] *addr* points to an invalid address, attempted to remap a MAP_SHARED segment as MAP_PRIVATE, or share contains an option set not contained in the above list of valid options.

[EACCES] The specified protection conflicts with the permissions granted by the file corresponding to that mapped segment.

[ENOMEM] There is insufficient swap space or memory to grant the request.

SEE ALSO

mmap(2), munmap(2), msleep(2), mwakeup(2), msync(2), tas(3)
Using Shared Memory

NAME

msleep, mwakeup – shared memory synchronization primitives

SYNOPSIS

```
#include      <sys/types.h>
msleep(sem)
semaphore *sem;
mwakeup(sem)
semaphore *sem;
```

DESCRIPTION

These routines provide services analogous to the kernel sleep and wakeup functions interpreted on the domain of a shared file.

sem is a pointer to a machine dependent semaphore structure, defined in *sys/types.h*. On the CONVEX Computer, the structure is:

```
struct      semaphore   {     /* machine dependent */
      char  lock;       /* the tas lock */
      char  awakened;   /* waiters have been awakened */
      };
typedef struct semaphore semaphore;
```

sem must be in a shared memory region, with at least **PROT_READ** and **PROT_WRITE**. The **MAP_HASSEMAPHORE** flag must have been set when the *mmap* call was executed. The region may be mapped in any type– MAP_FILE, MAP_ANON, or MAP_DEVICE.

msleep arranges that the calling process relinquish the processor if the semaphore *lock* is set. If *lock* is not set, *msleep* returns immediately. The process will remain asleep until some other process issues an *mwakeup* on the same semaphore, or a signal is received. *mwakeup* awakens any processes waiting on the same semaphore. The *awakened* byte should not be modified by user programs; it is used by the kernel to know if someone is waiting on the semaphore for a wakeup.

User programs should use *mset* and *mclear* instead of *msleep* and *mwakeup* if possible, since the latter are less machine dependent.

RETURN VALUE & ERRORS

msleep and *mwakeup* return zero if successful. If unsuccessful, they return a -1 with errno set as follows:

[EFAULT] *sem* did not point to a valid address for the semaphore structure, or PROT_READ or PROT_WRITE were not set for the mapped region.

[EINTR] A signal was received which interrupted the system call.

SEE ALSO

mmap(2), mset(3), tas(3)

NAME

msync – synchronize shared memory segment with file system

SYNOPSIS

msync(addr,len)
caddr_t addr;
int len;

DESCRIPTION

The *msync* system call causes dirty pages in shared memory segments (mapped MAP_FILE) to be flushed back to the buffer cache, so that subsequent *sync* and *fsync* calls will process the correct data for the file. All the pages from *addr* to *addr+len* are flushed.

RETURN VALUE & ERRORS

msync returns zero if successful. If unsuccessful, *msync* returns a -1 with errno set as follows:

[EFAULT] The region defined by *addr* and *len* includes pages invalid (unmapped) pages.

SEE ALSO

mmap(2)

NAME

munmap – unmap memory

SYNOPSIS

```
#include      <sys/types.h>
#include      <sys/mman.h>
munmap(addr,size)
caddr_t addr;
int size;
```

DESCRIPTION

The munmap system call unmaps *size* bytes of memory beginning at *addr*. Subsequent references to virtual addresses in the unmapped area will generate bus errors (SIGBUS).

The actual range is *addr*, rounded down to a page boundary, plus *size*, rounded up to a page length.

The unmap range may be an entire memory segment, a partial memory segment, or multiple memory segments. Unmapped memory within the range is ignored, but *addr* must begin within a memory segment.

All memory, except thread memory, is eligible for unmapping. This includes shared memory, text, stack, data, and bss.

RETURN VALUE & ERRORS

munmap returns zero if successful. If unsuccessful, munmap returns a -1 with errno set as follows:

[EINVAL] *addr* does not begin within a memory segment.

SEE ALSO

mmap(2)

NAME

open – open a file for reading or writing, or create a new file

SYNOPSIS

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

int open(path, flags, ...)
char *path;
int  flags;
```

DESCRIPTION

open() opens the file *path* for reading and/or writing, as specified by the *flags* argument and returns a descriptor for that file. The *flags* argument may indicate the file is to be created if it does not already exist (by specifying the O_CREAT flag), in which case the file is created with a mode specified as a third argument, *mode*, of type **mode_t**. In this case, *mode* is described as in *chmod*(2) and modified by the process's umask value (see *umask*(2)).

path is the address of a NULL terminated string of ASCII characters representing a path name. *flags* 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_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_LARGEFILE
If set, *read*(2) and *write*(2) will work on regular files at offsets greater than 2^31 - 1. If not set, *read*(2) will only read up to the offset 2^31 - 1. Attempts to read further will behave as if the end-of-file has been reached. Similarly, *write*(2) will only write up to the offset 2^31 - 1. Attempts to write further will return EINVAL. See the ''ConvexOS Large Files User's Guide'' for more information.

O_CREAT If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the group ID of the directory in which the file is created, 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 file mode creation mask of the process are cleared. See *umask*(2).

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

O_EXCL If O_EXCL and O_CREAT are set, *open()* will fail if the file exists. This can be used to implement a simple exclusive access locking mechanism. If O_EXCL is set and the last component of the pathname is a symbolic link, the open will fail even if the symbolic link points to a non-existent name.

O_NOCTTY If set, and *path* indicates a terminal device, the terminal device will not become the controlling terminal for the process.

O_NONBLOCK
When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NONBLOCK 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_NONBLOCK 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_NONBLOCK is set:

The open will return without waiting for carrier. The first time the process attempts to perform I/O on the open file it will block (not currently implemented).

If O_NONBLOCK is clear:

The open will block until carrier is present.

When opening a file on a filesystem under the control of a migration daemon:

If O_NONBLOCK is set:

The process wants to receive an *ENOSPC* error instead of blocking.

If O_NONBLOCK is clear:

The process will block until sufficient blocks or fragments become available to satisfy the request.

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

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

The new descriptor is set to remain open across *execve*(2) system calls; see *close*(2) and *fcntl*(2).

There is a system enforced limit on the number of open file descriptors per process with a value returned by the *getdtablesize*(2) call.

RETURN VALUE

The value −1 is returned if an error occurs, and external variable *errno* is set to indicate the cause of the error. Otherwise a non-negative numbered file descriptor for the new open file is returned.

ERRORS

open() fails if:

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

[ENOTDIR] *path* ends in a slash, but is not a directory.

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG]
The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

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

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

[ENOENT] The *path* argument points to an empty string.

[ELOOP] Too many symbolic links were encountered in translating *path*.

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

[EACCES] The required permissions (for reading and/or writing) are denied for the file

named by *path*.

[EACCES] The file referred to by *path* does not exist, O_CREAT is specified, and the directory in which it is to be created does not permit writing.

[EINTR] The *open()* call was interrupted by a signal.

[EISDIR] The named file is a directory, and the arguments specify it is to be opened for writing.

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

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

[EMFILE] The system limit for open file descriptors per process has already been reached.

[ENFILE] The system file table is full.

[ENOSPC] The file does not exist, O_CREAT is specified, and the directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] The file does not exist, O_CREAT is specified, and there are no free inodes on the file system on which the file is being created.

[EDQUOT] The file does not exist, O_CREAT is specified, and the directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The file does not exist, O_CREAT is specified, and the user's quota of inodes on the file system on which the file is being created has been exhausted.

[EROFS] The named file does not exist, O_CREAT is specified, and the file system on which it is to be created is a read-only file system.

[EROFS] The named file resides on a read-only file system, and the file is to be opened for writing.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and the *open()* call requests write access.

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

[ENODEV] The named file does not have an associated physical device.

[EIO] The named file corresponds to a tape device which is offline, or one which was opened for write access without having a write enable ring present on the tape.

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

[EEXIST] O_EXCL and O_CREAT were both specified and the file exists.

[EOPNOTSUPP] An attempt was made to open a socket (not currently implemented).

BACKWARD COMPATIBILITY

In previous versions of the operating system, the null string for *path* was a synonym for ., the current directory.

The O_NONBLOCK flag was previous known as O_NDELAY. O_NDELAY had slightly different semantics, in that it could modify other references to the same file which did not share the same file descriptor; in that sense, it was more like *ioctl()*.

SEE ALSO

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

NAME

pattach – process attach

SYNOPSIS

#include <sys/file.h>

pattach (pid, flags)
int pid, flags;

DESCRIPTION

pattach creates a connection to a process and returns a descriptor that may be used to read, write, or control the execution of another process.

The *pid* parameter specifies the process id that should be attached. The *flags* parameter is one of the following values

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_EXCL	open for exclusive execution access

Opening a process with O_EXCL grants the caller exclusive control of the execution of the process. A process may be opened only if it has the same effective user ID as the caller (see *geteuid*(2)) and has the same effective group ID as the caller (see *getegid*(2)) or has a process group id in the caller's group access list (see *getgroups*(2)). A superuser or process in group *kmem* may open any other process.

RETURN VALUE

If successful, a descriptor referencing the process is returned. If unsuccessful, a –1 is returned, and the global variable *errno* is set to indicate the error.

ERRORS

The *pattach* call fails if:

[EACCES]	The process is already opened for exclusive access.
[EACCES]	An attempt was made for exclusive access to a system process.
[EACCES]	The calling process attempted to attach exclusively to itself.
[EPERM]	The caller is not the owner of the process.
[ESRCH]	The process does not exist.
[EMFILE]	The open file limit has been reached.
[EINVAL]	An invalid *flags* parameter was specified.

SEE ALSO

pi(4)

NAME

pgetregid – get a process' real and effective group ID

SYNOPSIS

pgetregid(pid, rgid, egid)
int pid;
short *rgid, *egid;

DESCRIPTION

The real and effective group ID's of the process whose ID is *pid* are returned in the variables pointed to by the arguments *rgid* and *egid*.

Either *rgid* or *egid* may be a null pointer, in which case the respective ID is not copied.

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

[ESRCH] The process specified does not exist.

[EFAULT] The *addr* parameter is not in a writable part of the user address space.

SEE ALSO

getgid(2), psetregid(2), setreuid(2), setuid(3),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

NAME

pipe – create an interprocess communication channel

SYNOPSIS

#include <limits.h>
#include <unistd.h>

int pipe(fildes)
int fildes[2];

DESCRIPTION

The *pipe()* system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor *fildes*[1] up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor *fildes*[0] will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent *fork(2)* calls) will pass data through the pipe with *read(2)* and *write(2)* calls.

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

read(2) on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file condition (i.e. zero, indicating nothing read).

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

The O_NONBLOCK flag is clear for both file descriptors created by *pipe()*; the *fcntl()* call may be used to set the O_NONBLOCK flag.

RETURN VALUE

The function value zero is returned if the pipe was created; –1 if an error occurred.

ERRORS

The *pipe()* call will fail if:

[EMFILE]	Too many descriptors are active.
[EFAULT]	The *fildes* buffer is in an invalid area of the process's address space.

SEE ALSO

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

NOTES

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

Pipes are really a special case of the *socketpair*(2) call and, in fact, are implemented as such in the system.

NAME

profil, lprofil – execution time profile

SYNOPSIS

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

lprofil(lbuff, bufsiz, offset, scale)
char *lbuff;
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (10 milliseconds); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to an unsigned short inside *buff*, that unsigned short is incremented. In the case of *lprofil*, the buffer indicated by *lbuff* is maintained as a buffer of unsigned longs.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0x10000 gives a 1-1 mapping of instruction half-words to locations in *buff*; 0x8000 maps each pair of instruction half-words together. 0x2 maps 64K-byte regions of instructions onto locations of buff.

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

RETURN VALUE

A 0, indicating success, is always returned.

SEE ALSO

prof(1)(optional product), gprof(1)(optional product), getitimer(2), monitor(3)

BUGS

Programs using the *lprofil* system call will frequently produce the diagnostic "Bad system call: core dumped" when run on a version of the operating system older than 7.1.

NAME

psetregid – set a process' real and effective group ID

SYNOPSIS

psetregid(pid, rgid, egid)
int pid, rgid, egid;

DESCRIPTION

The real and effective group ID's of the process whose ID is *pid* are set to the arguments. Only the superuser may use this system call.

If a value of **–1** is supplied for either the real or effective group ID, the system does not change the respective ID.

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 superuser.

[ESRCH] The process specified does not exist.

SEE ALSO

getgid(2), pgetregid(2), setreuid(2), setuid(3),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

NAME

quotactl – manipulate disk quotas

SYNOPSIS

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

quotactl(cmd, special, uid, addr)
int cmd;
char *special;
int uid;
caddr_t addr;
```

DESCRIPTION

The *quotactl* call manipulates disk quotas. The *cmd* parameter indicates a command to be applied to the user ID *uid*. *special* is a pointer to a null-terminated string containing the path name of the block special device for the file system being manipulated. The block special device must be mounted. *addr* is the address of an optional, command specific, data structure which is copied in or out of the system. The interpretation of *addr* is given with each command below.

Q_QUOTAON

Turn on quotas for a file system. *addr* is a pointer to a null terminated string containing the path name of file containing the quotas for the file system. The quota file must exist; it is normally created with the *quotacheck*(8) program. This call is restricted to the super-user.

Q_QUOTAOFF

Turn off quotas for a file system. This call is restricted to the super-user.

Q_GETQUOTA

Get disk quota limits and current usage for user *uid*. *addr* is a pointer to a struct dqblk structure (defined in *<ufs/quota.h>*). Only the super-user may get the quotas of a user other than himself. If *uid* is 0, then the timelimit fields of the returned dqblk structure are the quota time limits for the filesystem, *special*. Otherwise, the timelimit fields are the system times at which the user's hardlimits for filesystem, *special*, will be reached.

Q_SETQUOTA

Set disk quota limits and current usage for user *uid*. *addr* is a pointer to a struct dqblk structure (defined in *<ufs/quota.h>*). This call is restricted to the superuser. If *uid* is 0, then the quota time limits for filesystem, *special*, are set to the time limit fields in the dqblk structure.

Q_SETQLIM

Set disk quota limits for user *uid*. *addr* is a pointer to a struct dqblk structure (defined in *<ufs/quota.h>*). This call is restricted to the super-user. If *uid* is 0, then the quota time limits for filesystem, *special*, are set to the time limit fields in the dqblk structure.

Q_SYNC

Update the on-disk copy of quota usages for the *special* filesystem. This call is restricted to the super-user. If *special* is 0, then quotas for all filesystems with quotas enabled will effectively be *sync*ed.

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

A *quotactl* call will fail when one of the following occurs:

[EINVAL] *cmd* is invalid.

[EPERM] The call is privileged and the caller was not the super-user.

[EINVAL] The *special* is not a mounted file system or is a mounted file system without quotas enabled.

[ENOTBLK] The *special* parameter is not a block device.

[EFAULT] An invalid *addr* is supplied; the associated structure could not be copied in or out of the kernel.

[EACCES] The *addr* parameter is being interpreted as the path of a quota file which exists but is either not a regular file or is not on the file system pointed to by the *special* parameter.

[EINVAL] An internal error (such as an I/O error) has occurred in the quota system.

[ENOTDIR] A pathname lookup was being performed on either *special* or *addr* (Q_QUOTAON) and a component of the prefix was not a valid directory.

[ENOENT] A pathname lookup was being performed on either *special* or *addr* (Q_QUOTAON) and no such entry was found in the directory.

[EFAULT] An invalid *addr* is supplied; the associated structure could not be copied in or out of the kernel.

[EUSERS] The quota table is full.

SEE ALSO

quotaon(8), quotacheck(8)

BUGS

There should be some way to integrate this call with the resource limit interface provided by *setrlimit*(2) and *getrlimit*(2). Incompatible with Melbourne quotas.

NAME

read – read from a file

SYNOPSIS

cc = read(d, buf, nbytes)
int cc, d;
char *buf;
unsigned nbytes;

DESCRIPTION

read() attempts to read *nbytes* of data from the object referenced by the descriptor *d* into the buffer pointed to by *buf*.

On objects capable of seeking, the *read()* starts at a position given by the pointer associated with *d*. (See *lseek*(2)). Upon return from *read()*, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

Note: For read access to a directory, use either the *readdir*(3) or *getdirentries*(2) functions. (Directory access via *read()* is no longer available.)

The system guarantees to read the number of bytes requested only if the descriptor references a file that has that many bytes left before the end-of-file.

When attempting to read regular files larger than 2^31 - 1 bytes when the current file offset is less than 2^31 - 1 and the amount requested would require reading beyond 2^31 - 1:

If O_LARGEFILE is set, the read succeeds to the extent otherwise possible.

If O_LARGEFILE is clear, the read will behave as if the file is 2^31 - 1 bytes in size.

When attempting to read regular files when the current file offset is greater than 2^31 - 1:

If O_LARGEFILE is set, the read succeeds to the extent otherwise possible.

If O_LARGEFILE is clear, the read will return a –1 and *errno* will be set to EOFFSET.

When attempting to read from a descriptor associated with an empty pipe, socket, or FIFO:

If O_NONBLOCK is set, the read will return a –1 and *errno* will be set to EAGAIN.

If O_NONBLOCK is clear, the read will block until data is written to the pipe or the file is no longer open for writing.

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

If O_NONBLOCK is set, the read will return a –1 and *errno* will be set to EAGAIN.

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

If O_NONBLOCK is set and less data is available than requested by the *read()*, only the data that is available is returned, and the count indicates how many bytes of data were actually read.

RETURN VALUE

If successful, the number of bytes actually read is returned unless the read was asynchronous (see EXTENSIONS below), in which case the number of bytes requested is returned. If unsuccessful, a –1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

read() will fail if one or more of the following are true:

[EAGAIN]	The file was marked for non-blocking I/O (O_NONBLOCK flag set), and no data were ready to be read.
[EBADF]	*d* is not a valid file descriptor open for reading.
[EINVAL]	The requested transfer size is too big for a single request. In order to prevent possible memory deadlocks, the kernel will refuse to do transfers larger than approximately half the size of physical memory.
[EINTR]	A *read()* from a slow device was interrupted before any data arrived by

the delivery of a signal.

[EISDIR] *d* refers to a directory that is on a file system mounted using the NFS.

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

[EIO] The process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group of the process orphaned. (This would most commonly result from the user logging out with a program left running in the background, and this program attempts to read from standard input).

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

[EOFFSET] The current value of the file pointer was greater than 2^31 - 1.

[ENODMON] *Migration only:* The file referenced by *d* is under daemon control, but no daemon is present.

[EISMIGRATED] *Migration only:* A block in the file reference by *d* is migrated, and the migration daemon was unable to stage the file in from secondary storage.

CONVEX EXTENSIONS

read() may operate synchronously (default) or asynchronously, depending on the FASIO flag set with the *fcntl(2)* system call. Synchronous reads suspend the caller until the system has processed the read request and return the number of bytes actually read and placed in the buffer or 0 if end-of-file has been reached. The file position pointer is incremented by the number of bytes actually read. Asynchronous reads return before the request has been fully processed and, unless there are errors in the arguments passed, always return the number of bytes requested whether it is actually possible to complete the request. The file position pointer is incremented by the number of bytes requested. To determine what actually happened during asynchronous transfers, use *asiostat*(2).

BACKWARD COMPATIBILITY

The O_NONBLOCK mode of operation was formerly known as O_NDELAY.

The current implementation of O_NONBLOCK is slightly different than the old O_NDELAY; O_NONBLOCK will affect no processes other than the caller, whereas O_NDELAY would affect all processes with file descriptors open for that device.

The error return EWOULDBLOCK occurs in backward compatibility mode in those situations that now return EAGAIN.

SEE ALSO

asiostat(2), dup(2), fcntl(2), open(2), pipe(2), readv(2), select(2), socket(2), socketpair(2)

NAME

readlink – read value of a symbolic link

SYNOPSIS

cc = readlink(path, buf, bufsiz)
int cc;
char *path, *buf;
int bufsiz;

DESCRIPTION

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

RETURN VALUE

The call returns the count of characters placed in the buffer if it succeeds, or a –1 if an error occurs, placing the error code in the global variable *errno*.

ERRORS

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

[EINVAL]	The *path* argument contained a byte with the high-order bit set.
[ENOENT]	The pathname was too long.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[ENXIO]	The named file is not a symbolic link.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EINVAL]	The named file is not a symbolic link.
[EFAULT]	*Buf* extends outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

stat(2), symlink(2)

NAME

readv – read scattered data

SYNOPSIS

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

cc = readv(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

readv() attempts to read from the object referenced by the descriptor *d*, scattering the input data into the *iovcnt* buffers specified by the members of the *iovec* array: iov[0], iov[1], ..., iov[iovcnt – 1].

The *iovec* structure is defined as

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

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. *readv()* will always fill an area completely before proceeding to the next.

On objects capable of seeking, the *readv()* starts at a position given by the pointer associated with *d*. See *lseek*(2). Upon returning from *readv()*, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

Note: For read access to a directory, use *readdir*(3) or *getdirentries*(2). function. (Directory access using *readv()* is no longer supported.)

readv() may operate synchronously (default) or asynchronously, depending on the FASIO flag set with the *fcntl*(2) system call. Synchronous reads suspend the caller until the system has processed the read request and return the number of bytes actually read and placed in the buffer or 0 if end-of-file has been reached. The file position pointer is incremented by the number of bytes actually read. Asynchronous reads return before the request has been fully processed and unless there are errors in the arguments passed, always return the number of bytes requested whether it is actually possible to complete the request. The file position pointer is incremented by the number of bytes requested. To determine what actually happened during asynchronous transfers, use *asiostat*(2).

The system guarantees to read the number of bytes requested only if the descriptor references a file that has that many bytes left before the end-of-file.

When attempting to read from a descriptor associated with an empty pipe, socket, or FIFO:

If O_NONBLOCK is set, the read will return a –1 and *errno* will be set to EWOULDBLOCK.

If O_NONBLOCK is clear, the read will block until data is written to the pipe or the file is no longer open for writing.

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

If O_NONBLOCK is set, the read will return a –1 and *errno* will be set to EWOULDBLOCK.

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

If O_NONBLOCK is set and less data are available than are requested by the *read()* or *readv()*, only the data that are available are returned, and the count indicates how many bytes of data were actually read.

RETURN VALUE

If successful, the number of bytes actually read is returned, unless the read was asynchronous, in which case the number of bytes requested is returned. If unsuccessful, a –1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

readv() will fail if one or more of the following are true:

[EINVAL]	The requested transfer size is too big for a single request. In order to prevent possible memory deadlocks, the kernel will refuse to do transfers larger than approximately half the size of physical memory.
[EBADF]	*d* is not a valid file descriptor open for reading.
[EISDIR]	*d* refers to a directory that is on a file system mounted using the NFS.
[EFAULT]	*buf* points outside the allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EINTR]	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
[EWOULDBLOCK]	The file was marked for non-blocking I/O, and no data were ready to be read.
[EINVAL]	*iovcnt* was either less than or equal to 0 or greater than 16.
[EINVAL]	For asynchronous requests, a maximum of one *iov* region may point to a given logical page.
[EINVAL]	One of the *iov_len* values in the *iov* array was negative.
[EINVAL]	The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.
[EFAULT]	Part of *iov* points outside the process's allocated address space.
[EINVAL]	Two or more *iov* buffers share the same virtual memory page, and the mode is asynchronous. (This restriction is necessary because the pages are locked in memory during asynchronous transfers.)
[EIO]	A read from a magnetic tape encountered an end of tape marker.
[ENODMON]	*Migration only:* The file referenced by *d* is under daemon control, but no daemon is present.
[EISMIGRATED]	*Migration only:* A block in the file reference by *d* is migrated, and the migration daemon was unable to stage the file in from secondary storage.

SEE ALSO

asiostat(2), dup(2), fcntl(2), open(2), pipe(2), read(2), select(2), socket(2), socketpair(2)

NAME

reboot – reboot system or halt processor

SYNOPSIS

#include <sys/reboot.h>

reboot(howto)
int howto;

DESCRIPTION

Reboot reboots the system, and is invoked automatically in the event of unrecoverable system failures. *Howto* is a mask of options passed to the bootstrap program. The system call interface permits only RB_HALT or RB_AUTOBOOT to be passed to the reboot program; the other flags are used in scripts stored on the console storage media, or used in manual bootstrap procedures. When none of these options (e.g. RB_AUTOBOOT) is given, the system is rebooted from file "vmunix". An automatic consistency check of the disks is then normally performed.

The bits of *howto* are:

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

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

RB_SINGLE
: Normally, the reboot procedure involves an automatic disk consistency check and then multi-user operations. RB_SINGLE prevents the consistency check, rather simply booting the system with a single-user shell on the console. RB_SINGLE is interpreted by the *init*(8) program in the newly booted system. This switch is not available from the system call interface.

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

RETURN VALUES

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

ERRORS

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

SEE ALSO

halt(8), init(8)

NAME

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

SYNOPSIS

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

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

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

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

DESCRIPTION

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

The *recv* call is normally used only on a *connected* socket (see *connect*(2)), while *recvfrom* and *recvmsg* may be used to receive data on a socket whether it is in a connected state or not.

If *from* is non-zero, the source address of the message is filled in. *Fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see *socket*(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see *ioctl*(2)) in which case a *cc* of –1 is returned with the external variable errno set to EWOULDBLOCK.

The *select*(2) call may be used to determine when more data arrives.

The *flags* argument to a recv call is formed by *or*'ing one or more of the values,

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

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

```
struct msghdr {
        caddr_t msg_name;               /* optional address */
        int     msg_namelen;            /* size of address */
        struct  iovec *msg_iov;         /* scatter/gather array */
        int     msg_iovlen;             /* # elements in msg_iov */
        caddr_t msg_accrights;          /* access rights sent/received */
        int     msg_accrightslen;
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter gather locations, as described in *read*(2). A buffer to receive any access rights sent along with the message is specified in *msg_accrights*, which has length *msg_accrightslen*. Access rights are currently limited to file descriptors, which each occupy the size of an **int**.

RETURN VALUE

These calls return the number of bytes received, or –1 if an error occurred.

ERRORS

The calls fail if:

[EBADF]	The argument *s* is an invalid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking and the receive operation would block.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available for the receive.
[EFAULT]	The data was specified to be received into a non-existent or protected part of the process address space.
[ENOTCONN]	No *connect* had been done on a socket whose protocol requires connections.

SEE ALSO

fcntl(2), read(2), send(2), select(2), getsockopt(2), socket(2)

NAME

rename – change the name of a file

SYNOPSIS

```
#include <unistd.h> /* POSIX */
#include <stdio.h>  /* ANSI C */

int rename(const char *from, const char *to);
```

DESCRIPTION

rename() causes the link named *from* to be renamed as *to*. If *to* exists, then it is removed first. 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 a nonzero value 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:

[ENOTEMPTY] The destination directory is not empty or has links to it.

[EISDIR] The destination is a directory and the source is not a directory.

[ENOTDIR] The destination is not a directory and the source is a directory.

[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.

[ENOENT] *from* or *to* points to an empty string.

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

[EPERM] The file named by *to* is not owned by the caller, and the directory containing *to* has the sticky bit set.

[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] *from* or *to* points outside the process' allocated address space.

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

[EEXIST] *from* or *to* specified the root directory.

[ESTATECHANGED] *MIGRATION ONLY:* The target file changed state during a call to a migration daemon. There are a number of cases to consider. If *to* existed before the call to the daemon, then it no longer exists. If the target file *to* did not exist prior to the call to the daemon, it now exists. If *from* was removed, then the operation may still have succeeded; *from* is now renamed to *to*, but the operating system was unable to remove the old *from* as part of the rename operation.

BACKWARD COMPATIBILITY

Previous versions of the operating system interpreted the empty path name as a synonym for the current directory.

Previous versions of the operating system had slightly different values of errno:

[EISDIR] The destination is a file, but the source is a directory.

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

NOTES

In some cases, only the *d_fileno* of the directory entry for the file being renamed is zeroed; the name itself may persist in the directory until the directory entry is reused or until the directory itself is removed.

The system can deadlock if a loop in the file system graph is present. This loop takes the form of an entry in directory **a** (e.g., **a/foo**) being a hard link to directory **b**, and an entry in directory **b** (e.g., **b/foo**) being a hard link to directory **a**. When such a loop exists and two separate processes attempt to perform **rename a/foo b/bar** and **rename b/bar a/foo**, respectively, the system may deadlock attempting to lock both directories for modification. Hard links to directories should be replaced by symbolic links by the system administrator. (Note that only root may create a hard link to a directory in the first place.)

SEE ALSO

open(2), link(2)

NAME

rmdir – remove a directory file

SYNOPSIS

```
#include <unistd.h>

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.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	The pathname was too long.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist, or *path* points to an empty string.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EROFS]	The directory entry to be removed resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

BACKWARD COMPATIBILITY

Previous versions of the operating system interpreted the empty path name as a synonym for the current directory.

SEE ALSO

mkdir(2), unlink(2)

NAME

select – synchronous I/O multiplexing

SYNOPSIS

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

nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION

select examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; i.e. the descriptors from 0 through *nfds*-1 in the descriptor sets are examined. On return, *select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in *nfound*.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: *FD_ZERO(&fdset)* initializes a descriptor set *fdset* to the null set. *FD_SET(fd, &fdset)* includes a particular descriptor *fd* in *fdset*. *FD_CLR(fd, &fdset)* removes *fd* from *fdset*. *FD_ISSET(fd, &fdset)* is nonzero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select blocks indefinitely. To effect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as zero pointers if no descriptors are of interest.

RETURN VALUE

select returns the number of ready descriptors that are contained in the descriptor sets, or –1 if an error occurred. If the time limit expires then *select* returns 0. If *select* returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from *select* indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EFAULT]	One of the descriptor set addresses is invalid.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.
[EINVAL]	The first argument, *nfds*, is negative.
[EINVAL]	*nfds* is zero, and at least one of *readfds*, *writefds*, or *exceptfds* is not zero.

SEE ALSO

accept(2), connect(2), read(2), write(2), recv(2), send(2), getdtablesize(2)

BUGS

Although the provision of *getdtablesize*(2) was intended to allow user programs to be written independent of the kernel limit on the number of open files, the dimension of a sufficiently large bit field for select remains a problem. The default size FD_SETSIZE (256) is equal to the current kernel limit to the number of open files. However, in order to accommodate programs which might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of <sys/types.h>.

select should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the *select* call.

NAME

send, sendto, sendmsg – send a message to a socket

SYNOPSIS

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

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

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

cc = sendmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

DESCRIPTION

Send, *sendto*, and *sendmsg* are used to transmit a message to another socket. *Send* may be used only when the socket is in a *connected* state, while *sendto* and *sendmsg* may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of –1 indicate some locally detected errors.

If no messages space is available at the socket to hold the message to be transmitted, then *send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The *select*(2) call may be used to determine when it is possible to send more data.

The *flags* parameter may include one or more of the following:

```
#define MSG_OOB          0x1    /* process out-of-band data */
#define MSG_DONTROUTE    0x4    /* bypass routing, use direct interface */
```

The flag MSG_OOB is used to send "out-of-band" data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support "out-of-band" data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

See *recv*(2) for a description of the *msghdr* structure.

RETURN VALUE

The call returns the number of characters sent, or –1 if an error occurred.

ERRORS

[EACCES]	An attempt was made to pass file descriptors in the *sendmsg* access rights when the system tunable *sendmsg_access_rights* has been configured to disallow this facility.
[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the requested operation would block.
[ENOBUFS]	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFS] The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

[ENOTCONN] No connection had been established when a *send* operation was tried.

SEE ALSO

fcntl(2), recv(2), select(2), getsockopt(2), socket(2), write(2)

NAME

setaid – set a process' activity ID

SYNOPSIS

setaid(pid, aid)
int pid, aid;

DESCRIPTION

The activity ID of the process whose ID is *pid* is set to *aid*.

An activity ID may be any 32-bit value except **–1**. Attempts to set an activity ID to **–1** are quietly ignored.

Only the superuser may use this system call.

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 superuser.
[ESRCH]	The process specified does not exist.

SEE ALSO

getaid(2),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

NAME

setgroups – set group access list

SYNOPSIS

#include <sys/param.h>

setgroups(ngroups, gidset)
int ngroups, *gidset;

DESCRIPTION

Setgroups sets the group access list of the current user process according to the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than NGROUPS, as defined in *<sys/param.h>*.

Only the superuser may set new groups.

RETURN VALUE

A **0** value is returned on success, **–1** on error, with an error code stored in *errno*.

ERRORS

The *setgroups* call will fail if:

[EINVAL]	*Ngroups* is too large.
[EPERM]	The caller is not the superuser.
[EFAULT]	The address specified for *gidset* is outside the process address space.

SEE ALSO

getgroups(2), initgroups(3X)

NAME

setpgid – set process group

SYNOPSIS

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

int setpgid(pid, pgrp)
pid_t pid,pgrp;
```

DESCRIPTION

setpgid causes process *pid* to join an existing process group or create a new process group within the session of the calling process. If *pid* is zero, then the process ID of the calling process is used. If *pgrp* is zero, then the process ID of the calling process is used.

The process group ID of a session leader may not be changed.

RETURN VALUE

setpgid returns zero when the operation is successful. If the request fails, –1 is returned and the global variable *errno* indicates the reason.

ERRORS

setpgid fails and the process group is not altered if one of the following occur:

[EACCES]	*pid* matches the process ID of a child process of the calling process and the child process has successfully executed one of the *exec* functions.
[EINVAL]	The value of *pgrp* is less than zero.
[EPERM]	The process indicated by *pid* is a session leader.
[EPERM]	The value of *pid* matches a child process of the calling process, but the child is not in the same session as the calling process.
[EPERM]	The value of *pgrp* does not match the process id of the process indicated by *pid* and there is no process with a process group ID that matches the value of *pgrp* in the same session as the calling process.
[ESRCH]	The value of *pid* does not match the process ID of the calling process or of a child process of the calling process.

BACKWARD COMPATIBILITY

This function was previously known as *setpgrp()*. The functionality of *setpgrp()* was essentially that of *setpgid()* without the restrictions on session membership.

SEE ALSO

getpgrp(2)

NAME

setpgrp – set process group

SYNOPSIS

setpgrp(pid, pgrp)
int pid, pgrp;

DESCRIPTION

setpgrp sets the process group of the specified process *pid* to the specified *pgrp*. If *pid* is zero, then the call applies to the current process.

RETURN VALUE

setpgrp returns 0 when the operation was successful. If the request failed, –1 is returned and the global variable *errno* indicates the reason.

ERRORS

setpgrp will fail and the process group will not be altered if one of the following occur:

[EACCES] *pid* matches the process ID of a child process of the calling process and the child process has successfully executed one of the *exec* functions.

[EINVAL] The value of *pgrp* is less than zero.

[EPERM] The process indicated by *pid* is a session leader.

[EPERM] The value of *pid* matches a child process of the calling process, but the child is not in the same session as the calling process.

[EPERM] The value of *pgrp* does not match the process id of the process indicated by *pid* and there is no process with a process group ID that matches the value of *pgrp* in the same session as the calling process.

[ESRCH] The value of *pid* does not match the process ID of the calling process or of a child process of the calling process.

BACKWARD COMPATIBILITY

This function is obsoleted by *setpgid*(2).

Compliance with POSIX P1003.1 introduced restrictions on *setpgrp*(2) which did not previously exist.

SEE ALSO

setpgid(2), getpgrp(2)

NAME

setpid – set process id

SYNOPSIS

#include <sys/types.h>

int setpid(pid_t pid)

DESCRIPTION

setpid sets the process id of the calling process to *pid* if *pid* is currently not in use by another process in the system. A process may set its *pid* successfully only once. A process may not change its process id while it has living children.

RETURN VALUE

setpid returns zero if it is successful in setting the *pid*. Otherwise it returns -1 and the global variable *errno* contains an error code.

ERRORS

[EACCES]	The process attempted to set its *pid* while it had living children.
[EAGAIN]	The *pid* is currently in use by another process in the system.
[EINVAL]	The *pid* argument is not in the range of zero through 30000.
[EPERM]	The process attempted to set its *pid* more than once.

SEE ALSO

getpid(2), getppid(2)

NOTES

restart(1) calls *setpid*(2) to restore a process's original process id. A call to *setpid*(2) by a process after it has been restarted will fail.

NAME

setregid – set real and effective group ID

SYNOPSIS

#include <sys/types.h>

int setregid(gid_t rgid, gid_t egid)

DESCRIPTION

The real and effective group ID's of the current process are set to the arguments. Only the superuser may change the real group ID of a process. Unprivileged users may change the effective group ID to the real group ID or the saved set-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 superuser and a change other than changing the effective group-ID to the real group-ID was specified.

[EINVAL] The caller is a POSIX-mode process, and both real and effective gid are –1.

NOTES

A POSIX mode process is allowed to change the effective gid freely between the values of the saved set-group-ID and the real group id.

For a POSIX mode process, the saved set-user-ID is set to the real gid if the caller has effective uid zero.

For a non-POSIX process, the saved set-group-ID is *always* set to the specified real group id; this behavior renders ineffective the POSIX-specified ability to toggle between the real and the saved set-group-ID values.

SEE ALSO

getgid(2), setreuid(2), setuid(3)

NAME

setreuid – set real and effective user ID's

SYNOPSIS

#include <sys/types.h>

int setreuid(uid_t ruid, uid_t 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 or the saved set-user-ID.

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.

[EINVAL] The caller is a POSIX-mode process, and both real and effective uid are –1.

NOTES

A POSIX mode process is allowed to change the effective uid freely between the values of the saved set-user-ID and the real user id.

For a POSIX mode process, the saved set-user-ID is set to the real uid if the caller had effective uid zero prior to the call.

For a non-POSIX process, the saved set-user-ID is *always* set to the specified real user id; this behavior renders ineffective the POSIX-specified ability to toggle between the real and the saved set-user-ID values.

SEE ALSO

getuid(2), setregid(2), setuid(3)

NAME

setsid – create session and set process group ID

SYNOPSIS

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

pid_t setsid()
```

DESCRIPTION

If the calling process is not a process group leader, the *setsid()* function creates a new session. The calling process is the leader of this new session, is the process group leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process. The calling process is the only process in the new process group and the only process in the new session.

RETURN VALUE

Upon success, *setsid()* returns the process group ID of the calling process, which is in fact its process ID. Upon failure, –1 is returned and *errno* indicates the cause for failure.

ERRORS

setsid() will fail if any one of the following occur:

[EPERM]	The calling process is already a process group leader.
[EPERM]	The process group ID of a process other than the calling process matches the process ID of the calling process.

SEE ALSO

intro(2), getpgrp(2), setpgid(2)

NAME

shutdown – shut down part of a full-duplex connection

SYNOPSIS

shutdown(s, how)
int s, how;

DESCRIPTION

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

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

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

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

[ENOTCONN] The specified socket is not connected.

SEE ALSO

connect(2), socket(2)

NAME

sigaction – examine and change signal action

SYNOPSIS

```
#include <signal.h>

struct sigaction {
        void    (*sa_handler)();
        int     sa_mask;
        int     sa_flags;
};

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

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered or specify that a signal is to be *blocked* or *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs.

All signals have the same *priority*. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a *sigprocmask*(3) call or when a signal is delivered to the process.

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

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

sigaction() assigns a handler for a specific signal. If *vec* is nonzero, it specifies a handler routine and mask to be used when delivering the specified signal. If *ovec* is nonzero, the previous handling information for the signal is returned to the user.

The mask specified in *vec* is not allowed to block SIGKILL or SIGSTOP. The system enforces this restriction silently.

The following is a list of all signals with names in <signal.h> . In this list, those names beginning with an underscore are not defined in POSIX P1003.1, and are provided as CONVEX extensions.

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
_SIGTRAP	5*	trace trap
_SIGIOT	6*	IOT instruction
SIGABRT	6*	*abort*(3)
_SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
_SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation

_SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe or other socket with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
_SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop (cannot be blocked)
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
_SIGIO	23•	i/o is possible on a descriptor (see *fcntl*(2))
_SIGXCPU	24	cpu time limit exceeded (see *setrlimit*(2))
_SIGXFSZ	25	file size limit exceeded (see *setrlimit*(2))
_SIGVTALRM		26virtual time alarm (see *setitimer*(2))
_SIGPROF	27	profiling timer alarm (see *setitimer*(2))
_SIGWINCH	28•	window size change
_SIGLOST	29*	resource lost (see *lockd*(8C))
SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

Once a signal handler is installed, it remains installed until another *sigaction()* call is made, or an *execve*(2) is performed. The default action for a signal may be reinstated by setting *sa_handler* to SIG_DFL; this default is termination except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop. If the process is terminated, a "core image" is made in the current working directory of the receiving process if the signal is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

The effective group ID and the real group ID of the receiving process are equal.

An ordinary file named *core* exists and is writable or can be created. If the file must be created, it has the following properties:

Mode of 0666 modified by the file creation mask (see *umask*(2))

File owner ID that is the same as the effective user ID of the receiving process

File group ID that is the same as the file group ID of the current directory

If *sa_handler* is SIG_IGN, the signal is subsequently ignored, and pending instances of the signal are discarded.

Note: the signals SIGKILL and SIGSTOP cannot be ignored.

The *sigaction*(2) call blocks if the signal in *sig* is currently being serviced by another thread in the process. The call is resumed after the thread servicing the signal returns or enables the signal via the *sigsetmask*(2) system call.

After a *fork*(2) or *vfork*(2), the child inherits all signals, the signal mask, and the signal stack, and the restart/interrupt flags.

The *execve*(2) call resets all caught signals to default action and resets all signals to be caught on the user stack. Ignored signals remain ignored, the signal mask remains the same, and the signal stack state is reset.

The *SA_NOCLDSTOP* bit of the *sa_flags* word of the *sigaction* structure controls generation of SIGCHLD in some circumstances. If *sig* is SIGCHLD and SA_NOCLDSTOP is not set in *sa_flags*, a SIGCHLD signal is generated for the calling process whenever any of its child processes stop; if SA_NOCLDSTOP is set, SIGCHLD is not generated.

CONVEX EXTENSIONS

If a caught signal occurs during certain system calls, the call is normally interrupted; it returns –1 and sets *errno* to EINTR. The call can be automatically restarted (the default BSD UNIX behavior) by setting the _SA_RESTARTSYS bit in *sa_flags*. The affected system calls are *read*(2) or *write*(2) on a slow device (such as a terminal or pipe or other socket, but not a file) and during

a *wait*(2). (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special stack.

As an extension, if the _SA_ONSTACK bit is set in *sa_flags*, the system delivers the signal to the process on a *signal stack*, specified with *sigstack*(2).

An alternate mode of signals can be used by setting the _SA_PARALLEL bit in *sa_flags*. When this bit is set, signals that are masked by the delivery of a signal is not be reflected in the masks returned by *sigblock*(2) and *sigsetmask*(2) system calls. It also changes the action taken upon return of signal handler to unblock signals blocked when the handler was invoked rather then resetting the mask to what it was prior to the signal delivery.

The handler routine can be declared:

```
void handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here, *sig* is the signal number into which the hardware faults and traps are mapped as defined below. *code* is a parameter that further defines the type of hardware exception that occurred. *scp* is a pointer to the *sigcontext* structure (defined in <signal.h>), used to restore the context from before the signal.

The following list defines the mapping of hardware traps to signals and codes. All of these symbols are defined in <signal.h>:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Reserved Operand trap	SIGFPE	FPE_RESOP_TRAP
Segmentation Violations:		
Read access violation	SIGSEGV	SEG_READ_TRAP
Write access violation	SIGSEGV	SEG_WRITE_TRAP
Execute access violation	SIGSEGV	SEG_EXEC_TRAP
Invalid segment	SIGSEGV	SEG_INVSDR_TRAP
Invalid page table page	SIGSEGV	SEG_INVPTP_TRAP
Invalid memory reference	SIGSEGV	SEG_INVDATA_TRAP
I/O access violation	SIGSEGV	SEG_IOACC_TRAP
Ring Violations:		
Inward address reference	SIGBUS	BUS_INWADDR_TRAP
Outward ring call	SIGBUS	BUS_OUTCALL_TRAP
Inward ring return	SIGBUS	BUS_INWRTN_TRAP
Invalid syscall gate	SIGBUS	BUS_INVGATE_TRAP
Invalid return frame length	SIGBUS	BUS_INVFRL_TRAP
Illegal instruction:		
Error exit instruction	SIGILL	ILL_ERRXIT_TRAP
Privileged instruction	SIGILL	ILL_PRIVIN_TRAP
Undefined op code	SIGILL	ILL_UNDFOP_TRAP
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	

RETURN VALUE

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

ERRORS

sigaction() fails and no new signal handler is installed if one of the following occurs:

[EFAULT] Either *vec* or *ovec* points to memory that is not a valid part of the process

address space.

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

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

[EINTR] The caller is multi-threaded, and while waiting for another thread to complete processing signal *sig* , a distinct, non-blocked signal arrives.

BACKWARD COMPATIBILITY

The *sigaction()* function replaces the *sigvec()* function.

The *struct sigaction* replaces the *struct sigvec.*

Signal handling functions formerly returned type *int*; they now return type *void.* This may be the source of many compile-time warnings.

For non-POSIX compliant calling processes, the _SA_RESTARTSYS bit of *sa_flags* is ignored. Instead, system calls are restarted by default, as they traditionally were under BSD UNIX. In this case, the _SA_INTERRUPT bit may be set to cause system calls *not* to be restarted.

Just as _SA_RESTARTSYS is ignored for non-POSIX callers, the _SA_INTERRUPT bit is ignored for POSIX callers.

In previous releases of the operating system, it was illegal to set the disposition of SIGCONT to SIG_IGN; this is now permitted but has no effect.

SEE ALSO

kill(1), pattach(2), kill(2), sigblock(2), sigsetmask(2), sigpause(2) sigstack(2), setjmp(3), tty(4)

NAME

sigblock, sigunblock – block or unblock signals

SYNOPSIS

```
#include <signal.h>

int sigblock(mask)
int mask;

int sigunblock(mask)
int mask;
```

DESCRIPTION

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

sigunblock removes the signals specified in *mask* from the set of signals currently being blocked from delivery. If bit *i* of mask is set, then bit *i* of the process's signal mask will be cleared.

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

The call will block if any other thread in the process is currently servicing a signal contained in mask. Execution of the call will continue when the thread in the signal handler returns or enables the signal via the *sigsetmask*(2) or *sigunblock*(2) system calls. In this case, the call is interruptible; see **RETURN VALUE**, below.

RETURN VALUE

On success, the previous set of masked signals is returned. The call is always successful for single threaded callers.

For multi-threaded callers, the call may return –1 and *errno* be set to **EINTR**, meaning that the calling thread received a signal which was not blocked while awaiting other threads to finish processing signals in the mask. Note that this return value of –1 is not ambiguous, since the bits representing SIGSTOP and SIGKILL can never be set in the mask returned by a successful call.

SEE ALSO

kill(2), sigvec(2), sigsetmask(2), sigpause(2), sigprocmask(3)

NOTES

The preferred mechanism for signal blocking is *sigprocmask()*.

The *sigunblock()* call is a CONVEX extension to allow reliable processing of parallel signals.

NAME

sigpause – atomically release blocked signals and wait for interrupt

SYNOPSIS

sigpause(sigmask)
int sigmask;

DESCRIPTION

sigpause assigns *sigmask* to the set of masked signals and then waits for a signal to arrive; on return, the set of masked signals is restored for processes that do not use the SV_PARALLEL option of the *sigvec* system call. *sigmask* is usually 0 to indicate that no signals are now to be blocked. *sigpause* always terminates by being interrupted and returns EINTR. The call will block if any other thread in the process is currently running a signal handler for any signal included in *sigmask*. Execution of the call will continue whenever the thread in the signal handler returns or makes a *sigsetmask* call to re-enable the signal.

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

SEE ALSO

sigblock(2), sigvec(2), sigsetmask(2), kill(2), sigsuspend(3)

NOTES

The preferred mechanism is now *sigsuspend(3)*.

NAME

sigpending – examine pending signals

SYNOPSIS

#include <signal.h>

int sigpending(set)
sigset_t *set;

DESCRIPTION

sigpending stores the set of signals that are blocked from delivery and pending into the location pointed to by *set*.

RETURN VALUE

sigpending always succeeds and returns a value of zero.

SEE ALSO

kill(1), sigaction(2), sigprocmask(3)

NAME

sigsetmask – set current signal mask

SYNOPSIS

sigsetmask(mask);
int mask;

DESCRIPTION

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

The system quietly disallows SIGKILL and SIGSTOP to be blocked.

The call will block if any other thread in the process is currently servicing a signal contained in mask. Execution of the call will continue when the thread in the signal handler returns or enables the signal via the *sigsetmask*(2) or *sigunblock*(2) system calls. In this case, the call is interruptible; see **RETURN VALUE**, below.

RETURN VALUE

On success, the previous set of masked signals is returned. The call is always successful for single threaded callers.

For multi-threaded callers, the call may return –1 and *errno* be set to **EINTR**, meaning that the calling thread received a signal which was not blocked while awaiting other threads to finish processing signals in the mask. Note that this return value of –1 is not ambiguous, since the bits representing SIGSTOP and SIGKILL can never be set in the mask returned by a successful call.

SEE ALSO

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

NAME

sigstack – set and/or get signal stack context

SYNOPSIS

```
#include <signal.h>

struct sigstack {
        caddr_t         ss_sp;
        int     ss_onstack;
};

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

DESCRIPTION

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

NOTES

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

RETURN VALUE

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

ERRORS

Sigstack will fail and the signal stack context will remain unchanged if one of the following occurs.

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

SEE ALSO

sigvec(2), setjmp(3)

NAME

sigvec – software signal facilities

SYNOPSIS

```
#include <signal.h>

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

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

DESCRIPTION

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

All signals have the same *priority*. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a *sigblock*(2) or *sigsetmask*(2) call, or when a signal is delivered to the process.

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

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

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

The mask specified in *vec* is not allowed to block SIGKILL, SIGSTOP, or SIGCONT. The system enforces this restriction silently.

The following is a list of all signals with names as in the include file *<signal.h>*:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation

SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe or other socket with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop (cannot be blocked)
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
SIGIO	23•	i/o is possible on a descriptor (see *fcntl*(2))
SIGXCPU	24	cpu time limit exceeded (see *setrlimit*(2))
SIGXFSZ	25	file size limit exceeded (see *setrlimit*(2))
SIGVTALRM	26	virtual time alarm (see *setitimer*(2))
SIGPROF	27	profiling timer alarm (see *setitimer*(2))
SIGWINCH	28•	window size change
SIGLOST	29*	resource lost (see *lockd*(8C))
SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

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

Once a signal handler is installed, it remains installed until another *sigvec* call is made, or an *execve*(2) is performed. The default action for a signal may be reinstated by setting *sv_handler* to SIG_DFL; this default is termination except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop. If the process is terminated, a "core image" will be made in the current working directory of the receiving process if the signal is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

The effective group ID and the real group ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask (see *umask*(2))

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the file group ID of the current directory

If *sv_handler* is SIG_IGN the signal is subsequently ignored, and pending instances of the signal are discarded.

Note: the signals SIGKILL, SIGSTOP, and SIGCONT cannot be ignored.

If a caught signal occurs during certain system calls, the call is normally restarted. The call can be forced to terminate prematurely with an EINTR error return by setting the SV_INTERRUPT bit in *sv_flags*. The affected system calls are *read*(2) or *write*(2) on a slow device (such as a terminal or pipe or other socket, but not a file) and during a *wait*(2).

An alternate mode of signals can be used by setting the SV_PARALLEL bit in sv_flags. When this bit is set, signals that are masked by the delivery of a signal will not be reflected in the masks returned by *sigblock*(2) and *sigsetmask*(2) system calls. It also changes the action taken upon return of signal handler to unblock signals blocked when the handler was invoked rather then resetting the mask to what it was prior to the signal delivery.

The *sigvec*(2) call will block if the signal in *sig* is currently being serviced by another thread in the process. The call will be resumed after the thread servicing the signal returns or enables the signal via the *sigsetmask*(2) system call.

After a *fork*(2) or *vfork*(2) the child inherits all signals, the signal mask, and the signal stack, and the restart/interrupt flags.

The *execve*(2) call resets all caught signals to default action and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; the signal stack state is reset.

NOTES

The handler routine can be declared:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number, into which the hardware faults and traps are mapped as defined below. *code* is a parameter which further defines the type of hardware exception which occurred. *scp* is a pointer to the *sigcontext* structure (defined in <*signal.h*>), used to restore the context from before the signal.

The following defines the mapping of hardware traps to signals and codes. All of these symbols are defined in <*signal.h*>:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Reserved Operand trap	SIGFPE	FPE_RESOP_TRAP
Segmentation Violations:		
Read access violation	SIGSEGV	SEG_READ_TRAP
Write access violation	SIGSEGV	SEG_WRITE_TRAP
Execute access violation	SIGSEGV	SEG_EXEC_TRAP
Invalid segment	SIGSEGV	SEG_INVSDR_TRAP
Invalid page table page	SIGSEGV	SEG_INVPTP_TRAP
Invalid memory reference	SIGSEGV	SEG_INVDATA_TRAP
I/O access violation	SIGSEGV	SEG_IOACC_TRAP
Ring Violations:		
Inward address reference	SIGBUS	BUS_INWADDR_TRAP
Outward ring call	SIGBUS	BUS_OUTCALL_TRAP
Inward ring return	SIGBUS	BUS_INWRTN_TRAP
Invalid syscall gate	SIGBUS	BUS_INVGATE_TRAP
Invalid return frame length	SIGBUS	BUS_INVFRL_TRAP
Illegal instruction:		
Error exit instruction	SIGILL	ILL_ERRXIT_TRAP
Privileged instruction	SIGILL	ILL_PRIVIN_TRAP
Undefined op code	SIGILL	ILL_UNDFOP_TRAP
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	

RETURN VALUE

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

ERRORS

sigvec will fail and no new signal handler will be installed if one of the following occurs:

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

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

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

[EINVAL] An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

SEE ALSO

kill(1), pattach(2), kill(2), sigblock(2), sigsetmask(2), sigpause(2) sigstack(2), setjmp(3), tty(4)

NAME

socket – create an endpoint for communication

SYNOPSIS

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

s = socket(domain, type, protocol)
int s, domain, type, protocol;

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file *<sys/socket.h>*. The currently understood formats are:

PF_UNIX	or AF_UNIX	(UNIX internal protocols)
PF_INET	or AF_INET	(ARPA Internet protocols)
PF_NS	or AF_NS	(Xerox Network Systems protocols)
PF_IMPLINK	or AF_IMPLINK	(IMP "host at IMP" link layer)

(UNIX is a registered trademark of UNIX System Laboratories, Inc.)

The socket has the indicated *type,* which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently implemented only for PF_NS. SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW, which is available only to the super-user, and SOCK_RDM, which is planned, but not yet implemented, are not described here.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see *protocols*(5).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect*(2) call. Once connected, data may be transferred using *read*(2) and *write*(2) calls or some variant of the *send*(2) and *recv*(2) calls. When a session has been completed a *close*(2) may be performed. Out-of-band data may also be transmitted as described in *send*(2) and received as described in *recv*(2).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with –1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that *read*(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2) calls. Datagrams are generally received with *recvfrom*(2), which returns the next datagram with its return address.

An *fcntl*(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events via SIGIO.

The operation of sockets is controlled by socket level *options*. These options are defined in the file *<sys/socket.h>*. *Setsockopt*(2) and *getsockopt*(2) are used to set and get options, respectively.

RETURN VALUE

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

ERRORS

The *socket* call fails if:

[EPROTONOSUPPORT] The protocol type or the specified protocol is not supported within this domain.

[EAFNOSUPPORT] The address family is not supported within this domain. Ordinarily equivalent to EPROTONOSUPPORT.

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

[ENFILE] The system file table is full.

[EACCESS] Permission to create a socket of the specified type and/or protocol is denied.

[ENOBUFS] Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), socketpair(2), write(2)

NOTES

The PF_NS and PF_IMPLINK protocol families are currently unsupported.

NAME

socketpair – create a pair of connected sockets

SYNOPSIS

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

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

DESCRIPTION

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

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

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

SEE ALSO

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

BUGS

This call is currently implemented only for the UNIX domain. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

NAME

stat, fstat, stat64, fstat64 – get file status

SYNOPSIS

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

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

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

stat64(path, buf)
char *path;
stat64_t *buf;

fstat64(fd, buf)
int fd;
stat64_t *buf;
```

DESCRIPTION

stat and *stat64* obtain information about the file named by *path*. 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.

fstat and *fstat64* obtain the same information about an open file referenced by the argument descriptor, such as would be obtained by an *open* call.

buf is a pointer to a *stat* structure into which information is placed concerning the file. The contents of the structure pointed to by *buf* include the following members:

```
struct stat {
        dev_t     st_dev;     /* device inode resides on */
        ino_t     st_ino;     /* this inode's number */
        mode_t    st_mode;    /* protection */
        nlink_t   st_nlink;   /* number of hard links to the file */
        uid_t     st_uid;     /* user-id of owner */
        gid_t     st_gid;     /* group-id of owner */
        off_t     st_size;    /* total size of file */
        time_t    st_atime;   /* file last access time */
        time_t    st_mtime;   /* file last modify time */
        time_t    st_ctime;   /* file last status change time */
        /* The following are CONVEX extensions */
        dev_t     st_rdev;    /* the device type, for inode that is device */
        long      st_blksize; /* optimal blocksize for file system i/o ops */
        long      st_blocks;  /* actual number of blocks allocated */
};
```

st_atime Time when file data was last read. Changed by the following system calls: *mknod*(2), *utimes*(2), *read*(2), and *readlink*(2). When a directory is searched st_atime for the directory is set.

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

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

mknod(2), *rename*(2), *unlink*(2), *utimes*(2), *write*(2), *truncate*(2).

st_mode The attributes of st_mode may be queried with the following macros:

S_ISDIR(st.st_mode)	True if a directory
S_ISCHR(st.st_mode)	True if a character special device
S_ISBLK(st.st_mode)	True if a block special device
S_ISREG(st.st_mode)	True if a regular file
S_ISFIFO(st.st_mode)	True if a pipe or fifo special file

The permissions associated with st_mode may be extracted with the S_IRWXU, S_IRWXG, and S_IRWXO masks. The set-user-ID bit is S_ISUID and the set-group-ID bit is S_ISGID. Refer to *chmod*(2) for more details.

st_blksize The block size of the file system. See *df*(1).

st_blocks Number of 512 byte disk sectors allocated to the file. Note that fragments are always an integral multiple of 512 bytes, and that storage for files is allocated in multiples of the fragment size (or block size, for files sizes greater than 12 times the block size). This means more sectors may be allocated to a file than would appear necessary as determined by dividing the file size by 512.

stat64 and *fstat64* take a *cvxstat* structure in place of the *stat* structure. For convience, the *cvxstat* structure has been typedef'ed to *stat64_t*.

The *stat* structure and the *stat64_t* structure have identical members, with the exception that the *st_size* member is an *off64_t* instead of an *off_t* thus providing size information for files larger than two gigabytes.

stat64 and *fstat64* are CONVEX extensions to POSIX 1003.1.

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

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

[ENOENT] *path* points to an empty string or *path* does not exist.

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

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG] The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters. The file referred to by *path* does not exist.

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

[ELOOP] Too many symbolic links were encountered in translating *path*.

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

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

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

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

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

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

BACKWARD COMPATIBILITY

In previous versions of the operating system, the empty pathname string was a synonym for the current directory.

The file status information was previously expressed as follows:

```
#define  S_IFMT    0170000   /* type of file */
#define  S_IFIFO   0010000   /* fifo special */
#define  S_IFCHR   0020000   /* character special */
#define  S_IFDIR   0040000   /* directory */
#define  S_IFBLK   0060000   /* block special */
#define  S_IFREG   0100000   /* regular */
#define  S_IFLNK   0120000   /* symbolic link */
#define  S_IFSOCK  0140000   /* socket */
#define  S_ISUID   0004000   /* set user id on execution */
#define  S_ISGID   0002000   /* set group id on execution */
#define  S_ISVTX   0001000   /* see sticky(8) */
#define  S_IREAD   0000400   /* read permission, owner */
#define  S_IWRITE  0000200   /* write permission, owner */
#define  S_IEXEC   0000100   /* execute/search permission, owner */
```

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

SEE ALSO

chmod(2), chown(2), lstat(2), readlink(2), utimes(2)

NOTES

Applying *fstat* to a socket (and thus to a pipe) returns a zero'd buffer, except for the st_blksize field, and a unique device and inode number.

NAME

statfs, fstatfs – get file system statistics

SYNOPSIS

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

statfs(path, buf)
char *path;
struct statfs *buf;

fstatfs(fd, buf)
int fd;
struct statfs *buf;
```

DESCRIPTION

statfs returns information about a mounted file system. *path* is the path name of any file within the mounted filesystem. *buf* is a pointer to a *statfs* structure defined as follows:

```
typedef struct {
        long    val[2];
} fsid_t;

struct statfs {
        long    f_type;       /* filesystem type, MOUNT_UFS or MOUNT_NFS */
        long    f_bsize;      /* fundamental file system block size */
        long    f_blocks;     /* total blocks in file system */
        long    f_bfree;      /* free blocks */
        long    f_bavail;     /* free blocks available to non-superuser */
        long    f_files;      /* total file nodes in file system */
        long    f_ffree;      /* free file nodes in fs */
        fsid_t  f_fsid;       /* file system ID */
        long    f_blkpre;     /* Convex local mig: prestage blk cnt */
        long    f_blkhi;      /* Convex local mig: hiwater blk cnt */
        long    f_blklo;      /* Convex local mig: lowater blk cnt */
        long    f_spare[4];   /* spare for later */
};
```

Fields that are undefined for a particular file system are set to –1. (Note that *f_bsize* is the frag size for 4.2 type file systems. The block size can be obtained from *fstat*.) *fstatfs* returns the same information about an open file referenced by descriptor *fd*. Migration information is only available for locally mounted filesystems (*f_type* is MOUNT_UFS).

RETURN VALUE

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

ERRORS

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

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

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG] The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

[ENOENT] The file referred to by *path* does not exist.

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

[ELOOP] Too many symbolic links were encountered in translating *path*.

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

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

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

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

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

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

NAME

swapon – add a swap device for interleaved paging/swapping

SYNOPSIS

swapon(special)
char *special;

DESCRIPTION

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

RETURN VALUE

If successful, swapon returns 0. If unsuccessful, a –1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

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

[EPERM] The *swapon* system call is only valid when issued by a superuser. Since the list of valid swap devices often includes partitions which are being used for normal file systems, and swapping on such a file system would destroy the data on that file system, ordinary users are not permitted to use *swapon.*

[ENOTBLK] The specified swap device is not a block device.

[EBUSY] The specified swap device is all ready in use.

[EINVAL] The specified swap device is not included in the list of valid swap devices. The default list is set by the *sysgen* utility and includes most "b" partitions but can be modified at boot time.

SEE ALSO

swapon(8) sysgen(8)
"Boot-Time Parameters" chapter of the CONVEX System Manager's Guide.

CAVEATS

Superusers should be careful not to swap on a partition unless they are willing to lose all existing data on that partition, and should *never* swap on a mounted file system.

BUGS

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

NAME

symlink – make symbolic link to a file

SYNOPSIS

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

DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name may be an arbitrary path name; the files need not be on the same file system.

RETURN VALUE

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in *errno* and a –1 value is returned.

ERRORS

The symbolic link is made unless on or more of the following are true:

[EINVAL]	Either *name1* or *name2* contains a character with the high-order bit set.
[ENOENT]	One of the pathnames specified was too long.
[ENOTDIR]	A component of the *name2* prefix is not a directory.
[EEXIST]	*Name2* already exists.
[EACCES]	A component of the *name2* path prefix denies search permission.
[EROFS]	The file *name2* would reside on a read-only file system.
[EFAULT]	*Name1* or *name2* points outside the process's allocated address space.
[ELOOP]	Too may symbolic links were encountered in translating the pathname.

SEE ALSO

link(2), ln(1), unlink(2)

NAME

sync – update super-block

SYNOPSIS

sync()

DESCRIPTION

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

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

SEE ALSO

fsync(2), sync(8), update(8)

BUGS

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

NAME

syscall – indirect system call

SYNOPSIS

syscall(number, arg, ...)

DESCRIPTION

Syscall performs the system call whose assembly language interface has the specified *number* and arguments *arg*.

The s0 return value of the system call is returned. See /usr/include.

DIAGNOSTICS

Syscall returns -1 and sets the external variable *errno* (see intro(2)) when the call returns unsuccessfully.

NAME

thread_create – create a new thread

SYNOPSIS

tid = **thread_create**(*pc, sp*)
int *tid*;
int *pc*;
int *sp*;

DESCRIPTION

The *thread_create* system call creates a thread in the kernel and starts the created thread at the program counter passed in *pc*, running on the stack address passed in *sp*. The system call performs much the same function as the hardware instruction *pfork*. The primary advantage to the *thread_create* system call over the instruction is that the thread is created even if an idle CPU does not exist.

The user is responsible for maintaining exactly the same state as would be required if a *pfork* instruction were substituted for the *thread_create* system call.

RETURN VALUE

Upon successful completion, *thread_create* returns the thread ID of the thread created to the caller and creates a thread that starts execution at the program counter passed. Otherwise, a value of -1 is returned to the caller, no thread is created, and *errno* is set to indicate the error.

ERRORS

[EFAULT] *pc* points to an address outside the user process segment.

[EACCES] The resource limit *MCONCUR* has been exceeded and thread creation is prohibited.

SEE ALSO

setrlimit(2),
CONVEX Architecture Reference

NAME

truncate, ftruncate, truncate64, ftruncate64 – truncate a file to a specified length

SYNOPSIS

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

truncate(path, length)
char *path;
off_t length;

truncate64(path, length)
char *path;
off64_t length;

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

ftruncate64(fd, length)
int fd;
off64_t length;
```

DESCRIPTION

All variants of *truncate* cause 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* or *ftruncate64*, 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 and *truncate64* succeed unless:

[EINVAL]	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 and *ftruncate64* succeed unless:

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

SEE ALSO

open(2), cvxtruncate(2) "ConvexOS Large Files User's Guide"

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 file creation mode mask

SYNOPSIS

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

mode_t umask(numask)
mode_t numask;
```

DESCRIPTION

umask sets the process's file mode creation mask to *numask* and returns the previous value of the mask. Only the file permission bits of cmask are used, as defined by the inclusive OR of S_IRWXU, S_IRWXG, and S_IRWXO (see *stat*(2)). The permission bits of *numask* are used whenever a file is created, clearing corresponding bits in the file mode (see *chmod*(2)). This clearing allows each user to restrict the default access to his files.

The value is initially (S_IWGRP|S_IWOTH), allowing write access for the file's owner only. The mask is inherited by child processes.

RETURN VALUE

The previous value of the file mode mask is returned by the call.

SEE ALSO

chmod(2), mknod(2), open(2), stat(2)

NAME

uname – get system information

SYNOPSIS

#include <sys/utsname.h>

uname(struct utsname *namep)

DESCRIPTION

uname stores information identifying the operating system and execution environment in the structure pointed to by *namep*. It includes the functionality of the *getsysinfo*(2) and *gethostname*(2) system calls of ConvexOS, although the old calls remain for backward compatibility.

The *utsname* structure is filled in with the following information:

```
#define _UTS_NMLN    64

        struct  utsname {
                struct  {
                        unsigned  short     system_sn;
                        unsigned  char      cpu_type;
                        unsigned  char      cpu_count;
                        unsigned  long      _flags_hi;
                        unsigned  long      flags;
                }       si;
                char    sysname[_UTS_NMLN];
                char    nodename[_UTSNMLN];
                char    release[_UTSNMLN];
                char    version[UTS_NMLN];
                char    machine[_UTSNMLN];
        };
```

The fields of *sysinfo* are interpreted as follows:

system_sn	Serial number of the system on which the program is running.
cpu_type	CPU type of the CPU(s) in the complex.
cpu_count	Number of CPUs in the complex. This duplicates information in the third ASCII digit of the *release* field described below.
flags	Flags indicating hardware and software capabilities. See the header file sys/utsname.h for detailed descriptions of the flags.
_flags_hi	Currently unused.

All remaining fields are null terminated character arrays:

sysname	Name of this implementation of the operating system, e.g. "vmunix".
nodename	Last host name configured with *sethostname*(2); this is the same information as returned by *gethostname*(2).
release	The configuration of the machine, e.g., "C210". Specifically, the first character is "C", the second is "1" or "2". The third is the number of processors, and the fourth is always "0".
version	The version of ConvexOS, e.g., "9.0".
machine	Always returns "convex".

RETURN VALUE

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

ERRORS

[EFAULT] The pointer *namep* is invalid.

SEE ALSO

getsysinfo(2), gethostname(2), sethostname(2)

NAME

remove, unlink – remove a file or directory entry

SYNOPSIS

int unlink(char *path);

#include <stdio.h>
int remove(const char *path);

DESCRIPTION

unlink removes the entry for the file *path* from its directory. If this entry was the last link to the file and no process has the file open, then all resources associated with the file are reclaimed. However, if the file was open in any process, the actual resource reclamation is delayed until it is closed even though the directory entry has disappeared.

remove performs the same function as *unlink*.

RETURN VALUE

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

ERRORS

The *unlink* succeeds unless:

[EINVAL]	The path contains a character with the high-order bit set.
[ENAMETOOLONG]	The pathname is too long.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[ENOENT]	The *path* argument points to the empty string.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EPERM]	The named file is a directory and the effective user ID of the process is not the super-user.
[EPERM]	The directory containing the named file has the sticky bit set and the effective user ID of the process does not match the file owner and is not the super-user.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

BACKWARD COMPATIBILITY

Previous versions of the operating system allowed the empty path name as a synonym for the current directory.

NOTES

The *d_fileno* of the directory entry for the file being removed by *unlink* or *remove* is zeroed, but the name persists in the directory until the directory entry is reused or until the directory itself is removed.

SEE ALSO

close(2), link(2), rmdir(2)

NAME

unmount – remove a file system

SYNOPSIS

unmount(name)
char *name;

DESCRIPTION

unmount announces to the system that the directory *name* is no longer to refer to the root of a mounted file system. The directory *name* reverts to its ordinary interpretation.

RETURN VALUE

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

ERRORS

unmount may fail with one of the following errors:

[EPERM]	The caller is not the superuser.
[EINVAL]	*name* is not the root of a mounted file system.
[EBUSY]	A process is holding a reference to a file located on the file system.
[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	The pathname was too long.
[ENOENT]	*name* does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*name* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

mount(2), mount(8), umount(8)

BUGS

The error codes are in a state of disarray; too many errors appear to the caller as one value.

NAME

utimes – set file times

SYNOPSIS

#include <sys/time.h>

utimes(file, tvp)
char *file;
struct timeval tvp[2];

DESCRIPTION

The *utimes* call uses the "accessed" and "updated" times in that order from the *tvp* vector to set the corresponding recorded times for *file*. If *tvp* is null, the current time is used.

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

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

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

[EINVAL]	The pathname contained a character with the high-order bit set.
[ENOENT]	The pathname was too long.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EACCES]	A component of the path prefix denies search permission.
[EPERM]	The process is not super-user and not the owner of the file.
[EACCES]	The effective user ID is not super-user and not the owner of the file, *times* is NULL, and write access is denied.
[EROFS]	The file resides in a file system which is mounted read only.
[EFAULT]	*tvp* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

stat(2)

NAME

vfork – spawn new process in a virtual memory efficient way

SYNOPSIS

pid = vfork()
int pid;

DESCRIPTION

Vfork can be used to create new processes without fully copying the address space of the old process, which is horrendously inefficient in a paged environment. It is useful when the purpose of *fork*(2) would have been to create a new system context for an *execve*. *Vfork* differs from *fork* in that the child borrows the parent's memory and thread of control until a call to *execve*(2) or an exit (either by a call to *exit*(2) or abnormally.) The parent process is suspended while the child is using its resources.

Vfork returns 0 in the child's context and (later) the pid of the child in the parent's context.

Vfork can normally be used just like *fork*. It does not work, however, to return while running in the childs context from the procedure which called *vfork* since the eventual return from *vfork* would then return to a no longer existent stack frame. Be careful, also, to call *_exit* rather than *exit* if you can't *execve*, since *exit* will flush and close standard I/O channels, and thereby mess up the parent processes standard I/O data structures. (Even with *fork* it is wrong to call *exit* since buffered data would then be flushed twice.)

SEE ALSO

fork(2), execve(2), sigvec(2), wait(2),

DIAGNOSTICS

Same as for *fork*.

BUGS

This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of *vfork* as it will, in that case, be made synonymous to *fork*.

To avoid a possible deadlock situation, processes which are children in the middle of a *vfork* are never sent SIGTTOU or SIGTTIN signals; rather, output or *ioctl*s are allowed and input attempts result in an end-of-file indication.

NAME

vhangup – virtually "hangup" the current control terminal

SYNOPSIS

vhangup()

DESCRIPTION

Vhangup is used by the initialization process *init*(8) (among others) to arrange that users are given "clean" terminals at login, by revoking access of the previous users' processes to the terminal. To effect this, *vhangup* searches the system tables for references to the control terminal of the invoking process, revoking access permissions on each instance of the terminal which it finds. Further attempts to read from the terminal will yield a return value of 0, as if end-of-file had been detected. Further attempts to write or otherwise access the terminal by the affected processes will yield i/o errors (EIO). Finally, a hangup signal (SIGHUP) is sent to the process group of the control terminal.

SEE ALSO

init (8)

BACKWARD COMPATIBILITY

In previous releases of the operating system, attempts to access a 'vhungup' terminal resulted in an *errno* of EBADF.

In previous releases of the operating system, access to the control terminal via **/dev/tty** was still possible; this is no longer the case.

NOTES

This call has been obsoleted by an automatic mechanism which takes place on process exit.

NAME

wait, wait3, waitpid, cvxwait – wait for process to terminate

SYNOPSIS

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

pid_t wait(status)
int *status;

pid_t waitpid(pid, status, options)
pid_t pid;
int *status;
int  options;
```

DESCRIPTION

wait causes its caller to delay until a signal is received or one of its child processes terminates. If any child has died since the last call to *wait*, return is immediate, returning the process ID and exit status of one of the terminated children. If there are no children, return is immediate with the value –1 returned.

The *waitpid()* function behaves identically to *wait()* if the *pid* argument is –1 and the *options* argument is zero. Otherwise, *pid* specifies a set of child processes for which status is requested. If *pid* is –1, status is desired for any child. If *pid* is greater than zero, it specifies a single child for which status is desired. If *pid* is zero, status is requested for any process whose process group ID is equal to that of the caller. If *pid* is less than –1, status is desired for any child whose process group ID is equal to the absolute value of *pid*.

On return from a successful *wait()* or *waitpid()* call, if *status* is not NULL, it is filled in with information revealing the terminated child's cause for exit. Macros are provided in **<sys/wait.h>** to test the exit status.

WIFEXITED(status)
: Evaluates non-zero if status was returned for a child that terminated normally (by calling _exit(2)).

WEXITSTATUS(status)
: If WIFEXITED(status) is non-zero, this macro provides the low-order 8 bits of the argument that the child passed to _exit().

WIFSIGNALED(status)
: Evaluates non-zero if the child terminated due to receipt of a signal that was not caught.

WTERMSIG(status)
: If WIFSIGNALED(status) is non-zero, this macro provides the number of the signal that caused the termination of the child process.

WIFSTOPPED(status)
: Evaluates non-zero if the child is currently stopped.

WSTOPSIG(status)
: If WIFSTOPPED (status) is non-zero, this macro yields the number of the signal that caused the child to stop.

The *options* argument to *waitpid* supports the following values:

WNOHANG
: The caller will not be suspended if no child status is immediately available; in such a case, the return value from *waitpid* will be zero.

WUNTRACED
: Also reports child processes that are stopped and whose status has not been reported since they stopped.

RETURN VALUE

If *wait* or *waitpid* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If *waitpid* was called with WUNTRACED set in *options* and it has at least one child specified by *pid*, but status is available for no child specified *pid*, zero will

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

wait3 and *cvxwait* (see the "CONVEX EXTENSIONS" section below) return –1 if there are no children not previously waited for; 0 is returned if WNOHANG is specified and there are no stopped or exited children.

ERRORS

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

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

[EFAULT] The *status* or *rusage* arguments point to an illegal address.

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

[ECHILD] The process or process group specified by *pid* does not exist or is not a child of the calling process.

[EINTR] The function was interrupted by a signal.

[EINVAL] The *options* argument contains bits other than WNOHANG or WUNTRACED.

SEE ALSO

exit(2)

CONVEX EXTENSIONS

CONVEX supplies the following extensions:

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

pid = wait3(status, options, rusage)
int pid;
int *status;
int options;
struct rusage *rusage;

pid = cvxwait(status, options, rusage)
int pid;
int *status;
int options;
struct cvxrusage *rusage;
```

wait3 provides an alternate interface for programs that must not block when collecting the status of child processes. The *status* and *options* parameters are defined as above. If *rusage* is non-zero, a summary of the resources used by the terminated process and all its children is returned (this information is currently not available for stopped processes).

The *cvxwait* system call is identical to *wait3* but returns a struct cvxrusage instead of a struct rusage. The primary difference between these is that struct cvxrusage contains information about the level of parallelism of the terminated process.

BACKWARD COMPATIBILITY

Previous releases of the operating system documented use of a pointer to *union wait*, where the current release advocates use of a pointer to integer. To convert existing code, simply change calls of the following type: **union wait wait_union; wait(&wait_union)** to this new idiom: **wait(&wait_union.w_status)** or this one: **wait((int *)&wait_union).** The existing code will then continue to work as before.

Previous versions of the operating system by default restarted the *wait* family functions when a signal was received while awaiting termination of a child. See *sigaction(2)* for details of controlling this behavior.

NOTES

See *sigaction*(2) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process that has not terminated and can be restarted. See *pattach*(2). If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

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

NAME

write – write output on a file

SYNOPSIS

```
#include <unistd.h>

write(d, buf, nbytes)
int d;
char *buf;
unsigned nbytes;
```

DESCRIPTION

write attempts to write *nbytes* of data to the object referenced by the descriptor *d* from the buffer pointed to by *buf*.

On objects capable of seeking, the *write* starts at a position given by the pointer associated with *d*. See *lseek*(2). Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object 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.

If the O_LARGEFILE flag of the file status flags is not set, writes at offsets greater than 2^31 - 1 bytes are disallowed. *write* will behave as if the maximum file size is 2^31 - 1 bytes in such cases.

If the real user is not the super-user, then *write* clears the set-user-ID bit on a file. This prevents penetration of system security by a user who "captures" a writable set-user-ID file owned by the super-user.

When using non-blocking I/O on objects that are subject to flow control, such as sockets, pipes (or FIFOs), or terminals, *write* may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible. If such an object's buffers are full, so that it cannot accept any data, then *write* will return –1 and set *errno* to EAGAIN. Otherwise, *write* will block until space becomes available.

CONVEX EXTENSIONS

write may operate synchronously (default) or asynchronously, depending on the FASIO flag set with the *fcntl* system call. Synchronous writes suspend the caller until the system has processed the write request and return the number of bytes actually written. The file position pointer is incremented by the number of bytes actually written. Asynchronous writes return before the request has been fully processed, and unless there are errors in the arguments passed in, asynchronous writes always return the number of bytes requested and increment the file pointer by the number of bytes requested. Use *asiostat*(2). to determine the completion status of an asynchronous transfer.

RETURN VALUE

Upon successful completion, the number of bytes actually written is returned unless the write was asynchronous, in which case the number of bytes requested to be transferred is returned. If unsuccessful, a –1 is returned and *errno* is set to indicate the error.

ERRORS

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

[EAGAIN]	The file was marked for non-blocking I/O, and no data could be written immediately.
[EBADF]	*d* is not a valid descriptor open for writing.
[EDQUOT]	The user's quota of disk blocks on the file system containing the file has been exhausted.
[EFAULT]	Part of the data to be written to the file points outside the process's allocated address space.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
[EINTR]	The call is forced to terminate prematurely due to the arrival of a signal whose _SA_INTERRUPT bit in **sa_flags** is set (see *sigaction*(2))

[EIO] The process is in a background process group and attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU signals, and the process group of the process is orphaned. (The most likely scenario is that the user logged off with a job running in the background that attempts to write to the terminal.)

[EINVAL] The requested transfer size is too big for a single request. In order to prevent possible memory deadlocks, the kernel will refuse to do transfers larger than approximately half the size of physical memory.

[EINVAL] The pointer associated with *d* was negative.

[ENOSPC] There is no free space remaining on the file system containing the file.

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

[EPIPE] An attempt is made to write to a pipe that is not open for reading by any process (or to a socket of type SOCK_STREAM that is connected to a peer socket). Note: an attempted write of this kind will also cause you to receive a SIGPIPE signal from the kernel. If you've not made a special provision to catch or ignore this signal, your process will die.

[ENODMON] *Migration only:* The file referenced by *d* is under daemon control, but no daemon is present.

[EISMIGRATED] *Migration only:* A block in the file reference by *d* is migrated, and the migration daemon was unable to stage the file in from secondary storage.

BACKWARD COMPATIBILITY

Where previous versions of the operating system returned EWOULDBLOCK for *errno*, EAGAIN is now returned.

The default signal behavior is now to interrupt and not restart a *write* operation.

SEE ALSO

asiostat(2), fcntl(2), lseek(2), open(2), pipe(2), select(2)

[EPIPE] An attempt is made to write to a pipe that is not open for reading by any process (or to a socket of type SOCK_STREAM that is connected to a peer socket). Note: an attempted write of this kind will also cause you to receive a SIGPIPE signal from the kernel. If you've not made a special provision to catch or ignore this signal, your process will die.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

[EFAULT] Part of *iov* or data to be written to the file points outside the process's allocated address space.

[EINTR] The call is forced to terminate prematurely due to the arrival of a signal whose SV_INTERRUPT bit in **sv_flags** is set (see *sigvec*(2) or *signal*(3)).

[EINVAL] The pointer associated with *d* was negative.

[ENOSPC] There is no free space remaining on the file system containing the file.

[EDQUOT] The user's quota of disk blocks on the file system containing the file has been exhausted.

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

[EWOULDBLOCK] The file was marked for non-blocking I/O, and no data could be written immediately.

[EINVAL] The requested transfer size is too big for a single request. In order to prevent possible memory deadlocks, the kernel will refuse to do transfers larger than approximately half the size of physical memory.

[EINVAL] For asynchronous requests, a maximum of one *iov* region may point to a given logical page.

[EINVAL] *iovcnt* was either less than or equal to 0 or greater than 16.

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

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

[EINVAL] Two or more *iov* buffers share the same virtual memory page, and the mode is asynchronous. (This restriction is necessary because the pages are locked in memory during asynchronous transfers.)

[ENODMON] *Migration only:* The file referenced by *d* is under daemon control, but no daemon is present.

[EISMIGRATED] *Migration only:* A block in the file reference by *d* is migrated, and the migration daemon was unable to stage the file in from secondary storage.

SEE ALSO

asiostat(2), fcntl(2), lseek(2), open(2), pipe(2), select(2), write(2)

NAME

writev – gather output to file

SYNOPSIS

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

int writev(d, iov, ioveclen)
int d;
struct iovec *iov;
int ioveclen;
```

DESCRIPTION

writev gathers output data from the *ioveclen* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt – 1] and writes the data to the file indicated by descriptor *d*.

The *iovec* structure is defined as

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

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. *writev* will always write a complete area before proceeding to the next.

On objects capable of seeking, the *writev* starts at a position given by the pointer associated with *d*. See *lseek*(2). Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object 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.

writev may operate synchronously (default) or asynchronously, depending on the FASIO flag set with the *fcntl* system call. Synchronous writes suspend the caller until the system has processed the write request and return the number of bytes actually written. The file position pointer is incremented by the number of bytes actually written. Asynchronous writes return before the request has been fully processed, and unless there are errors in the arguments passed in, asynchronous writes always return the number of bytes requested and increment the file pointer by the number of bytes requested. Use *asiostat*(2). to determine the completion status of an asynchronous transfer.

If the real user is not the super-user, then *writev* clears the set-user-ID bit on a file. This prevents penetration of system security by a user who "captures" a writable set-user-ID file owned by the super-user.

When using non-blocking I/O on objects that are subject to flow control, such as sockets, pipes (or FIFOs), or terminals, *writev* may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible. If such an object's buffers are full so that it cannot accept any data, then *writev* will return –1 and set *errno* to EWOULDBLOCK. Otherwise, they will block until space becomes available.

RETURN VALUE

Upon successful completion, the number of bytes actually written is returned unless the write was asynchronous, in which case the number of bytes requested to be transferred is returned. If unsuccessful, a –1 is returned and *errno* is set to indicate the error.

ERRORS

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

[EBADF] *d* is not a valid descriptor open for writing.

Section 3
Library functions

NAME

intro – introduction to library functions

DESCRIPTION

This section describes functions available in libraries. Library functions are those other than the functions which directly invoke ConvexOS system primitives, described in Section 2.

Libraries distributed with ConvexOS are 3 (standard C library functions), 3C (system compatibilty), 3M (math library, libm), 3S (I/O), and 3X (specialized libraries). Man pages for these libraries are included in the ConvexOS hardcopy man pages; an intro man page is printed for each. Man pages for CONVEX Internet Services (an optional product) are also included with ConvexOS hardcopy man pages.

If an optional CONVEX product is installed at your site, you will find man pages for that product online; hardcopy man pages are distributed with the optional product.

Library functions are grouped into the following libraries.

(3) and (3S)
: The plain "3" functions are the standard C library functions. The C library also includes all the functions described in Section 2. The "3S" functions comprise the standard I/O library. Together with the (3X) and (3C) routines, these functions constitute 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.

(3A)
: This section describes functions that are in the CONVEX Ada utility library.

(3BIT)
: This section describes functions included in the C bit manipulation library. These functions are implementations of some of Cray's bit intrinsic functions.

(3B)
: This section describes subprograms included in CONVEX VECLIB (an optional product). VECLIB consists of three libraries of FORTRAN-callable mathematical subprograms that have been optimized for use on CONVEX supercomputers.

(3C)
: Routines included for compatibility with other systems. In particular, a number of system call interfaces provided in previous releases of 4BSD have been included for source code compatibility. The man page entry for each compatibility routine indicates the proper interface to use.

(3F)
: This section describes those functions that are in the FORTRAN utility library. The "3F" functions provide an interface from *fc* programs to the system in the same manner as the C library does for C programs. (CONVEX FORTRAN is an optional product.)

(3M)
: These functions constitute the math library, *libm*. The link editor searches this library under the "-lm" option. Declarations for these functions may be obtained from the include file <*math.h*>.

(3N)
: This section describes network library functions that are applicable to the DARPA Internet network.

(3R)
: These functions comprise the RPC Services Library. This library provides a high-level interface to the RPC Services as well as defining the necessary XDR routines needed to access RPC Services at a much lower level. (CONVEX NFS is an optional product.)

(3S)
: These functions constitute the "standard I/O package", see *intro*(3S). These functions are in the library *libc* already mentioned. Declarations for these functions may be obtained from the include file <*stdio.h*>.

(3SCI)
: This section describes subprograms contained in CONVEX UMSLIB. UMSLIB is a library of FORTRAN-callable mathematical subprograms that emulate many of the routines found in Cray's UNICOS Math and Scientific Library. UMSLIB is packaged with

CONVEX VECLIB.

(3V) These functions are part of the COVUE family library routines. The COVUEnet library is the programmer's interface to the Remote File Access System of COVUEnet. (CONVEX COVUEnet is an optional product.)

(3X) Various specialized libraries have not been given distinctive captions. Files in which such libraries are found are named on appropriate pages.

FILES

/lib/libc.a
/usr/lib/libm.a
/usr/lib/libc_p.a
/usr/lib/libm_p.a
/usr/lib/libnfars.a
/usr/lib/libbint.a

SEE ALSO

intro(3B), intro(3BIT), intro(3C), intro(3F), intro(3S), intro(3SCI), intro(3M), intro(3R), intro(3V), intro(3X), nm(1), ld(1), cc(1), intro(2)

DIAGNOSTICS

Functions in the math library (3M) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable *errno* (see *intro (2)* and *errno.h (3))* is set to the value EDOM (domain error) or ERANGE (range error). The values of EDOM and ERANGE are defined in the include file *<errno.h>*. Many other standard C functions set **errno** to a positive value indicating the error. C programmers may set **errno** to zero, although the C library routines never do.

BACKWARD COMPATIBILITY

The default libraries linked with **cc** command conform to the ANSI and POSIX specifications. The functionality of the routines in the backward-compatible libraries (linked when using the *-pcc* option of **cc**) often is different from the routines in the conforming libraries. When using the *-str* option of the **cc** command, strict conformance to the ANSI standard is supplied. The ANSI standard guarantees the C programmer of a namespace free from conflicting library names. In this mode, the user may use any name not mentioned by ANSI except those beginning with two underscores and those beginning with an underscore and an uppercase letter. As an additional constraint, names beginning with an underscore and a lowercase letter may only be used in a local scope of a routine. If the C programmer uses names that are reserved for the implementation, errors may occur while linking. When using the *-std* option of the **cc** command, additional conformance to the POSIX standard is supplied. In this mode, the ANSI namespace rules are still followed. POSIX does not reserve its names, so the programmer is guaranteed that the accidental redefinition of a POSIX name will not cause any other function to fail.

NAME

intro – introduction to Cray compatible bit manipulation library functions

DESCRIPTION

These functions constitute the C bit manipulation library, *libbint*. The link editor searches this library under the "–lbint" option. Declarations for these functions may be obtained from the include file <*bint.h*>.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
_count	bitcount(3BIT)	count the number of "1" bits
_dshiftl	bitshift(3BIT)	rotate bits left or shift left-most bits of *source* into right-most bits of *target*
_dshiftr	bitshift(3BIT)	rotate bits right or shift right-most bits of *source* into left-most bits of *target*
_gbit	bitchange(3BIT)	get a single bit
_gbits	bitchange(3BIT)	get a consecutive sequence of bits
IBCLR	bitmil(3BIT)	clear a bit
IBITS	bitmil(3BIT)	get a consecutive sequence of bits
IBSET	bitmil(3BIT)	set a bit
_ldzero	bitcount(3BIT)	count the number of leading zeros
_leadz	bitcount(3BIT)	count the number of leading zeros
_mask	bitmask(3BIT)	create a mask of right or left justified "1" bits
_maskl	bitmask(3BIT)	create a mask of left justified bits
_maskr	bitmask(3BIT)	create a mask of right justified bits
MVBITS	bitmil(3BIT)	move a consecutive sequence of bits
_parity	bitcount(3BIT)	determine the parity
_pbit	bitchange(3BIT)	replace a single bit
_pbits	bitchange(3BIT)	replace a consecutive sequence of bits
_popcnt	bitcount(3BIT)	count the number of "1" bits
_poppar	bitcount(3BIT)	determine the parity

DIAGNOSTICS

Some of the functions can be accessed using faster function-like macros. These macros are called by default in the extended and backward-compatible modes. These macros offer increased speed because they call intrinsic functions.

To prevent the intrinsic functions from being linked into your program, compile with the *-D__NO_INLINE_BINT* command line option. You could also compile your program with the *-D__NO_INLINE* command line option which disables all function-like macros.

When an error occurs, the *errno* variable is not set.

FILES

/usr/lib/libbint.a
/usr/include/bint.h

SEE ALSO

bitchange(3BIT), bitmil(3BIT), bitcount(3BIT), bitmask(3BIT), bitshift(3BIT), bint.h(3BIT)

NAME

intro – introduction to compatibility library functions

DESCRIPTION

These functions constitute the compatibility library portion of *libc*. They are automatically loaded as needed by the C compiler *cc*(1). The link editor searches this library under the "–lc" option. Use of these routines should, for the most part, be avoided. Manual entries for the functions in this library describe the proper routine to use.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
alarm	alarm(3C)	schedule signal after specified time
ftime	time(3C)	get date and time
getpw	getpw(3C)	get name from uid
gtty	stty(3C)	set and get terminal state (defunct)
nice	nice(3C)	set program priority
pause	pause(3C)	stop until signal
rand	rand(3C)	random number generator
signal	signal(3C)	simplified software signal facilities
srand	rand(3C)	random number generator
stty	stty(3C)	set and get terminal state (defunct)
time	time(3C)	get date and time
times	times(3C)	get process times
utime	utime(3C)	set file times
vlimit	vlimit(3C)	control maximum system resource consumption
vtimes	vtimes(3C)	get information about resource utilization

NAME

intro – introduction to mathematical library functions

DESCRIPTION

These functions constitute the C math library, *libm*. The link editor searches this library under the "-lm" option. Declarations for these functions may be obtained from the include file *<math.h>*.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
acos	sin(3M)	double-precision arc cosine
acosf	sin(3M)	single-precision arc cosine
asin	sin(3M)	double-precision arc sine
asinf	sin(3M)	single-precision arc sine
atan	sin(3M)	double-precision arc tangent
atanf	sin(3M)	single-precision arc tangent
atan2	sin(3M)	double-precision arc tangent with two arguments
atan2f	sin(3M)	single-precision arc tangent with two arguments
cabs	hypot(3M)	double-precision complex absolute value
ceil	floor(3M)	ceiling function
cos	sin(3M)	double-precision cosine
cosf	sin(3M)	double-precision cosine
cosh	sinh(3M)	double-precision hyperbolic cosine
coshf	sinh(3M)	single-precision hyperbolic cosine
exp	exp(3M)	double-precision exponential
expf	exp(3M)	single-precision exponential
fabs	floor(3M)	double-precision absolute value
fabsf	floor(3M)	single-precision absolute value
floor	floor(3M)	floor function
gamma	gamma(3M)	gamma function
hypot	hypot(3M)	double-precision Euclidean distance
ipow	exp(3M)	integer power
j0	j0(3M)	bessel functions
j1	j0(3M)	bessel functions
jn	j0(3M)	bessel functions
log	exp(3M)	double-precision natural logarithm
logf	exp(3M)	single-precision natural logarithm
log10	exp(3M)	double-precision base 10 logarithm
log10f	exp(3M)	single-precision base 10 logarithm
lpow	exp(3M)	long-long-int power
pow	exp(3M)	double-precision power
powf	exp(3M)	single-precision power
sacos	sin(3M)	single-precision arc cosine (-pcc mode)
sasin	sin(3M)	single-precision arc sine (-pcc mode)
satan	sin(3M)	single-precision arc tangent (-pcc mode)
satan2	sin(3M)	single-precision arc tangent with two arguments (-pcc mode)
scabs	hypot(3M)	single-precision complex absolute value
scos	sin(3M)	single-precision cosine (-pcc mode)
scosh	sinh(3M)	single-precision hyperbolic cosine
sexp	exp(3M)	single-precision exponential (-pcc mode)
sfabs	floor(3M)	single-precision absolute value (-pcc mode)
shypot	hypot(3M)	single-precision Euclidean distance
sin	sin(3M)	double-precision sine
sinf	sin(3M)	single-precision sine

sinh	sinh(3M)	double-precision hyperbolic sine
sinhf	sinh(3M)	single-precision hyperbolic sine
slog	exp(3M)	single-precision natural logarithm (-pcc mode)
slog10	exp(3M)	single-precision base 10 logarithm (-pcc mode)
spow	exp(3M)	single-precision power (-pcc mode)
sqrt	exp(3M)	double-precision square root
sqrtf	exp(3M)	single-precision square root
tan	sin(3M)	double-precision tangent
tanf	sin(3M)	double-precision tangent
tanh	sinh(3M)	double-precision hyperbolic tangent
tanhf	sinh(3M)	single-precision hyperbolic tangent
y0	j0(3M)	bessel functions
y1	j0(3M)	bessel functions
yn	j0(3M)	bessel functions

DIAGNOSTICS

All of the mathematical functions can be called with arguments which lead to error conditions. Errors can be grouped into two categories: domain and range errors. Domain errors occur when the argument of a function is out of the domain of the function. Range errors occur when the computed value is not representable within the machine's precision, or the size of the argument is such that evaluation of the function would lead to a significant loss of accuracy.

The mathematical error codes are listed in the include file < *errno.h* >. In general, the errors reported are more specific than EDOM (domain error) and ERANGE (range error). For example, the error code MTH_NEG_BASE is returned when the floating-point power function is called with a negative base.

On an error condition, a message is printed to *stderr* and the external variable *errno* (see *intro* (2)) is set to the corresponding error code. The return value is function dependent.

Some of the functions can be accessed using faster function-like macros. These macros are called by default in the extended and backward-compatible modes. These macros offer increased speed because they call intrinsic functions. A disadvantage of the intrinsic functions is that they do not report errors using the *errno* variable.

To prevent the intrinsic math functions from being linked into your program, compile with the *-D__NO_INLINE_MATH* command line option. You could also compile your program with the *-D__NO_INLINE* command line option which disables all intrinsic functions. For additional information on the use of intrinsic functions in a program, refer to the *CONVEX C Guide*.

NAME

intro – introduction to network library functions

DESCRIPTION

This section describes functions that are applicable to the DARPA Internet network.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
endhostent	gethostbyname(3N)	get network host entry
endnetent	getnetent(3N)	get network entry
endprotoent	getprotoent(3N)	get protocol entry
endrpcent	getrpcent(3N)	get rpc entry
endservent	getservent(3N)	get service entry
ether_aton	ethers(3N)	ethernet address mapping operations
ether_hostton	ethers(3N)	ethernet address mapping operations
ether_line	ethers(3N)	ethernet address mapping operations
ether_ntoa	ethers(3N)	ethernet address mapping operations
ether_ntohost	ethers(3N)	ethernet address mapping operations
gethostbyaddr	gethostbyname(3N)	get network host entry
gethostbyname	gethostbyname(3N)	get network host entry
gethostent	gethostbyname(3N)	get network host entry
getnetbyaddr	getnetent(3N)	get network entry
getnetbyname	getnetent(3N)	get network entry
getnetent	getnetent(3N)	get network entry
getprotobyname	getprotoent(3N)	get protocol entry
getprotobynumber	getprotoent(3N)	get protocol entry
getprotoent	getprotoent(3N)	get protocol entry
getrpcbyname	getrpcent(3N)	get rpc entry
getrpcbynumber	getrpcent(3N)	get rpc entry
getrpcent	getrpcent(3N)	get rpc entry
getservbyname	getservent(3N)	get service entry
getservbyport	getservent(3N)	get service entry
getservent	getservent(3N)	get service entry
htonl	byteorder(3N)	convert values between host and network byte order
htons	byteorder(3N)	convert values between host and network byte order
inet_addr	inet(3N)	Internet address manipulation routines
inet_lnaof	inet(3N)	Internet address manipulation routines
inet_makeaddr	inet(3N)	Internet address manipulation routines
inet_netof	inet(3N)	Internet address manipulation routines
inet_network	inet(3N)	Internet address manipulation routines
ntohl	byteorder(3N)	convert values between host and network byte order
ntohs	byteorder(3N)	convert values between host and network byte order
rpc	rpc(3N)	library routines for remote procedure calls
sethostent	gethostbyname(3N)	get network host entry
setnetent	getnetent(3N)	get network entry
setprotoent	getprotoent(3N)	get protocol entry
setrpcent	getrpcent(3N)	get rpc entry
setservent	getservent(3N)	get service entry
xdr	xdr(3N)	library routines for external data representation
yp	ypclnt(3N)	yellow pages client interface

NAME

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

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

DESCRIPTION

The functions described in section 3S constitute a user-level buffering scheme. The inline macros *getc* and *putc*(3S) handle characters quickly. The higher level routines *gets*, *fgets*, *scanf*, *fscanf*, *fread*, *puts*, *fputs*, *printf*, *fprintf*, *fwrite* all use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to a defined type **FILE**. *fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

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

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

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

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

SEE ALSO

open(2), close(2), read(2), write(2), fread(3S), fseek(3S), f*(3S)

DIAGNOSTICS

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

For purposes of efficiency, this implementation of the standard library has been changed to line buffer output to a terminal by default and attempts to do this transparently by flushing the output whenever a *read*(2) from the standard input is necessary. This is almost always transparent, but may cause confusion or malfunctioning of programs which use standard I/O routines but use *read*(2) themselves to read from the standard input.

In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to *fflush*(3S) the standard output before going off and computing so that the output will appear.

BUGS

The standard buffered functions do not interact well with certain other library and system functions, especially *vfork* and *abort*.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
clearerr	ferror(3S)	stream status inquiries
fclose	fclose(3S)	close or flush a stream

fdopen	fopen(3S)	open a stream
feof	ferror(3S)	stream status inquiries
ferror	ferror(3S)	stream status inquiries
fflush	fclose(3S)	close or flush a stream
fgetc	getc(3S)	get character or word from stream
fgets	gets(3S)	get a string from a stream
fileno	ferror(3S)	stream status inquiries
fopen	fopen(3S)	open a stream
fprintf	printf(3S)	formatted output conversion
fputc	putc(3S)	put character or word on a stream
fputs	puts(3S)	put a string on a stream
fread	fread(3S)	buffered binary input/output
freopen	fopen(3S)	open a stream
fscanf	scanf(3S)	formatted input conversion
fseek	fseek(3S)	reposition a stream
ftell	fseek(3S)	reposition a stream
fwrite	fread(3S)	buffered binary input/output
getc	getc(3S)	get character or word from stream
getchar	getc(3S)	get character or word from stream
gets	gets(3S)	get a string from a stream
getw	getc(3S)	get character or word from stream
popen	popen(3S)	initiate I/O to/from a process
printf	printf(3S)	formatted output conversion
putc	putc(3S)	put character or word on a stream
putchar	putc(3S)	put character or word on a stream
puts	puts(3S)	put a string on a stream
putw	putc(3S)	put character or word on a stream
rewind	fseek(3S)	reposition a stream
scanf	scanf(3S)	formatted input conversion
setbuf	setbuf(3S)	assign buffering to a stream
setbuffer	setbuf(3S)	assign buffering to a stream
setlinebuf	setbuf(3S)	assign buffering to a stream
sprintf	printf(3S)	formatted output conversion
sscanf	scanf(3S)	formatted input conversion
ungetc	ungetc(3S)	push character back into input stream

NAME

intro – introduction to miscellaneous library functions

DESCRIPTION

These functions constitute minor libraries and other miscellaneous runtime facilities. Most are available only when programming in C. The list below includes libraries which provide device independent plotting functions, terminal independent screen management routines for two dimensional non-bitmap display terminals, functions for managing data bases with inverted indexes, and sundry routines used in executing commands on remote machines. The routines *getdiskbyname*, *rcmd*, *rresvport*, *ruserok*, *getfsent*, *initgroups*, and *rexec* reside in the standard C runtime library "–lc". All other functions are located in separate libraries indicated in each manual entry.

FILES

/lib/libc.a
/usr/lib/libdbm.a
/usr/lib/libtermcap.a
/usr/lib/libcurses.a
/usr/lib/lib2648.a
/usr/lib/libplot.a

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
assert	assert(3X)	program verification
curses	curses(3X)	screen functions with "optimal" cursor motion
dbminit	dbm(3X)	data base subroutines
delete	dbm(3X)	data base subroutines
endfsent	getfsent(3X)	get file system descriptor file entry
fetch	dbm(3X)	data base subroutines
firstkey	dbm(3X)	data base subroutines
getdiskbyname	getdisk(3X)	get disk description by its name
getfsent	getfsent(3X)	get file system descriptor file entry
getfsfile	getfsent(3X)	get file system descriptor file entry
getfsspec	getfsent(3X)	get file system descriptor file entry
getfstype	getfsent(3X)	get file system descriptor file entry
initgroups	initgroups(3X)	initialize group access list
nextkey	dbm(3X)	data base subroutines
plot	plot(3X)	graphics interface
setfsent	getfsent(3X)	get file system descriptor file entry
store	dbm(3X)	data base subroutines
tgetent	termcap(3X)	terminal independent operation routines
tgetflag	termcap(3X)	terminal independent operation routines
tgetnum	termcap(3X)	terminal independent operation routines
tgetstr	termcap(3X)	terminal independent operation routines
tgoto	termcap(3X)	terminal independent operation routines
tputs	termcap(3X)	terminal independent operation routines

NAME

abort – generate a fault

SYNOPSIS

#include <stdlib.h>

void abort(void);

DESCRIPTION

abort sends the signal *SIGABRT* to the executing process. If the process has a handler for *SIGABRT*, the handler is replaced by the default function *SIG_DFL*. Then the old handler of the process is executed. Next, the buffers are flushed and the process is killed with the signal *SIGABRT* and core is dumped.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. *abort* differs in these backward-compatible libraries in the following ways:

- The standard I/O buffers are not flushed in the backward-compatible mode.
- In the backward-compatible library, an illegal instruction is executed, sending **SIGILL** to the process rather than **SIGABRT.**

SEE ALSO

adb(1), sigvec(2), exit(2), kill(2)

DIAGNOSTICS

NAME

abs, labs, div, ldiv – standard integral functions

SYNOPSIS

#include <stdlib.h>

int abs(int j);

long int labs(long int j);

div_t div(int numer, int denom);

ldiv_t ldiv(long int numer, long int denom);

DESCRIPTION

abs and *labs* return the absolute value of the operand.

ldiv and *div* return a structure containing the quotient and remainder of the operands.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- *labs* is unavailable in the backward-compatible mode.
- *div* is unavailable in the backward-compatible mode.
- *ldiv* is unavailable in the backward-compatible mode.

SEE ALSO

floor(3M) for *fabs*

BUGS

Applying the *abs* function to the most negative integer generates a result which is the most negative integer. That is,

abs(0x80000000)

returns 0x80000000 as a result.

NAME

alarm – schedule signal after specified time

SYNOPSIS

unsigned int alarm(seconds)
unsigned int seconds;

DESCRIPTION

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

Alarm requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any alarm request is canceled. Because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 2147483647 seconds.

RETURN VALUE

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

SEE ALSO

sigpause(2), sigvec(2), signal(3C), sleep(3)

NAME

assert – program verification

SYNOPSIS

#include <assert.h>

void assert(int expression)

DESCRIPTION

Assert is a macro that indicates *expression* is expected to be non-zero. It causes a diagnostic comment on the standard error and then an *abort* when *expression* is false (0). Compilation with **NDEBUG** defined as a macro, causes *assert* to be defined as **((void) 0).** Multiple inclusions of *<stdio.h>* may result in different definitions of *assert* macro, depending on whether **NDEBUG** has been defined.

DIAGNOSTICS

'Assertion failed: file *f* line *n.*' *F* is the source file and *n* the source line number of the *assert* statement. Core is dumped.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- *assert* in the backward-compatible mode calls *exit* rather than *abort.*
- Multiple inclusion of *<assert.h>* will result in compile time warnings pertaining to macro redefinition in the backwards-compatible mode.

NAME

atof, atoi, atol, atoll – convert ASCII to numbers

SYNOPSIS

#include <stdlib.h>

double atof(const char *nptr);

atoi(const char *nptr);

long int atol(const char *nptr);

long long int atoll(const char *nptr);

DESCRIPTION

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

atof recognizes an optional string of spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

atoi, *atol*, and *atoll* recognize an optional string of spaces, then an optional sign, then a string of digits.

DIAGNOSTICS

atof, *atoi*, *atol* and *atoll* all return zero on a error in processing the string passed to them.

BACKWARD COMPATIBILITY

atoll is only available in backward-compatible and exteneded modes of CONVEX C.

SEE ALSO

scanf(3S), strtod(3)

BUGS

There are no provisions for overflow.

NAME

bindresvport – bind a socket to a privileged IP port

SYNOPSIS

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

bindresvport(sd, sin)
        int sd;
        struct sockaddr_in *sin;
```

DESCRIPTION

bindresvport is used to bind a socket descriptor to a privileged IP port, that is, a port number in the range 0-1023. The routine returns 0 if it is successful; otherwise, –1 is returned, and *errno* is set to reflect the cause of the error. This routine differs with **rresvport** (see **rcmd**(3N)) in that this works for any IP socket, whereas **rresvport( )** only works for TCP.

Only a socket decriptor created by the superuser can bind to a privileged port.

SEE ALSO

rcmd(3N)

NAME

bint.h, __NO_INLINE_BINT, __NO_INLINE, bint_t – header file contains declarations and function prototypes for Cray-compatible C bit manipulation functions.

SYNOPSIS

#include <bint.h>

typedef long long bint_t;

DESCRIPTION

bint.h contains declarations for C bit manipulation functions. Most of the functions can be accessed using faster function-like macros. If you want to call the function instead of the function-like macro, compile with the *-D__NO_INLINE_BINT* or the *-D__NO_INLINE* command-line options. The latter eliminates use of all function-like macros when a choice between a function or a function-like macro exists. Compile your program with the *-lbint* option at the end of the *cc* command-line; this option tells the linker that the libbint.a library contains the executable form of the bit manipulation functions.

bint_t is the data type returned by all of the bit manipulation functions.

COMPATIBILITY

Because the functions use an argument of type *long long*, only the extended and backward-compatible compatibility modes can be used with the bit manipulation functions. The extended compatibility mode is the default mode of the compiler.

The bit manipulation functions implemented by CONVEX depend on type coercion to match the actual function parameters with the formal function parameters. If you include the < *bint.h* > header file in your program, the arguments will be coerced to the type expected by the library routines.

This coercion may cause a problem with the *_leadz* and *_ldzero* functions which count the number of leading zeros in their argument. If the argument is a *char*, there will be 56 more zeros than expected because the *char* data type is coerced from an 8-bit value to a 64-bit value.

FILES

/usr/lib/libbint.a
/usr/include/bint.h

SEE ALSO

The following man pages describe the Cray compatible bit manipulation functions declared in the *bint.h* header file:

bitchange(3BIT),
bitcount(3BIT),
bitmask(3BIT),
bitmil(3BIT),
bitshift(3BIT).

DIAGNOSTICS

No domain or range checking is performed on these functions. *errno* is not set when an error occurs.

NAME

bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

bcopy(src, dest, length)
char *src, *dest;
int length;

bcmp(b1, b2, length)
char *b1, *b2;
int length;

bzero(b, length)
char *b;
int length;

ffs(i)
int i;

DESCRIPTION

The functions *bcopy*, *bcmp*, and *bzero* operate on variable length strings of bytes. They do not check for null bytes as the routines in *stringcpy*(3), *stringcmp*(3). do.

Bcopy copies *length* bytes from string *src* to the string *dest*.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* 0 bytes in the string *b1*.

Ffs finds the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates the value passed is zero.

BUGS

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy*.

NAME

bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

```
bcopy(src, dest, length)
char *src, *dest;
int length;

bcmp(b1, b2, length)
char *b1, *b2;
int length;

bzero(b, length)
char *b;
int length;

ffs(i)
int i;
```

DESCRIPTION

The functions *bcopy*, *bcmp*, and *bzero* operate on variable length strings of bytes. They do not check for null bytes as the routines in *stringcpy*(3), *stringcmp*(3). do.

Bcopy copies *length* bytes from string *src* to the string *dest*.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* 0 bytes in the string *b1*.

Ffs finds the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates the value passed is zero.

BUGS

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy*.

NAME

htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

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

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;
```

DESCRIPTION

These routines convert 16- and 32-bit quantities between network byte order and host byte order. On machines such as the CONVEX, these routines are defined as null macros in the include file *<netinet/in.h>*.

These routines are most often used in conjunction with Internet addresses and ports as returned by *gethostbyname*(3N) and *getservent*(3N).

SEE ALSO

intro(3N), gethostbyname(3N), getservent(3N)

NOTES

Byteorder is an optional product; for more information, contact your CONVEX sales representative.

NAME

cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed - get/set terminal input/output speed

SYNOPSIS

```
#include <termios.h>

speed_t cfgetospeed(termios_p)
struct termios *termios_p;

int cfsetospeed(termios_p, speed)
struct termios *termios_p;
speed_t speed;

speed_t cfgetispeed(termios_p)
struct termios *termios_p;

int cfsetispeed(termios_p, speed)
struct termios *termios_p;
speed_t speed;
```

DESCRIPTION

cfgetospeed() returns the output baud rate stored in the termios structure pointed to by *termios_p*.

cfsetospeed() sets the output baud rate stored in the termios structure pointed to by *termios_p* to *speed*. The zero baud rate, B0, is used to terminate the connection. If B0 is specified, the modem control lines shall no longer be asserted. Normally, this will disconnect the line.

cfgetispeed() returns the input baud rate stored in the termios structure.

cfsetispeed() sets the input baud rate stored in the termios structure to *speed*. If the input baud rate is set to zero, the input baud rate will be specified by the value of the output baud rate.

The type *speed_t* and the name symbols for the supported speeds are defined in **<termios.h>**.

RETURNS

Both *cfsetispeed()* and *cfsetospeed()* return a value of zero if successful and -1 to indicate the desired speed was not within the range of supported speeds.

NOTES

Attempts to set baud rates unsupported by the hardware are silently ignored.

SEE ALSO

tcgetattr(3), tcsetattr(3), tcgetpgrp(3), tcsetpgrp(3).

NAME

chkpnt - checkpoint a process or process family

SYNOPSIS

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

int chkpnt(class, pid, name, options, signo)
int class;
pid_t pid;
char *name;
int options;
int signo;
```

DESCRIPTION

The *chkpnt* function creates one or more files that contain all the process state information necessary to restart a process or process hierarchy. One checkpoint file is created for each successfully checkpointed process.

The *chkpnt* function is actually implemented by invoking the *chkpnt*(1) utility with the appropriate arguments.

The arguments passed to *chkpnt* have the following meanings:

class

CHKPNT_FAMILY - checkpoint the process *pid* and all of its descendents. The default is to checkpoint only the process specified by *pid.*

CHKPNT_PROC - checkpoint the single process whose process id is *pid.*

pid The identifier of the target process to checkpoint. A *pid* of 0 indicates the calling process.

name The name of the checkpoint files. Checkpoint files are created by default with the name *comm.pid* where *comm* is the trailing component of the command name of the process and *pid* is the process id. If the *name* string is non-null, the checkpoint files will be named *name.pid.*

options Optional actions to be performed by the *chkpnt* routine. The CHKPNT_SIGFAMILY and CHKPNT_SIGROOT flags are mutually exclusive; all other flags will be OR'ed together.

CHKPNT_FORCE - Force a checkpoint even if non-checkpointable conditions exist. This may create a checkpoint file that will not restart.

CHKPNT_KILL - Kill all the target processes if the checkpoint operation was successful. If the checkpoint fails for any reason, the target processes continue normal execution.

CHKPNT_PFD - Interpret the *pid* as a process file descriptor returned from *pattach*(2). Normally, as part of the checkpoint process, the target process is stopped with a call to *pattach.* The CHKPNT_PFD option is used by a parent process that has already called *pattach* to gain exclusive control of the target process.

CHKPNT_SIGFAMILY - Send all of the target processes a signal *signo* after the checkpoint has completed successfully. No signal will be sent if the checkpoint fails.

CHKPNT_SIGROOT - Send the root process of the process hierarchy a *signo* signal. No signal will be sent if the checkpoint fails.

signo The signal to send to the checkpointed processes if the CHKPNT_SIGFAMILY or CHKPNT_SIGROOT option is present.

NOTES

See *chkpnt*(1) for a list of facilities that **cannot** be checkpointed.

Processes may not call *chkpnt*() from within a multithreaded region.

RETURN VALUE

If an error occurs, *chkpnt* returns -1 and the external variable *errno* is set to indicate the cause of the error. Otherwise, a value of 0 is returned.

ERRORS

If any of the following conditions occur, the *chkpnt*() function will return -1 and set *errno* to the corresponding value:

[EACCESS] Search permission is denied on a component of the checkpoint file *path* argument.

[EEXIST] The file *path* already exists.

[EFBIG] The checkpoint file would exceed an implementation-defined maximum file size.

[EINTR] The *chkpnt*() operation was interrupted by a signal.

[EINVAL] An invalid argument was passed to the function call.

[EISCONN] One or more of the target processes has an open pipe with a source outside of the target process set.

[EMFILE] The target process exceeds the maximum number of open file descriptors.

[ENAMETOOLONG] The length of the *path* string exceeds {PATH_MAX}.

[ENOSPC] There is no free space remaining on the device containing *path*.

[ENOTSUP] A file or record lock is held by the target process.

[ENOTSUP] The target process exceeds the maximum number of memory regions.

[ENOTSUP] The target process is prepaged or non-swappable.

[EPERM] The calling process does not have permission to checkpoint one or more of the target processes.

[EPIPE] One or more of the target processes has an open pipe with a destination outside of the target process set.

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

[ESRCH] The *class* argument is CKHPNT_PROC or CHKPNT_FAMILY, and no process exists with the requested process ID *id*.

SEE ALSO

chkpnt(1), restart(1), restart(3), chkpnt(5)

NAME

chkpnt – checkpoint a process or process family

SYNOPSIS

integer function chkpnt (class, pid, name, options, signo)
integer class
integer pid
character*(*) name
integer options
integer signo

DESCRIPTION

The *chkpnt* function creates one or more files that contain all the process state information necessary to restart a process or process hierarchy. One checkpoint file is created for each successfully checkpointed process.

The arguments have the following meanings:

class	The type of checkpoint to perform.
pid	The identifier of the target process to checkpoint. A *pid* of 0 indicates the calling process.
name	The name of the checkpoint files. By default, checkpoint files are created with the name *comm.pid*, where *comm* is the trailing component of the command name of the process and *pid* is the process id. If the argument *name* is non-null, the checkpoint files will be named *name.pid*.
options	Optional actions to be performed by the *chkpnt* routine.
signo	Depending on the value of the *flags* argument, the *signo* parameter may be interpreted as a signal to send to the target process if the checkpoint is successful.

See *chkpnt*(3) for a more complete description of the arguments to *chkpnt*.

RETURN VALUE

chkpnt returns 0 if successful; otherwise an error code is returned.

FILES

/usr/lib/libc_old.a

SEE ALSO

chkpnt(3), restart(3), restart(3f)

NAME

clock - get processor time used

SYNOPSIS

#include <time.h>

clock_t clock(void)

DESCRIPTION

The *clock* function determines the processor time used.

RETURN VALUE

The *clock* function returns the implementation's best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the *clock* function should be divided by the value of the macro **CLOCKS_PER_SEC**. If the processor time used is not available or its value cannot be represented, the function returns the value **(clock_t)-1**.

SEE ALSO

difftime(3), time(3)

NAME

crypt, setkey, encrypt — encryption operations

SYNOPSIS

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

setkey(key)
char *key;

encrypt(block, edflag)
char *block;

DESCRIPTION — UNITED STATES DISTRIBUTION

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

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

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

The argument to the *encrypt* entry is likewise a character array of length 64 containing 0's and 1's. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is 0, the argument is encrypted; if nonzero, it is decrypted.

DESCRIPTION — INTERNATIONAL DISTRIBUTION

This functionality is not supported in the international distribution due to export restrictions on the DES algorithm.

SEE ALSO

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

BUGS

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

NAME

ctermid – generate terminal pathname

SYNOPSIS

```
#include <stdio.h>

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

DESCRIPTION

The *ctermid()* function generates a string that, when used as a pathname, refers to the current controlling terminal for the current process.

If the *ctermid()* function returns a pathname, access to the file is not guaranteed.

RETURNS

If **s** is a **NULL** pointer, the string is generated in an area that may be static (and therefore may be overwritten by each call), the address of which is returned. Otherwise *s* is assumed to point to a character array of at least L_ctermid bytes; the string is placed in this array and the value of *s* is returned. The symbolic constant L_ctermid is defined in **<stdio.h>**, and shall have a value greater than zero.

The *ctermid()* function returns an empty string if the pathname that would refer to the controlling terminal cannot be determined.

SEE ALSO

ttyname(3)

NAME

ctime, localtime, gmtime, asctime, timezone, mktime, strftime – date and time manipulation routines

SYNOPSIS

#include <time.h>

char *ctime(const time_t *clock);

struct tm *localtime(const time_t *clock);

struct tm *gmtime(const time_t *clock);

char *asctime(const struct tm *tm);

char *timezone(int zone, int dst);

time_t mktime(struct tm *timeptr);

size_t strftime(char *s, size_t maxsize, const char *format, const struct tm *timeptr);

DESCRIPTION

ctime converts a time pointed to by *clock* such as returned by *time*(3C) into ASCII and returns a pointer to a 26-character string in the following form. (All the fields have constant width.)

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

localtime and *gmtime* return pointers to structures containing the broken-down time. *localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT, which is the time UNIX uses. (UNIX is a registered trademark of UNIX System Laboratories, Inc.) *asctime* converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file is:

```
struct tm {
        int tm_sec;     /* 0-59    seconds */
        int tm_min;     /* 0-59    minutes */
        int tm_hour;    /* 0-23    hour */
        int tm_mday;    /* 1-31    day of month */
        int tm_mon;     /* 0-11    month */
        int tm_year;    /* 0-      year - 1900 */
        int tm_wday;    /* 0-6     day of week (Sunday = 0) */
        int tm_yday;    /* 0-365   day of year */
        int tm_isdst;   /* flag:   daylight savings time in effect */
};
```

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

When local time is called for, the program consults the system to determine the time zone and whether the U.S.A., Australian, Eastern European, Middle European, or Western European daylight saving time adjustment is appropriate. The program knows about various peculiarities in time conversion over the past 10-20 years; if necessary, this understanding can be extended.

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

mktime converts the broken down local time, contained in a *struct tm* into a calendar time value with the same encoding as that of the values returned by the *time* function. The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated. This allows positive values of *tm_isdst* to indicate that Daylight Saving Time is initially in effect, zero to indicate that Daylight Saving Time is not initially in effect, and negative values indicate that *mktime* is to attempt to determine if Daylight Saving Time is in effect for the specified time.

Upon successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined.

strftime places characters in the array pointed to by *s* as controlled by the string pointed to by *format*. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The *format* string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a % character followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than *maxsize* characters are placed in the array. Each conversion specifier is replaced by appropriate characters as described in the following table. The appropriate characters are determined by the *LC_TIME* category of the current locale and by the values contained in the structure pointed to by *timeptr*.

Specifier	Definition
%a	locale's abbreviated weekday name.
%A	locale's full weekday name.
%b	locale's abbreviated month name.
%B	locale's full month name.
%c	locale's appropriate date and time representation.
%d	day of the month as a decimal number (01-31).
%H	hour (24 hour clock) as a decimal number (00-23).
%I	hour (12 hour clock) as a decimal number (01-12).
%j	day of the year as a decimal number (001-366).
%m	month as a decimal number (01-12).
%M	minute as a decimal number (00-59).
%p	locale's equivalent of the AM/PM designations associated with a 12 hour clock.
%S	second as a decimal number (00-61).
%U	week of the year as a decimal number (00-53, first Sunday as first day of week 1).
%w	weekday as a decimal number (0-6) where Sunday is 0.
%W	week of the year as a decimal number (00-53, first Monday as first day of week 1).
%x	locale's appropriate date representation.
%X	locale's appropriate time representation.
%y	year without century as a decimal number (00-99).
%Y	year with century as a decimal number.
%Z	time zone name or abbreviation, or empty string if there is no time zone information.
%%	the % character

If a conversion specifier is not one of the above, the behavior is undefined.

strftime returns the number of characters actually placed in the array, not including the null character, if the total number of characters, including the null character, is less than *maxsize*. Otherwise, zero is returned, and the contents of the array are indeterminate.

SEE ALSO

gettimeofday(2), time(3c), tzset(3)

BUGS

The return values of mktime, ctime, localtime, and gmtime, point to static data whose content is overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii, toupper, tolower, _toupper, _tolower, toascii – character testing and mapping macros

SYNOPSIS

#include <ctype.h>

int isalpha(int c);

. . .

DESCRIPTION

These functions and macros are used for testing and mapping characters.

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

isalpha	*c* is a letter
isupper	*c* is an uppercase letter
islower	*c* is a lowercase letter
isdigit	*c* is a digit
isxdigit	*c* is a hexadecimal digit
isalnum	*c* is an alphanumeric character
isspace	*c* is a space, tab, carriage return, newline, vertical tab, or formfeed
ispunct	*c* is a punctuation character (neither control nor alphanumeric)
isprint	*c* is a printing character, code 040(8) (space) through 0176 (tilde)
isgraph	*c* is a printing character, similar to *isprint* except false for space
iscntrl	*c* is a delete character (0177) or ordinary control character (less than 040)
isascii	*c* is an ASCII character, code less than 0200

The next five macros and functions are used for character mapping. The first two are functions; the remaining are macros.

tolower	if the argument is an uppercase letter, the corresponding lowercase letter is returned; otherwise the argument is returned unchanged.
toupper	if the argument is a lowercase letter, the corresponding uppercase letter is returned; otherwise the argument is returned unchanged.
_tolower	accomplishes the same thing as *tolower*, but has a restricted domain and is faster. If the argument to *_tolower* is not an upper-case letter, its result is undefined.
_toupper	accomplishes the same thing as *toupper*, but has a restricted domain and is faster. If the argument to *_toupper* is not an lower-case letter, its result is undefined.
toascii	yields its argument with all bits turned off that are not part of the standard ASCII character.

SEE ALSO

ascii(7)

BUGS

toupper and *tolower* are System V compatible and if you intend to write codes that will be ported

to other BSD systems you should use the following constructs:

```
if(isupper(c))
    c = tolower(c);

if(islower(c))
    c = toupper(c);
```

Additionally, if speed is an issue, you could include the following defines in your CONVEX source.

```
#define tolower _tolower
#define toupper _toupper
```

NAME

curses – screen functions with "optimal" cursor motion

SYNOPSIS

#include <curses.h>

cc [flags] files **-lcurses -ltermcap** [libraries]

DESCRIPTION

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

SEE ALSO

"Screen Updating and Cursor Movement Optimization: A Library Package" in the *ConvexOS Tutorial Papers*
ioctl(2), getenv(3), tty(4), termcap(5)

AUTHOR

Ken Arnold

FUNCTIONS

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

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

NAME

cuserid, getlogin – get user name

SYNOPSIS

char *getlogin()

#include <stdio.h>

char *cuserid(s)
char *s;

DESCRIPTION

These functions return a string giving the name of the user associated with the current process. The *cuserid()* function returns a name associated with the effective user ID of the process, and the *getlogin()* function returns the name associated by the login activity with the control terminal.

The recommended procedure is either to call the *cuserid()* function, or to call *getlogin()* and, if that fails, to call the *getpwuid()* function with the name returned by the *getuid()* function.

The *getlogin()* function returns a pointer to the user's login name. The same user ID may be shared by several login names. Therefore, to ensure that the correct user database entry is found, the *getlogin()* function should be used with the *getpwnam()* function.

If *getlogin()* returns a non-**NULL** pointer, that pointer is to the name the user logged in under, even if there are several login names with the same user ID.

The *cuserid()* function generates a character representation of the login name of the owner of the current process. If **s** is not a **NULL** pointer, it is assumed that **s** points to an array of at least L_cuserid bytes; the representation is returned in this array. The symbolic constant L_cuserid is defined in **<stdio.h>**, and shall have a value greater than zero.

RETURNS

The *getlogin()* function returns a pointer to a string containing the user's login name, or a **NULL** pointer if the user's login name cannot be found.

The return value from *getlogin()* may point to static data and therefore may be overwritten by each call.

If **s** is a **NULL** pointer, the result from *cuserid()* is generated in an area that may be static, the address of which is returned. If the login name cannot be found, *cuserid()* returns a **NULL** Pointer. If **s** is not a **NULL** pointer, **s** is returned. If the login name cannot be found, the null character shall be placed at ***s**. The return value from *cuserid() may point to static data and therefor may* overwritten by each call.

The implementation of the *cuserid()* function may use the *getpwnam()* function, as the results of a user's call to either routine may be overwritten by a subsequent call to the other routine.

SEE ALSO

getpwnam(3), getpwuid(3)

NAME

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

SYNOPSIS

```
#include <dbm.h>

typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

DESCRIPTION

Note: the dbm library has been superceded by ndbm(3), and is now implemented using ndbm. These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option **–ldbm.**

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

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

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *Firstkey* will return the first key in the database. With any key *nextkey* will return the next key in the database. This code will traverse the data base:

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

DIAGNOSTICS

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

SEE ALSO

ndbm(3)

BUGS

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

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

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Store* will return an error in the event that a disk block fills with inseparable data.

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

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

NAME

difftime - compute the difference between two times.

SYNOPSIS

#include <time.h>

double difftime(time_t time1, time_t time0);

DESCRIPTION

difftime computes the difference between two calendar times: **time1 - time0.**

RETURN VALUE

difftime returns the difference expressed in seconds as a **double**.

SEE ALSO

time(3c)

NAME

opendir, readdir, rewinddir, closedir, telldir, seekdir – directory operations

SYNOPSIS

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

DIR *opendir(filename)
char *filename;

struct dirent *readdir(dirp)
DIR *dirp;

void rewinddir(dirp)
DIR *dirp;

closedir(dirp)
DIR *dirp;
```

DESCRIPTION

opendir() opens the directory named by *filename* and associates a *directory stream* with it. *opendir()* returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer **NULL** is returned if *filename* cannot be accessed, or if it cannot *malloc*(3) enough memory to hold the whole thing.

readdir() returns a pointer to the next directory entry. It returns **NULL** upon reaching the end of the directory (or detecting an invalid *seekdir()* operation, see **CONVEX EXTENSIONS** below). *rewinddir()* resets the position of the named *directory stream* to the beginning of the directory.

closedir() closes the named *directory stream* and frees the structure associated with the DIR pointer.

Sample code that searches a directory for entry "name" is:

```
len = strlen(name);
dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
        if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                closedir(dirp);
                return FOUND;
        }
closedir(dirp);
return NOT_FOUND;
```

CONVEX EXTENSIONS

```
long telldir(dirp)
DIR *dirp;

seekdir(dirp, loc)
DIR *dirp;
long loc;
```

seekdir() sets the position of the next *readdir()* operation on the *directory stream*. The new position reverts to the one associated with the *directory stream* when the *telldir()* operation was performed. Values returned by *telldir()* are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then reopened, the *telldir()* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir()* value

immediately after a call to *opendir()* and before any calls to *readdir()*.

BACKWARD COMPATIBILITY

Old versions of the operating system wanted **<sys/dir.h>**, not **<dirent.h>**, to be included.

SEE ALSO

open(2), close(2), getdirentries(2), lseek(2), read(2), getwd(3), scandir(3), dir(5)

NAME

ecvt, fcvt, gcvt – output conversion

SYNOPSIS

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

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

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

DESCRIPTION

Ecvt converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer thereto. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero. The low-order digit is rounded.

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

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

SEE ALSO

printf(3s)

BUGS

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

NAME

end, etext, edata – last locations in program

SYNOPSIS

extern end;
extern etext;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region.

When execution begins, the program break coincides with *end*, but it is reset by the routines *brk*(2), *malloc*(3), standard input/output (*stdio*(3)), the profile (**-p**) option of *cc*(1), etc. The current value of the program break is reliably returned by 'sbrk(0)', see *brk*(2).

SEE ALSO

brk(2), malloc(3)

NAME

erf, erfc – error functions

SYNOPSIS

double erf(x)
double x;

double erfc(x)
double x;

DESCRIPTION

Erf(x) returns the error function of x; where $\mathrm{erf}(x) := (2/\sqrt{\pi}) \int_0^x \exp(-t^2)\,dt$.

Erfc(x) returns 1.0–erf(x).

The entry for erfc is provided because of the extreme loss of relative accuracy if erf(x) is called for large x and the result subtracted from 1. (e.g. for x = 10, 12 places are lost).

SEE ALSO

intro(3m)

NAME

errno.h, errno – header file relating to error reporting

SYNOPSIS

#include <errno.h>

int errno;

DESCRIPTION

The value of **errno** is zero at program startup. Library functions set **errno** to positive values indicating various error conditions. No library function sets **errno** to zero, although the C programmer may do so. The programmer should not set **errno** to any value except zero. *errno.h* contains the values (macros) that **errno** may acquire. All names beginning with **E** and followed by a digit or uppercase letter are reserved by the implementation for use as macro definitions for error conditions.

The *intro*(2) man page describes the various errno's and their values.

SEE ALSO

intro(2)

NAME

execl, execle, execlp, exect, execv, execvp, environ – execute a file

SYNOPSIS

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

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

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

extern char **environ;
```

DESCRIPTION

These routines provide various interfaces to the *execve*(2) system call. Refer to *execve*(2) for a description of their properties; only brief descriptions are provided here.

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

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg*[*0*], *arg*[*1*] ... address null-terminated strings. Conventionally *arg*[*0*] is the name of the file.

Two interfaces are available. *execl()* is useful when a known file with known arguments is being called; the arguments to *execl()* are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A null pointer argument must end the argument list.

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

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

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

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least 1 and the first member of the array points to a string containing the name of the file.

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

envp is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an "=", and a null-terminated value. The array of pointers is terminated by a null pointer. The shell *sh*(1) passes an environment entry for each global shell variable defined when the program is called. See *environ*(7) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell *environ*, which is used by *execv()* and *execl()* to pass the environment to any subprograms executed by the current program.

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

RETURN VALUES

The *execl* calls return a -1 if the file cannot be found, or if it is not executable, or if it does not start with a valid magic number (see *a.out*(5)), or if maximum memory is exceeded, or if the arguments require too much space,

There is no return from a successful call.

CONVEX EXTENSIONS

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

The *exect()* version is used when the executed file is to be manipulated with *pattach*(2). The program is forced to single step a single instruction giving the parent an opportunity to manipulate its state.

NOTES

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

execvp() should know that files beginning with a "#" are intended to be processed by /bin/csh and not /bin/sh.

At least 1 of the execute-permission bits must be set for a file to be executed, even for the superuser.

SEE ALSO

sh(1), csh(1), execve(2), fork(2), environ(7)

NAME

atexit, exit – terminate a process after flushing any pending output

SYNOPSIS

#include <stdlib.h>

int atexit(void (*func)(void));

void exit(int status);

DESCRIPTION

atexit registers the function, *func*, for execution at program termination; that is, *func* will be executed when the program exits. On program exit, functions registered are executed in the reverse order of registration.

exit terminates a process after flushing any buffered output.

RETURNS

atexit returns 0 if successful and non-zero if the request could not be honored. Currently, up to 50 functions may be registered with the *atexit* function.

exit never returns.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- *atexit* is unavailable in the backward-compatible mode.

SEE ALSO

exit(2), intro(3S), fclose(3S)

NAME

exp, ipow, log, log10, lpow, pow, sqrt, expf, logf, log10f, powf, sqrtf, sexp, slog, slog10, spow, ssqrt – exponential, logarithm, power, square root

SYNOPSIS

#include <math.h>

double exp(double x);

int ipow(int x, int y);

double log(double x);

double log10(double x);

long long int lpow(long long int x, long long int y);

double pow(double x, double y);

double sqrt(double x);

float expf(float x);

float logf(float x);

float log10f(float x);

float powf(float x, float y);

float sqrtf(float x);

DESCRIPTION

exp returns the double-precision exponential function of x; *expf* returns the single-precision exponential.

log returns the double-precision natural logarithm of x; *logf* returns the single-precision natural logarithm.

log10 returns the double-precision base 10 logarithm of x; *log10f* returns the single-precision base 10 logarithm.

pow returns x^y for double-precision x and y; *powf* returns x^y for single-precision x and y. *ipow* returns x^y for integer x and y; *lpow* returns x^y for long long integer x and y.

sqrt returns the double-precision square root of x; *sqrtf* returns the single-precision square root.

SEE ALSO

hypot(3M), sinh(3M), intro(3M)

DIAGNOSTICS

The exponential, logarithmic, and power functions set **errno** to **EDOM** on domain error. The power and exponential functions set **errno** to **ERANGE** on range error. On overflow, **HUGE_VAL** of the same sign is returned.

Evaluation of the exponential function can lead to overflow. The valid range of arguments is [ln(*xmin*),ln(*xmax*)], where *xmin* is the smallest positive floating-point value and *xmax* is the largest positive floating-point value. Arguments outside of this range are replaced by ln(*xmin*) or ln(*xmax*) and evaluation is continued. The logarithm functions are undefined for nonpositive arguments. If x is negative, evaluation continues with $|x|$. If x is zero, the most negative floating-point value is returned. The power functions are undefined when x in x^y is negative and y is not integral. In this case, evaluation continues with $|x|$. The power functions are undefined when y in x^y is nonpositive, and x is zero. In this case, the largest floating-point value is returned. The square root functions are undefined for negative x. The square root functions return **sqrt(abs(**x**))** for negative x.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- The names of the floating-point routines are *sexp, slog, slog10, spow,* and *ssqrt.*
- In the backward-compatible mode **errno** is set to **MTH_OVF_EXP, MTH_UNDEF_LOG, MTH_NEG_BASE, MTH_ZERO_BASE, MTH_OVF_POW,** or **MTH_UNDEF_SQRT** on domain and range error. A message describing the error is printed.

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose(FILE *stream);

int fflush(FILE *stream);

DESCRIPTION

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

fclose is performed automatically upon calling *exit*(3).

fflush causes any buffered data for the named output *stream* to be written to that file. If the file is opened for reading, unread buffered data is invalidated. The stream remains open. If **NULL** is supplied as an argument, all the open streams are flushed.

RETURN VALUE

fclose returns 0 if *stream* is successfully closed. If an error occurs, **EOF** is returned.

fflush returns 0 if the buffers associated with *stream* are successfully emptied. If an error occurs, **EOF** is returned.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- The backward-compatible mode does not recognize **NULL** as a special argument to *fflush*.
- The backward-compatible mode does not always set **errno** as the conforming *fflush* does.
- *fflush* in backward-compatible mode does not invalidate the buffer or call the *lseek* routine.
- *fclose* calls *fflush* to flush the buffers. *fclose* differs as *fflush* differs in the backward-compatible mode.

SEE ALSO

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

DIAGNOSTICS

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

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

#include <stdio.h>

int feof(FILE *stream);

int ferror(FILE *stream);

void clearerr(FILE *stream);

int fileno(FILE *stream);

DESCRIPTION

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

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

clearerr resets the error indication on the named *stream*.

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

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

SEE ALSO

fopen(3S), open(2)

NAME

float.h – header file containing information on floating-point numbers

SYNOPSIS

#include <float.h>

DESCRIPTION

float.h contains macros for describing the floating-point representation available on the machine.

FLT_RADIX is the radix of the machine. Because CONVEX uses a binary representation the **FLT_RADIX** is 2.

In the following macros, a suffix of **FLT** refers to a **float; DBL** refers to a **double;** and **LDBL** refers to a **long double.**

FLT_MANT_DIG, DBL_MANT_DIG, and **LDBL_MANT_DIG** describe the number bits (binary digits) in the mantissa.

FLT_DIG, DBL_DIG, and **LDBL_DIG** are the number of accurate decimal digits a floating-point number.

FLT_MIN_EXP, DBL_MIN_EXP, and **LDBL_MIN_EXP** are the smallest negative integers such that 2 to that power minus one are normalized floating-point numbers.

FLT_MIN_10_EXP, DBL_MIN_10_EXP, and **LDBL_MIN_10_EXP** are the smallest negative integers such that 10 raised to that power are in the range of normalized floating-point numbers.

FLT_MAX_EXP, DBL_MAX_EXP, and **LDBL_MAX_EXP** are the largest integers such that 2 to that power minus one are normalized floating-point numbers.

FLT_MIN_10_EXP, DBL_MIN_10_EXP, and **LDBL_MIN_10_EXP** are the largest integers such that 10 raised to that power are in the range of normalized floating-point numbers.

FLT_MAX, DBL_MAX, LDBL_MAX, FLT_MIN, DBL_MIN, and **LDBL_MIN** are the largest and smallest floating-point numbers.

FLT_EPSILON, DBL_EPSILON, and **LDBL_EPSILON** represent the difference between 1.0 and the next representable floating-point number.

NAME

fabs, floor, ceil, fabsf, sfabs – absolute value, floor, ceiling functions

SYNOPSIS

#include <math.h>

double floor(double x);

double ceil(double x);

double fabs(double x);

float fabsf(float x);

DESCRIPTION

fabs returns the double-precision absolute value $|x|$; *fabsf* returns the single-precision absolute value $|x|$.

floor returns the largest integer not greater than x.

ceil returns the smallest integer not less than x.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- The name of the floating-point routine is *sfabs*.

SEE ALSO

abs(3)

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);

FILE *freopen(const char *filename, const char *mode, FILE *stream);

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

DESCRIPTION

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

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

"r" open for reading

"w" create (or truncate to zero length) for writing

"a" append: open for writing at end of file, or create for writing. All subsequent writes go to the end of the file regardless of intervening calls to *fseek* or other file positioning functions. The file position indicator is placed at the end of the file when the file is opened.

"rb", "wb", and "ab" are the same as "r", "w", and "a". They indicate binary files and are for compatibility with other operating systems because text files and binary files are identical on ConvexOS.

"r+" open for reading and writing.

"w+" create (or truncate to zero length) for writing and reading.

"a+" Open for appending and reading. All subsequent writes go to the end of the file regardless of intervening calls to the *fseek* or other file positioning functions. The file position indicator is placed at the end of the file when the file is opened.

"rb+", "r+b", "wb+", "w+b", "ab+", and "a+b" are the same as "r+", "w+", and "a+". They indicate binary files and are for compatibility with other operating systems because text files and binary files are identical on ConvexOS.

Using an "l" as a portion of the *mode* string opens the file for access with offsets of greater than two gigabytes, and file sizes up to one terabyte.

Both reads and writes may be used on read/write streams, with the limitation that an *fseek, fseek64, rewind,* or reading an end-of-file must be used between a read and a write or vice-versa.

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

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

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

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- Files opened for append with the backward-compatible *fopen* and *freopen* do not insure that all writes are to the end of the file. Writes occur at the location specified by the *current* file position indicator. The initial position of the file position indicator is to the end of the file.

- The binary modes are not available with *fopen* and *freopen* in the backward-compatible libraries.

SEE ALSO

open(2), fclose(3s), fseek(3s)

DIAGNOSTICS

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

NAME

fread, fwrite – buffered binary input/output

SYNOPSIS

#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

DESCRIPTION

fread reads, into a block beginning at *ptr*, *nmemb* items of data of size *size* from the named input *stream*. It returns the number of items actually read. If *stream* is **stdin** and the standard output is line buffered, then any partial output line will be flushed before any call to *read*(2) to satisfy the *fread*.

fwrite appends at most *nmemb* items of data of size *size* beginning at *ptr* , a pointer to the named output *stream*. It returns the number of items actually written.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- **Errno** is not always set on error in the backward compatible mode.

SEE ALSO

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

DIAGNOSTICS

fread and *fwrite* return 0 upon end of file or error and **errno** is set to a nonzero value.

NAME

frexp, ldexp, modf, fmod – split into mantissa and exponent

SYNOPSIS

#include <math.h>

double frexp(double value, int *eptr);

double ldexp(double value, int exp);

double modf(double value, double *iptr);

double fmod(double x, double y);

DESCRIPTION

frexp returns the mantissa of a double *value* as a double quantity, x, of magnitude less than 1 and stores an integer n such that $value = x * 2^n$ indirectly through *eptr*.

ldexp returns the quantity $value * 2^{exp}$.

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

fmod computes the floating point remainder of x and y.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- *fmod* is not available in the backward-compatible mode.

NAME

fseek, ftell, fsetpos, fgetpos, rewind, fseek64, ftell64 – reposition a stream

SYNOPSIS

#include <stdio.h>

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

long int ftell(FILE *stream);

int fsetpos(FILE *stream, const fpos_t *pos);

int fgetpos(FILE *stream, fpos_t *pos);

void rewind(FILE *stream);

int fseek64(FILE *stream, off64_t offset, int ptrname);

off64_t ftell64(FILE *stream);

DESCRIPTION

fseek and *fseek64* set the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according as *ptrname* has the value **SEEK_SET** (beginning of file), **SEEK_CUR** (current value of file position indicator), or **SEEK_END** (end of file). *fseek* and *fseek64* undo any effects of *ungetc*(3S).

ftell and *ftell64* return the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes on the ConvexOS operating system.

On error, -1 is returned and **errno** is set to the value **EBADF, ESPIPE,** or **EINVAL.**

fsetpos sets the file pointer according to the value of the object to pointed to by *pos*. This value may only be obtained by an earlier call to *fgetpos* on the same stream. The *eof* indicator will be cleared for that file. Input or output may be followed by a call to *fgetpos*. *fgetpos* gets the current file position indicator for the stream. For binary streams (all streams on ConvexOS are binary), it is the number of characters from the beginning of the stream.

rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0).

RETURN VALUE

If *fseek* or *fseek64* is successful, a 0 is returned. *fseek* and *fseek64* return –1 for improper seeks. *fsetpos* returns 0 if successful and –1 on error. **errno** is set to **EBADF, ESPIPE,** or **EINVAL** on error.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- *ftell* and *ftell64* do not always set **errno** on error.
- The *fseek64*, *ftell64*, *fgetpos* and *fsetpos* are not available in the backward-compatible mode.

CONVEX EXTENSIONS

fseek64 and *ftell64* are CONVEX extensions to the ANSI C standard, which allow access to files larger than two gigabytes in size. They are only available in the extended (*-ext*) mode of the compiler.

Successful calls to *fseek64* and *ftell64* result in making the stream valid for large file access.

fsetpos and *fgetpos* use an *fpos_t* object which is valid for the entire possible range of file sizes.

SEE ALSO

lseek(2), fopen(3S)

NAME

ftime – get formatted date and time

SYNOPSIS

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

DESCRIPTION

This interface is obsoleted by *gettimeofday*(2).

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

```
/*      $CHeader: timeb.h 1.1 90/11/06 00:18:29 $*/
/*      Copyright 1984 Convex Computer Corp.*/

#ifndef _SYS_TIMEB_H_
#define _SYS_TIMEB_H_

#ifdef  _KERNEL
#include        "h/time.h"
#else
#include        <sys/time.h>
#endif

/*
 * Structure returned by ftime system call
 */
struct timeb
{
        time_t    time;
        unsigned short millitm;
        short     timezone;
        short     dstflag;
};

#endif
```

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

SEE ALSO

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

NAME

gamma – log gamma function

SYNOPSIS

#include <math.h>

double gamma(x)
double x;

DESCRIPTION

Gamma returns ln $|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer *signgam*. The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.0)
        error();
y = exp(y);
if(signgam < 0)
        y = -y;
```

DIAGNOSTICS

A huge value is returned for negative integer arguments.

BUGS

There should be a positive indication of error.

NAME

getactent, getactaid, getactnam, setactent, endactent – get activity file entry

SYNOPSIS

```
#include <act.h>

struct actstruct *getactent()

struct actstruct *getactaid(aid)
int aid;

struct actstruct *getactnam(name)
char *name;

setactent()

endactent()
```

DESCRIPTION

Getactent, getactaid, and *getactnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the activities file.

```
/*      $CHeader: act.h 0.1 86/01/28 19:43:10 $*/
/*      Copyright 1984 Convex Computer Corp.*/
struct actstruct
{
   char  *act_name;        /* activity name */
   int    act_aid;         /* activity ID */
};

#define ACTFILE         "/etc/activities"
struct actstruct *getactent (), *getactnam (), *getactaid ();
```

Getactent simply reads the next line while *getactaid* and *getactnam* search until a matching *aid* or *name* is found (or until EOF is encountered). Each routine picks up where the others leave off so successive calls may be used to search the entire file. When using these routines it is not necessary to first open the file with *setactent* or to close the file with *endactent.*

A call to *setactent* has the effect of rewinding the activity file to allow repeated searches. *Endactent* may be called to close the activity file when processing is complete.

FILES

/etc/activities

SEE ALSO

activities(5),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

DIAGNOSTICS

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

BUGS

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

NAME

getacwent, setacwent, endacwent – get actwho file entry

SYNOPSIS

#include <actwho.h>

struct actwho *getacwent()

void setacwent()

void endacwent()

DESCRIPTION

Getacwent returns a pointer to an object with the following structure containing the broken-out fields of a line in the group-activity access control file. This routine is useful for user programs which sort or summarize */etc/actwho*. An example of a program that would use *getacwent* is a program that reads the log file generated by the *bill* command every night and sorts the *actwho* file to optimize search time, putting entries referenced most often at the beginning of the *actwho* file.

```
/*      $CHeader: actwho.h 0.3 86/02/24 17:37:03 $*/
/*      Copyright 1984 Convex Computer Corp.*/
struct actwho {
        char        *acw_group;    /* project */
        char        *acw_activity; /* activity */
        char        **acw_members;/* list of users */
};

#define ACTWHOFILE              "/etc/actwho"
#define ACTWHOFILELOCK          "/etc/actwho.lock"
struct actwho                   *getacwent();
```

The members of this structure are:

acw_group
: The group name part of the group-activity combination for which access is being defined.

acw_activity
: The activity name part of the group-activity combination for which access is being defined.

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

Getacwent reads the next line in the *actwho* file until EOF is encountered. Each call picks up where the previous call left off, so successive calls may be used to search the entire file. When using this routine it is not necessary to open the file with *setacwent* or to close the file with *endacwent*.

A call to *setacwent* has the effect of rewinding the *actwho* file to allow repeated searches. *Endacwent* may be called to close the *actwho* file when processing is complete.

FILES

/etc/actwho

SEE ALSO

actwho(5),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

DIAGNOSTICS

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

BUGS

All information is contained in a static area, so it must be copied if it is to be saved. Lines longer than 1024 characters can cause unpredictable results.

NAME

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

SYNOPSIS

```
#include <stdio.h>

int getc(FILE *stream);

int getchar(void);

int fgetc(FILE *stream);

int getw(FILE *stream);
```

DESCRIPTION

getc returns the next character from the named input *stream*.

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

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

BUGS

The end-of-file return from *getchar* is incompatible with that in UNIX editions 1-6. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

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

To get *lint* to be quiet about references to *getc* and *getchar*, one must actually use the value that these functions return; using **void** will not work. Below is an example of a reference to *getc* that will keep *lint* happy:

```
if (getc (stdin) == EOF) {
        fprintf (stderr, "Could not getc from stdin.\n");
        exit (1);
}
```

NAME

getcwd – get the current working directory for the process

SYNOPSIS

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

DESCRIPTION

The *getcwd()* function copies an absolute pathname of the current working directory into the character array pointed to by the argument *buf* and returns a pointer to the result. The *size* argument is the size in bytes of the character array pointed to by the *buf* argument. If *buf* is a **NULL** pointer, the behavior of *getcwd()* is undefined.

If successful, the *buf* pointer is returned. A **NULL** pointer is returned if an error occurs and the variable *errno* is set to indicate the error. The contents of *buf* after an error are undefined.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The *getcwd()* function is a POSIX function. Consequently, it is unavailable in the backward-compatibility mode (*-pcc*) of the CONVEX C compiler.

DIAGNOSTICS

If any of the following conditions occur, the *getcwd()* function returns a value of **NULL** and set *errno* to indicate the error.

[EINVAL] The *size* argument is less than or equal to zero.

[ERANGE] The *size* argument is greater than zero, but is smaller than the length of the pathname plus one.

[EACCES] Read or search permission was denied for a component of the pathname.

SEE ALSO

chdir(2), cc(1), errno.h(3), stddef.h(3)

NAME

getdiskbyname – get disk description by its name

SYNOPSIS

```
#include <disktab.h>

struct disktab *
getdiskbyname(name)
char *name;
```

DESCRIPTION

getdiskbyname takes a disk name (e.g. dkd-001) and returns a structure describing its geometry information and the standard disk partition tables. All information obtained from the *disktab*(5) file.

<disktab.h> has the following form:

```
/*      $CHeader: disktab.h 0.2 89/07/14 11:51:55 $     */
/*      Copyright 1984 Convex Computer Corp.*/

/*
 * Disk description table, see disktab(5)
 */
#define DISKTAB                 "/etc/disktab"

struct  disktab {
        char    *d_name;                /* drive name */
        char    *d_type;                /* drive type */
        int     d_secsize;              /* sector size in bytes */
        int     d_ntracks;              /* # tracks/cylinder */
        int     d_nsectors;             /* # sectors/track */
        int     d_ncylinders;           /* # cylinders */
        int     d_spares;               /* # spares/cyl (IDC only) */
        int     d_rpm;                  /* revolutions/minute */
        struct  partition {
                int     p_size;         /* #sectors in partition */
                int     p_bsize; /* block size in bytes */
                int     p_fsize; /* frag size in bytes */
        } d_partitions[8];
};

struct  disktab *getdiskbyname();
```

SEE ALSO

disktab(5)

DIAGNOSTICS

Returns NULL (0) if *name* is not found or an error occurs.

BUGS

This information should be obtained from the system for locally available disks (in particular, the disk partition tables). *getdiskbyname* returns a pointer to a static structure so it must be copied if it is to be saved.

NAME

getenv, setenv, unsetenv – manipulate environmental variables

SYNOPSIS

#include <stdlib.h>

char *getenv(const char *name);

int setenv(char *name, char *value, int overwrite);

void unsetenv(char *name);

DESCRIPTION

getenv searches the environment list (see *environ*(7)) for a string of the form *name=value* and returns a pointer to the string *value* if such a string is present, and 0 (NULL) if it is not.

setenv searches the environment list as *getenv* does; if the string *name* is not found, a string of the form *name=value* is added to the environment. If it is found, and *overwrite* is non-zero, its value is changed to *value*. *Setenv* returns 0 on success and -1 on failure, where failure is caused by an inability to allocate space for the environment.

unsetenv removes all occurrences of the string *name* from the environment. There is no library provision for completely removing the current environment. It is suggested that the following code be used to do so.

```
static char     *envinit[1];
extern char     **environ;
environ = envinit;
```

All of these routines permit, but do not require, a trailing equals ("=") sign on *name* or a leading equals sign on *value*.

COMPATIBILITY

When compiling or linking with the *-str* option of the **cc** command, different library routines are linked. In this mode, strict conformance to the ANSI C standard is specified. ANSI C reserves variable name *environ* for the C programmer. Another variable acceptable for use by the implementation is used instead.

SEE ALSO

environ(7), csh(1), sh(1), execve(2)

NAME

getfpmode – get current floating point mode

SYNOPSIS

#include <convex/fpmode.h>

int getfpmode(level)
int level;

DESCRIPTION

getfpmode returns the floating point mode that is in operation at the specified *level*. The integer value returned will be either FPMODE_IEEE or FPMODE_NATIVE, which are both defined in *<convex/fpmode.h>* and indicate the obvious setting of the IEEE bit in the PSW (on for IEEE, off for native).

If the argument, *level*, is 1, the mode for the current (calling) routine will be returned. For a *level* greater than 1, the *level*th ancestor routine will be checked and its mode returned.

SEE ALSO

setfpmode(3)

DIAGNOSTICS

A value of -1 is returned if an error is encountered.

NAME

getfsent, getfsspec, getfsfile, getfstype, setfsent, endfsent – get file system descriptor file entry

SYNOPSIS

```
#include <fstab.h>

struct fstab *getfsent()

struct fstab *getfsspec(spec)
char *spec;

struct fstab *getfsfile(file)
char *file;

struct fstab *getfstype(type)
char *type;

int setfsent()

int endfsent()
```

DESCRIPTION

These routines are included for compatibility with 4.2 BSD; they have been superseded by the *getmntent*(3) library routines.

getfsent, *getfsspec*, *getfstype*, and *getfsfile* each return a pointer to an object with the following structure containing the broken-out fields of a line in the file system description file, *<fstab.h>*.

```
struct fstab{
        char    *fs_spec;
        char    *fs_file;
        char    *fs_type;
        int     fs_freq;
        int     fs_passno;
};
```

The fields have meanings described in *fstab*(5).

getfsent reads the next line of the file, opening the file if necessary.

setfsent opens and rewinds the file.

endfsent closes the file.

getfsspec and *getfsfile* sequentially search from the beginning of the file until a matching special file name or file system file name is found, or until EOF is encountered. *getfstype* does likewise, matching on the file system type field.

NOTES

The above routines will not return file systems of type **nfs**. Those entries are ignored.

The *fs_type* field correspond to *mnt_opts*, not to *mnt_type*.

FILES

/etc/fstab

SEE ALSO

fstab(5), getmntent(3)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

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

NAME

getgrent, setgrent, endgrent, fgetgrent – get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent()

void setgrent()

void endgrent()

struct group *fgetgrent(f)
FILE *f;
```

DESCRIPTION

getgrent returns a pointer to an object with the following structure containing the broken-out fields of a line in the group file. Each line contains a "group" structure, defined in the *<grp.h>* header file.

```
struct group {
        char    *gr_name;
        char    *gr_passwd;
        int     gr_gid;
        char    **gr_mem;
};
```

The members of this structure are:

gr_name	The name of the group.
gr_passwd	The encrypted password of the group.
gr_gid	The numerical group ID.
gr_mem	A null-terminated array of pointers to the individual member names.

getgrent when first called 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. 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. Both *setgrent* and *endgrent* are void functions and therefore do not return a value.

fgetgrent returns a pointer to the next group structure in the stream *f*, which must refer to an open file in the same format as the group file */etc/group*. Yellow pages + or - entries are not expanded when using *fgetgrent*.

FILES

/etc/group
/etc/yp/*domainname*/group.byname
/etc/yp/*domainname*/group.bygid

SEE ALSO

getlogin(3), getgrgid(3), getpwent(3), group(5), ypserv(8), ypclnt(3N)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

A valid line in the */etc/group* file must have a non-NULL group ID field. Any line that does not will be ignored for security reasons.

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

These routines return only information available in the */etc/group* file. Depending upon site configuration, a user may or may not show up in the group entry for his primary group.

BUGS

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

NAME

getgrgid, getgrnam – group database access routines

SYNOPSIS

```
#include <grp.h>

struct group *getgrgid(gid)
int gid;

struct group *getgrnam(name)
char *name;
```

DESCRIPTION

The *getgrgid()* and *getgrnam()* routines both return pointers to an object of type *struct group* containing an entry from the group database with the matching *gid* or *name*. This structure, which is defined in **<grp.h>**, includes the members shown below:

Member Type	Member Name	Description
char *	gr_name	the name of the group.
gid_t	gr_gid	the numerical group id.
char **	gr_mem	a NULL terminated vector of pointers to the individual member names.

A **NULL** pointer is returned on error or if the requested entry is not found. The return values may point to static data that is overwritten by each call.

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

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

BACKWARD COMPATIBILITY

See getgrent(3)

FILES

/etc/group
/etc/yp/*domainname*/group.byname
/etc/yp/*domainname*/group.bygid

SEE ALSO

getlogin(3), getpwuid(3), getgrent(3), getpwent(3)

BUGS

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

NAME

gethostbyname, gethostbyaddr, gethostent, sethostent, endhostent, herror – get network host entry

SYNOPSIS

```
#include <netdb.h>

extern int h_errno;

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;

struct hostent *gethostent()

sethostent(stayopen)
int stayopen;

endhostent()

herror(string)
char *string;
```

DESCRIPTION

gethostbyname and *gethostbyaddr* each return a pointer to an object with the following structure describing an internet host referenced by name or by address, respectively. This structure contains either the information obtained from the name server, *named*(8), or broken-out fields from a line in */etc/hosts*. If the local name server is not running these routines do a lookup in */etc/hosts*. The file */etc/use_nameserver* determines which function to perform. If the file exists, these routines perform as described in "WITH THE NAMESERVER" below. If the file does not exist they perform as described in "WITHOUT THE NAMESERVER".

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
};
#define h_addr  h_addr_list[0]  /* address, for backward compatibility */
```

The members of this structure are:

h_name Official name of the host.

h_aliases A zero terminated array of alternate names for the host.

h_addrtype The type of address being returned; currently always AF_INET.

h_length The length, in bytes, of the address.

h_addr_list A zero terminated array of network addresses for the host. Host addresses are returned in network byte order.

h_addr The first address in h_addr_list; this is for backward compatiblity.

WITH THE NAMESERVER

When using the nameserver, *gethostbyname* will search for the named host in the current domain and its parents unless the name ends in a dot. If the name contains no dot, and if the environment variable "HOSTALIASES" contains the name of an alias file, the alias file will first be searched for an alias matching the input name. See *hostname*(7) for the domain search procedure and the alias file format.

Sethostent may be used to request the use of a connected TCP socket for queries. If the *stayopen* flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to *gethostbyname* or *gethostbyaddr*. Otherwise, queries are performed using UDP datagrams.

Endhostent closes the TCP connection.

WITHOUT THE NAMESERVER

gethostent reads the next line of the file, opening the file if necessary.

sethostent opens and rewinds the file. If the *stayopen* flag is non-zero, the host data base will not be closed after each call to *gethostent* (either directly, or indirectly through one of the other "gethost" calls).

endhostent closes the file.

gethostbyname and *gethostbyaddr* sequentially search from the beginning of the file until a matching host name or host address is found, or until EOF is encountered. Host addresses are supplied in network order.

DIAGNOSTICS

Error return status from *gethostbyname* and *gethostbyaddr* is indicated by return of a null pointer. The external integer *h_errno* may then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine *herror* can be used to print an error message describing the failure. If its argument *string* is non-NULL, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

h_errno can have the following values:

HOST_NOT_FOUND	No such host is known.
TRY_AGAIN	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
NO_RECOVERY	Some unexpected server failure was encountered. This is a non-recoverable error.
NO_DATA	The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

FILES

/etc/hosts
/etc/use_nameserver
/etc/yp/*domainname*/hosts.byname
/etc/yp/*domainname*/hosts.byaddr

SEE ALSO

resolver(3), hosts(5), hostname(7), named(8), ypserv(8)

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

NAME

getlogin – get login name

SYNOPSIS

char *getlogin()

DESCRIPTION

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

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

FILES

/etc/utmp

SEE ALSO

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

DIAGNOSTICS

Returns NULL (0) if name not found.

BUGS

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

NAME

setmntent, getmntent, addmntent, endmntent, hasmntopt – get file system descriptor file entry

SYNOPSIS

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

FILE *setmntent(filep, type)
char *filep;
char *type;

struct mntent *getmntent(filep)
FILE *filep;

int addmntent(filep, mnt)
FILE *filep;
struct mntent *mnt;

char *hasmntopt(mnt, opt)
struct mntent *mnt;
char *opt;

int endmntent(filep)
FILE *filep;
```

DESCRIPTION

These routines replace the *getfsent* routines for accessing the file system description file */etc/fstab*. They are also used to access the mounted file system description file */etc/mtab*.

setmntent opens a file system description file and returns a file pointer which can then be used with *getmntent*, *addmntent*, or *endmntent*. The *type* argument is the same as in *fopen*(3). *getmntent* reads the next line from *filep* and returns a pointer to an object with the following structure containing the broken-out fields of a line in the filesystem description file, *<mntent.h>*. The fields have meanings described in *fstab*(5).

```
struct mntent {
        char    *mnt_fsname;    /* file system name */
        char    *mnt_dir;       /* file system path prefix */
        char    *mnt_type;      /* 4.2, nfs, swap, or xx */
        char    *mnt_opts;      /* ro, quota, etc. */
        int     mnt_freq;       /* dump frequency, in days */
        int     mnt_passno;     /* pass number on parallel fsck */
};
```

addmntent adds the *mntent* structure *mnt* to the end of the open file *filep*. Note that *filep* has to be opened for writing if this is to work. *hasmntopt* scans the *mnt_opts* field of the *mntent* structure *mnt* for a substring that matches *opt*. It returns the address of the substring if a match is found, 0 otherwise. *endmntent* closes the file.

FILES

/etc/fstab
/etc/mtab

SEE ALSO

fstab(5), mtab(5), getfsent(3X)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

The returned *mntent* structure points to static information that is overwritten in each call.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetent()

struct netent *getnetbyname(name)
char *name;

struct netent *getnetbyaddr(net, type)
long net;

setnetent(stayopen)
int stayopen

endnetent()
```

DESCRIPTION

getnetent, *getnetbyname*, and *getnetbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, */etc/networks*.

```
struct  netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net number type */
        long    n_net;          /* net number */
};
```

The members of this structure are:

n_name The official name of the network.

n_aliases A zero terminated list of alternate names for the network.

n_addrtype The type of the network number returned; currently only AF_INET.

n_net The network number. Network numbers are returned in machine byte order.

getnetent reads the next line of the file, opening the file if necessary.

setnetent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getnetent* (either directly, or indirectly through one of the other "getnet" calls).

endnetent closes the file.

getnetbyname and *getnetbyaddr* sequentially search from the beginning of the file until a matching net name or net address is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks
/etc/yp/*domainname*/networks.byaddr
/etc/yp/*domainname*/networks.byname

SEE ALSO

networks(5), ypserv(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

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

Only Internet network numbers are currently understood.

NAME

getopt – get option letter from argv

SYNOPSIS

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

extern char *optarg;
extern int optind;

DESCRIPTION

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

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

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

DIAGNOSTICS

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

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;
                        break;
                case '?':
                default:
                        errflg++;
                        break;
                }
        if (errflg) {
                fprintf(stderr, "Usage: ...");
                exit(2);
        }
        for (; optind < argc; optind++) {
                .
                .
                .
        }
        .
        .
        .
}
```

HISTORY

Written by Henry Spencer, working from a Bell Labs manual page. Modified by Keith Bostic to behave more like the System V version.

BUGS

It is not obvious how '–' standing alone should be treated; this version treats it as a non-option argument, which is not always right.

Option arguments are allowed to begin with '–'; this is reasonable but reduces the amount of error checking possible.

getopt is quite flexible but the obvious price must be paid: there is much it could do that it doesn't, like checking mutually exclusive options, checking type of option arguments, etc.

NAME

getpass – read a password

SYNOPSIS

char *getpass(prompt)
char *prompt;

DESCRIPTION

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

FILES

/dev/tty

SEE ALSO

crypt(3)

BUGS

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

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

#include <netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

setprotoent(stayopen)
int stayopen

endprotoent()

DESCRIPTION

getprotoent, *getprotobyname*, and *getprotobynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, */etc/protocols*.

```
struct  protoent {
        char    *p_name;        /* official name of protocol */
        char    **p_aliases;    /* alias list */
        long    p_proto;        /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.

p_aliases A zero terminated list of alternate names for the protocol.

p_proto The protocol number.

getprotoent reads the next line of the file, opening the file if necessary.

setprotoent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getprotoent* (either directly, or indirectly through one of the other "getproto" calls).

endprotoent closes the file.

getprotobyname and *getprotobynumber* sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

FILES

/etc/protocols
/etc/yp/*domainname*/protocols.byname
/etc/yp/*domainname*/protocols.bynumber

SEE ALSO

getservent(3n), protocols(5), ypserv(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

NAME

getpty – get a pseudo-teletype device name

SYNOPSIS

char *getpty()

DESCRIPTION

Getpty returns a pointer to the name of a pseudo-teletype device. It is preferable to use *getpty*, as it functions regardless of the number of pseudo-teletypes configured into the system, and independently of the arbitrary pseudo-teletype naming convention. The name *getpty* returns is an advisory name only, and does not guarantee that a call to *open*(2) with this name will succeed. A typical code sequence to obtain the master and slave file descriptors for a pseudo-teletype is as follows:

```
/*
 * open master side.
 */
for (;;) {
        if ((pty_name = getpty( )) == NULL)
            fatal ("Can't get a pty");

        if ((master_fd = open (pty_name, 2)) >= 0) {
            /*
             * open slave side.
             */
            pty_name[5] = 't';    /* /dev/ptyXY ==> /dev/ttyXY */
            if ((slave_fd = open (pty_name, 2)) >= 0)
                    break;
            else
                    close (master_fd);
        }
}
```

RETURN VALUE

If the call succeeds, a character pointer is returned which references the name of a valid pseudo-teletype device. If the call fails, NULL is returned, which indicates that there are no more pseudo-teletype devices.

SEE ALSO

open(2)

BUGS

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

NAME

getpw – get name from uid

SYNOPSIS

getpw(uid, buf)
char *buf;

DESCRIPTION

Getpw is obsoleted by getpwuid(3).

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

FILES

/etc/passwd

SEE ALSO

getpwent(3), passwd(5)

DIAGNOSTICS

Non-zero return on error.

NAME

getpwent, setpwent, endpwent, fgetpwent – get password file entry

SYNOPSIS

#include <pwd.h>

struct passwd *getpwent(void)

void setpwent(void)

void endpwent(void)

struct passwd *fgetpwent(FILE *f)

DESCRIPTION

getpwent() returns a pointer to an object with the following structure containing the broken-out fields of a line in the password file. Each line in the file contains a "passwd" structure, declared in the *<pwd.h>* header file, which contains the following members:

char *pw_name	login name
char *pw_passwd	encrypted password
uid_t pw_uid	unique numeric user ID.
gid_t pw_gid	numeric group to which the user belongs.
char *pw_gecos	typically used as a comment
char *pw_dir	home directory.
char *pw_shell	login shell.

The fields *pw_quota* and *pw_comment* are unused; the others have meanings more fully described in *passwd*(5). When first called, *getpwent* returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. If an end-of-file or an error is encountered on reading, *getpwent* and *fgetpwent* 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.

fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of the password file */etc/passwd*. Yellow pages + or - entries are not expanded when using *fgetpwent*.

FILES

/etc/passwd
/etc/passwd.dir
/etc/passwd.pag
/etc/yp/*domainname*/passwd.byname
/etc/yp/*domainname*/passwd.byuid

SEE ALSO

getpwuid(3), getlogin(3), getgrent(3), passwd(5), ypserv(8), ypclnt(3N)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

A valid line in the */etc/passwd* file must have non-NULL user ID and group ID fields. Any line that does not will be ignored for security reasons.

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

BUGS

The return value points to static information that is overwritten on each call. Furthermore, the structure returned *contains* pointers to static information, so copying the returned structure requires a member–wise copy using (e.g.) *strcpy(3)*.

NAME

getpwrestent, getpwrestuid, getpwrestnam, setpwrestent, endpwrestent, fgetpwrestent – get entry from password restrictions file

SYNOPSIS

```
#include <pwrest.h>

struct pwrestrict *getpwrestent()

struct pwrestrict *getpwrestuid(uid)
int uid;

struct pwrestrict *getpwrestnam(name)
char *name;

int setpwrestent()

int endpwrestent()

struct pwrestrict *fgetpwrestent(f)
FILE *f;
```

DESCRIPTION

getpwrestent, *getpwrestuid*, and *getpwrestnam* each return a pointer to an object with the following structure containing the broken-out fields of a line in the password restrictions file. Each line in the file contains a "pwrestrict" structure, declared in the *<pwrest.h>* header file:

```
/*      $CHeader: pwrest.h 0.3 89/06/25 16:45:23 $      */
/*      Copyright 1986 Convex Computer Corp.*/

#ifdef MAX_PWLEN
#define MAX_PWRLENMAX_PWLEN    /* max length of a pwrestrict file entry */
#else
#define MAX_PWRLEN1024         /* max length of a pwrestrict file entry */
#endif

struct          pwrestrict {    /* see getpwrestent(3) */
        char    *pwr_name;
        char    *pwr_passwd;
        char    *pwr_age;
        char    pwr_restrict;
        char    *pwr_usrrest;   /* unused */
        int     pwr_uid;
};

struct pwrestrict *getpwrestent(), *getpwrestuid(),
                  *getpwrestnam(), *fgetpwrestent();
```

This structure is declared in *<pwrest.h>* so it is not necessary to redeclare it.

The fields shown above have meanings described in *pwrestrict(5)*. When first called, *getpwrestent* returns a pointer to the first pwrestrict structure in the file; thereafter, it returns a pointer to the next pwrestrict structure in the file; so successive calls can be used to search the entire file. *getpwrestuid* searches the dbm(3) style password database if /etc/pwrestrict.dir and /etc/pwrestrict.pag are accessible, otherwise it 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. You must pass the -ldbm option to the loader to use *getpwrestuid*. *getpwrestnam* searches the dbm(3) style password database if /etc/pwrestrict.dir and /etc/pwrestrict.pag are accessible, otherwise it 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. You must pass

the -ldbm option to the loader to use *getpwrestnam*. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

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

fgetpwrestent returns a pointer to the next pwrestrict structure in the stream *f*, which matches the format of the password file */etc/pwrestrict*. Yellow pages + or - entries are not expanded when using *fgetpwrestent*.

FILES

/etc/pwrestrict
/etc/pwrestrict.dir
/etc/pwrestrict.pag
/etc/yp/*domainname*/pwrestrict.byname
/etc/yp/*domainname*/pwrestrict.byuid

SEE ALSO

getlogin(3), getgrent(3), getpwent(3), passwd(5), pwrestrict(5), ypserv(8), ypclnt(3N)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

A valid line in the */etc/pwrestrict* file must have a non-NULL user ID field. Any line that does not will be ignored for security reasons.

The above routines use *<stdio.h>* and Yellow Pages, 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

gets, fgets – get a string from a stream

SYNOPSIS

#include <stdio.h>

char *gets(char *s);

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

DESCRIPTION

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

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

SEE ALSO

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

DIAGNOSTICS

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

BUGS

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

NAME

getpwuid, getpwnam – user database access

SYNOPSIS

#include <pwd.h>

struct passwd *getpwuid(uid_t)

struct passwd *getpwnam(const char *)

DESCRIPTION

The *getpwuid()* and *getpwnam()* functions both return a pointer to an object of type *struct passwd* containing an entry from the user database with a matching *uid* or *name*. This structure, which is defined in **<pwd.h>**, includes the members shown below:

Member Type	Member Name	Description
char *	pw_name	the users login name.
uid_t	pw_uid	the users id number.
gid_t	pw_gid	the users group id number.
char *	pw_dir	the users initial working directory (home directory.)
char *	pw_shell	the users initial program (shell.)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNINGS

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

The implementation of the *cuserid()* function uses the *getpwnam()* function; thus the results of a user's call to either routine will be overwritten by a subsequent call to the other routine.

BACKWARDS COMPATIBILITY

See getpwent(3)

FILES

/etc/passwd
/etc/passwd.dir
/etc/passwd.pag
/etc/yp/*domainname*/passwd.byname
/etc/yp/*domainname*/passwd.byuid

SEE ALSO

getlogin(3), cuserid(3), getgrgid(3), getpwent(3), getgrent(3)

BUGS

The return value points to static information that is overwritten on each call.

NAME

getservent, getservbyport, getservbyname, setservent, endservent – get service entry

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent()

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port; char *proto;

setservent(stayopen)
int stayopen;

endservent()
```

DESCRIPTION

getservent, *getservbyname*, and *getservbyport* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, */etc/services*.

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        long    s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name The official name of the service.

s_aliases A zero terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto The name of the protocol to use when contacting the service.

getservent reads the next line of the file, opening the file if necessary.

setservent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getservent* (either directly, or indirectly through one of the other "getserv" calls).

endservent closes the file.

getservbyname and *getservbyport* sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

FILES

/etc/services
/usr/etc/yp/*domainname*/services
/usr/etc/yp/*domainname*/services.byname

SEE ALSO

getprotoent(3N), services(5), ypserv(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Expecting port numbers to fit in a 32-bit quantity is probably naive.

The Yellow Pages map file /usr/etc/yp/*domain*/services.byname is really keyed by port number/protocol rather than service name/protocol as the file name suggests.

NAME

getttyent, getttynam, setttyent, endttyent – get ttys file entry

SYNOPSIS

```
#include <ttyent.h>

struct ttyent *getttyent()

struct ttyent *getttynam(name)
char *name;

setttyent()

endttyent()
```

DESCRIPTION

Getttyent, and *getttynam* each return a pointer to an object with the following structure containing the broken-out fields of a line from the tty description file.

```
/*      $CHeader: ttyent.h 0.1 89/10/04 16:10:51 $      */
/*      Copyright 1989 Convex Computer Corp.*/
/* $Cheader : $
 */
struct  ttyent { /* see getttyent(3) */
        char    *ty_name;       /* terminal device name */
        char    *ty_getty;      /* command to execute, usually getty */
        char    *ty_type;       /* terminal type for termcap (3X) */
        int     ty_status;      /* status flags (see below for defines) */
        char    *ty_window;     /* command to start up window manager */
        char    *ty_comment;    /* usually the location of the terminal */
};

#define TTY_ON              0x1     /* enable logins (startup getty) */
#define TTY_SECURE  0x2     /* allow root to login */
#define TTY_DIALUP  0x4     /* check dialup password */
#define TTY_RESTRICTED  0x8     /* check if there are any access or deny */
#define TTY_UUCP    0x10    /* check uucp only */
extern struct ttyent *getttyent();
extern struct ttyent *getttynam();
```

ty_name is the name of the character-special file in the directory "/dev". For various reasons, it must reside in the directory "/dev".

ty_getty is the command (usually *getty*(8)) which is invoked by *init* to initialize tty line characteristics. In fact, any arbitrary command can be used; a typical use is to initiate a terminal emulator in a window system.

ty_type is the name of the default terminal type connected to this tty line. This is typically a name from the *termcap*(5) data base. The environment variable 'TERM' is initialized with this name by *getty*(8) or *login*(1).

ty_status is a mask of bit fields which indicate various actions to be allowed on this tty line. The following is a description of each flag.

TTY_ON	Enables logins (i.e., *init*(8) will start the specified "getty" command on this entry).
TTY_SECURE	Allows root to login on this terminal. Note that 'TTY_ON' must be included for this to be useful on hard-wired terminal lines.
TTY_DIALUP	Indicates that this a dialup terminal and a dialup pass-

word is required for login. Note that 'TTY_ON' must be included for this to be useful.

TTY_RESTRICTED Indicates that this terminal is restricted for login by some users. The restriction list is specified in the */etc/ttys* file. Note that 'TTY_ON' must be included for this to be useful.

ty_window is the command to execute for a window system associated with the line. The window system will be started before the command specified in the *ty_getty* entry is executed. If none is specified, this will be null.

ty_comment is the trailing comment field, if any; a leading delimiter and white space will be removed.

Getttyent reads the next line from the ttys file, opening the file if necessary; *setttyent* rewinds the file; *endttyent* closes it.

Getttynam searches from the beginning of the file until a matching *name* is found (or until EOF is encountered).

FILES

/etc/ttys

SEE ALSO

login(1), ttyslot(3), ttys(5), gettytab(5), termcap(5), getty(8), init(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

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

NAME

getusershell, setusershell, endusershell – get legal user shells

SYNOPSIS

char *getusershell()

setusershell()

endusershell()

DESCRIPTION

Getusershell returns a pointer to a legal user shell as defined by the system manager in the file */etc/shells*. Successive calls to *getusershell* return successive entries from /etc/shells. NULL is returned on calls made after the last entry has been returned. If */etc/shells* does not exist, *getusershell* returns in succession the two standard system shells */bin/sh* and */bin/csh*.

Getusershell reads the next line (opening the file if necessary); *setusershell* rewinds the file; *endusershell* closes it.

FILES

/etc/shells

SEE ALSO

shells(5)

DIAGNOSTICS

The routine *getusershell* returns a null pointer (0) on EOF or error.

BUGS

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

NAME

getwd – get current working directory pathname

SYNOPSIS

char *getwd(pathname)
char *pathname;

DESCRIPTION

Getwd copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result.

LIMITATIONS

Maximum pathname length is MAXPATHLEN characters (1024).

DIAGNOSTICS

Getwd returns zero and places a message in *pathname* if an error occurs.

BUGS

Getwd may fail to return to the current directory if an error occurs.

NAME

cabs, cabsf, hypot, hypotf, scabs, shypot – Euclidean distance

SYNOPSIS

```
#include <math.h>

double cabs(z)
struct { double real, imag;} z;

double cabsf(z)
struct { float real, imag;} z;

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

float hypotf(x, y)
double x, y;

float scabs(z)
struct { float real, imag;} z;

float shypot(x, y)
double x, y;
```

DESCRIPTION

Hypot, hypotf and *shypot* return

sqrt(x*x + y*y).

Cabs, cabsf and *scabs* return

sqrt(real*real + imag*imag).

The functions take precautions against unwarranted overflows.

BACKWARD COMPATIBLE

cabs, cabsf, hypot, shypot and *scabs* are only available in backward-compatible and extended modes of CONVEX C.

hypotf is only in extended mode of CONVEX C.

SEE ALSO

exp(3M) for *sqrt*

NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(cp)
char *cp;

unsigned long inet_network(cp)
char *cp;

char *inet_ntoa(in)
struct in_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

int inet_lnaof(in)
struct in_addr in;

int inet_netof(in)
struct in_addr in;
```

DESCRIPTION

The routines *inet_addr* and *inet_network* each interpret character strings representing numbers expressed in the Internet standard . notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine *inet_ntoa* takes an Internet address and returns an ASCII string representing the address in . notation. The routine *inet_makeaddr* takes an Internet network number and a local network address and constructs an Internet address from it. The routines *inet_netof* and *inet_lnaof* break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet address are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the . notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When 4 parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the 4 bytes of an Internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost 2 bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as **128.net.host**.

When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the rightmost 3 bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as **net.host**.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a . notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e. a leading **0x** or **0X** implies hexadecimal; otherwise, a leading **0** implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO

gethostbyname(3N), getnetent(3N), hosts(5), networks(5),

DIAGNOSTICS

The value **-1** is returned by *inet_addr* and *inet_network* for malformed requests.

BUGS

The problem of host-byte ordering versus network-byte ordering is confusing. A simple way is needed to specify Class C network addresses in a manner similar to that for Class A and Class B. The string returned by *inet_ntoa* resides in a static memory area.

NOTES

Inet is an optional product; for more information, contact your CONVEX sales representative.

NAME

initgroups – initialize group access list

SYNOPSIS

initgroups(name, basegid)
char *name;
int basegid;

DESCRIPTION

Initgroups reads through the group file and sets up, using the *setgroups*(2) call, the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file.

FILES

/etc/group

SEE ALSO

setgroups(2)

DIAGNOSTICS

Initgroups returns –1 if it was not invoked by the super-user.

BUGS

Initgroups uses the routines based on *getgrent*(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to *initgroups*.

No one seems to keep /etc/group up to date.

NAME

insque, remque – insert/remove element from a queue

SYNOPSIS

```
struct qelem {
        struct  qelem *q_forw;
        struct  qelem *q_back;
        char    q_data[];
};

insque(elem, pred)
struct qelem *elem, *pred;

remque(elem)
struct qelem *elem;
```

DESCRIPTION

Insque and *remque* manipulate queues built from doubly linked lists. Each element in the queue must in the form of "struct qelem". *Insque* inserts *elem* in a queue immediately after *pred*; *remque* removes an entry *elem* from a queue.

NAME

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

SYNOPSIS

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

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

double y0(x)
double x;

double y1(x)
double x;

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

DESCRIPTION

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

DIAGNOSTICS

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

NAME

limits.h – header file contain numerical limits

SYNOPSIS

#include <limits.h>

DESCRIPTION

limits.h contains information on the numerical limits of the system.

CHAR_BIT is the number of bits in a byte.

SCHAR_MIN, SCHAR_MAX, UCHAR_MIN, UCHAR_MAX, CHAR_MIN and **CHAR_MAX** are the minimum and maximun value of signed characters, unsigned characters and plain characters, respectively.

SHRT_MIN, SHRT_MAX, INT_MIN, INT_MAX, LONG_MIN, LONG_MAX, LONG_LONG_MIN, and **LONG_LONG_MAX** are the minimum and maximum values that can be acquired by objects of type **short int, int, long int,** and **long long int** , respectively. **USHRT_MAX, UINT_MAX, ULONG_MAX** and **ULONG_LONG_MAX** are the maximum values for the unsigned integral types.

NAME

lockf – advisory record locking on files

SYNOPSIS

```
#include <fcntl.h>

#define     F_ULOCK     0     /* Unlock a previously locked section */
#define     F_LOCK      1     /* Lock a section for exclusive use */
#define     F_TLOCK     2     /* Test and lock a section (non-blocking) */
#define     F_TEST      3     /* Test section for other process' locks */

lockf(fd, cmd, size)
int fd, cmd;
long size;
```

DESCRIPTION

lockf may be used to test, apply, or remove an *advisory* record lock on the file associated with the open descriptor *fd*. (See *fcntl*(2) for more information about advisory record locking.)

A lock is obtained by specifying a *cmd* parameter of F_LOCK or F_TLOCK. To unlock an existing lock, the F_ULOCK *cmd* is used. F_TEST is used to detect if a lock by another process is present on the specified segment.

F_LOCK and F_TLOCK requests differ only by the action taken if the lock may not be immediately granted. F_TLOCK will cause the function to return a –1 and set *errno* to EAGAIN if the section is already locked by another process. F_LOCK will cause the process to sleep until the lock may be granted or a signal is caught.

size is the number of contiguous bytes to be locked or unlocked. The lock starts at the current file offset in the file and extends forward for a positive *size* or backward for a negative *size* (preceding but not including the current offset). A segment need not be allocated to the file in order to be locked; however, a segment may not extend to a negative offset relative to the beginning of the file. If *size* is zero, the lock will extend from the current offset through the end-of-file. If such a lock starts at offset 0, then the entire file will be locked (regardless of future file extensions).

NOTES

The descriptor *fd* must have been opened with O_WRONLY or O_RDWR permission in order to establish locks with this function call.

All locks associated with a file for a given process are removed when the file is closed or the process terminates. Locks are not inherited by the child process in a *fork*(2) system call.

RETURN VALUE

Zero is returned on success, –1 on error, with an error code stored in *errno*.

ERRORS

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

[EBADF]	*fd* is not a valid open descriptor.
[EBADF]	*cmd* is F_LOCK or F_TLOCK and the process does not have write permission on the file.
[EAGAIN]	*cmd* is F_TLOCK or F_TEST and the section is already locked by another process.
[EINTR]	*cmd* is F_LOCK and a signal interrupted the process while it was waiting for the lock to be granted.
[EDEADLK]	*cmd* is F_LOCK, F_TLOCK, or F_ULOCK and there are no more file lock entries available.
[EINVAL]	The specified *cmd* is not one of the supported ones listed above.

[EACCES] The *cmd* is F_TEST or F_TLOCK and the specified region (or part of it) is exclusively locked by another process.

SEE ALSO

fcntl(2), lockd(8C)

BUGS

File locks obtained through the *lockf* mechanism do not interact in any way with those acquired via *flock*(2). They do, however, work correctly with the locks claimed by *fcntl*(2).

NAME

malloc, free, realloc, calloc, cfree, alloca – memory allocator

SYNOPSIS

#include <stdlib.h>

void *malloc(size_t size);

void free(void *ptr);

void *realloc(void *ptr, size_t size);

void *calloc(size_t nelem, size_t elsize);

void cfree(void *ptr, size_t nelem, size_t elsize);

char *alloca(int size);

DESCRIPTION

malloc and *free* provide a general-purpose memory allocation package. *malloc* returns a pointer to a block of at least *size* bytes beginning on a long-word (64 bits) boundary.

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

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

malloc maintains multiple lists of free blocks according to size, allocating space from the appropriate list. 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.

In order to be compatible with older versions, *realloc* also works if *ptr* points to a block freed since the last call of *malloc*, *realloc* or *calloc*; sequences of *free*, *malloc* and *realloc* were previously used to attempt storage compaction. This procedure is no longer recommended.

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros. *free* frees space previously allocated by *calloc*.

cfree also frees space previously allocated by *calloc*, but it use is consider obsolete.

alloca allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object. If the space is of *pagesize* or larger, the memory returned will be page-aligned.

BACKWARD COMPATIBILITY

cfree is available only in the backward-compatible and extended modes of CONVEX C.

SEE ALSO

brk(2), getpagesize(2), pagesize(1)

DIAGNOSTICS

malloc, *realloc* and *calloc* return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. *malloc* may be recompiled to check the arena very stringently on every transaction; those sites with a source code license may check the source code to see how this can be done.

BUGS

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

The current implementation of *malloc* does not always fail gracefully when system memory limits are approached. It may fail to allocate memory when larger free blocks could be broken up, or when limits are exceeded because the size is rounded up. It is optimized for sizes that are powers of two.

alloca is machine dependent; its use is discouraged.

free does not return memory allocated by *malloc* back to the system, for use by other processes.

NAME

mblen, mbtowc, wctomb, mbstowcs, wcstombs – multibyte and wide character functions

SYNOPSIS

#include <stdlib.h>

int mblen(const char *s, size_t n);

int mbtowc(wchar_t *pwc, const char *s, size_t n);

int wctomb(char *s, wchar_t wchar);

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);

size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);

DESCRIPTION

These functions are of limited usefulness because CONVEX C supports only the "C" locale. If **s* is **NULL** in any of the following functions, 0 is returned indicating that no state-dependent encodings exist. The conversion between multibyte character strings and wide characters are trivial because no special encodings are supplied by CONVEX.

mblen returns the number of bytes, up to *n*, of the multibyte character pointed to by *s*. This is always 1 unless *n* = 0 or **s* == **NULL**.

mbtowc converts a multibyte string to a wide character. Because all multibyte characters are length 1, the first character pointed to by **s* is copied to **pwc* and 1 is returned. However, if *n* = 0, no conversion is performed and -1 is returned.

wctomb converts a wide character to a multibyte character code stored in the object *wchar* into a multibyte character whose base address is *s*. 1 is returned because all multibyte characters are of length 1.

mbstowcs copies *n* bytes, or until **NULL** is encountered, of the multibyte string pointed to by **s* to the wide character string pointed to by **pwcs*. Returns the number of multibyte characters copied.

wcstombs copies an array of *n* or fewer multibyte character encodings into a multibyte character string. Returns one less than the number of characters copied.

NOTES

These functions are intended for use on systems implementing locales with extended character sets and are provided for standard conformance.

NAME

mkfifo – make a FIFO special file

SYNOPSIS

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

int mkfifo(path, mode)
char *path;
mode_t mode;
```

DESCRIPTION

mkfifo() creates a new FIFO special file with name *path*. The mode of the new FIFO special file is initialized from *mode*.

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

RETURN VALUE

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

ERRORS

mkfifo() will fail and no FIFO special file will be created if:

[EACCES]	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the FIFO special file to be created.
[EEXIST]	The named file exists.
[EINVAL]	The *path* argument contains a byte with the high-order bit set.
[ENAMETOOLONG]	The length of *path* exceeds PATH_MAX for the file system.
[ENAMETOOLONG]	The length of a pathname component exceeds NAME_MAX and the file system enforces _POSIX_NO_TRUNC.
[ENOENT]	A component of the path prefix does not exist.
[ENOENT]	The *path* argument points to the empty string.
[ENOSPC]	The file system does not contain enough space to expand the parent directory.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while writing to the file system.

SEE ALSO

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

NAME

mktemp, mkstemp – make a unique file name

SYNOPSIS

char *mktemp(template)
char *template;

mkstemp(template)
char *template;

DESCRIPTION

mktemp creates a unique file name, typically in a temporary filesystem, by replacing *template* with a unique file name, and returns the address of the template. The template should contain a file name with six trailing X's, which are replaced with the current process id and a unique letter. *mktemp* does not create a file, rather it checks for the existence of a file with the specified name. It is the user's responsibility to create a file with the name supplied by *mktemp*.

mkstemp makes the same replacement to the template but returns a file descriptor for the template file open for reading and writing. *mkstemp* avoids the race between testing whether the file exists and opening it for use.

SEE ALSO

getpid(2), open(2)

DIAGNOSTICS

mkstemp returns an open file descriptor upon success. It returns -1 if no suitable file could be created.

NAME

monitor, monstartup, moncontrol – prepare execution profile

SYNOPSIS

#include <mon.h>

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

monstartup(*lowpc, highpc***)**
int (*lowpc)(), (*highpc)();

moncontrol(*mode***)**

DESCRIPTION

There are two different forms of monitoring available: An executable program created by:

cc –p . . .

automatically includes calls for the *prof*(1) monitor and includes an initial call to its start-up routine *monstartup* with default parameters; *monitor* need not be called explicitly except to gain fine control over profile buffer allocation. An executable program created by:

cc –pg . . .

automatically includes calls for the *gprof*(1) monitor.

Monstartup is a high-level interface to *lprofil*(2). *Lowpc* and *highpc* specify the address range that is to be sampled; the lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *Monstartup* allocates space using *sbrk*(2) and passes it to *monitor* (see below) to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. Only calls of functions compiled with the profiling option –p of *cc*(1) are recorded.

To profile the entire program, it is sufficient to use

```
extern etext();
. . .
monstartup(0x80001000, etext);
```

Etext lies just above all the program text, see *end*(3).

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

monitor(0);

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

Moncontrol is used to selectively control profiling within a program. This works with either *prof*(1) or *gprof*(1) type profiling. When the program starts, profiling begins. To stop the collection of histogram ticks and call counts, use *moncontrol*(0); to resume the collection of histogram ticks and call counts, use *moncontrol*(1). This allows the cost of particular operations to be measured. Note that an output file will be produced upon program exit regardless of the state of *moncontrol*.

Monitor is a low-level interface to *lprofil*(2). *Lowpc* and *highpc* are two address in the executable code and specify the range in which profiling si to be done; *buffer* is the address of a (user-supplied) memory buffer which must be large enough for a header of type *struct phdr* defined in *(mon.h)* , an array of *nfunc struct cnt* defined in *(mon.h)* in which the profiler will count calls to routines compiled with –p or –pg, and an array of *bufsize WORD* defined in *(mon.h)*. At most *nfunc* call counts can be kept. For the results to be significant, especially where there are small,

heavily used routines, it is suggested that the *WORD* part of the buffer be no more than a few times smaller than the range of locations sampled. *Monitor* divides the buffer into space to record the histogram of program counter samples over the range *lowpc* to *highpc*, and space to record call counts of functions compiled with the **-p** option to *cc*(1).

To profile the entire program, it is sufficient to use

```
extern etext();
. . .
monitor(0x80001000, etext, buf, bufsize, nfunc);
```

FILES

mon.out

SEE ALSO

cc(1), prof(1)(optional product), gprof(1)(optional product), lprofil(2), brk(2)

NAME

mset, mclear – shared memory synchronization primitives

SYNOPSIS

```
#include      <sys/types.h>
mset(sem,wait)
semaphore *sem;
int wait;
mclear(sem)
semaphore *sem;
```

DESCRIPTION

These routines provide interprocess synchronization for shared memory.

mset indivisibly tests and sets a lock in semaphore *sem*. If the previous value of the lock was zero, the calling process has acquired the lock and *mset* returns true (**1**) immediately. If the previous value of the lock was non-zero, and the *wait* flag was set, *mset* will relinquish the processor until some other process releases the lock, at which time it again trys to acquire the lock. If the previous value of the lock was non-zero, and the *wait* flag is zero, *mset* returns failure (**0**).

mclear indivisibly clears the lock in *sem*, and awakens all other processes waiting on the semaphore.

sem is a pointer to a machine dependent semaphore structure, defined in *sys/types.h*. On the Convex Computer, the structure is:

```
struct      semaphore   {     /* machine dependent */
      char  lock;       /* the tas lock */
      char  awakened;   /* waiters have been awakened */
      };
#typedef struct semaphore semaphore;
```

sem must be in a shared memory region, with at least **PROT_READ** and **PROT_WRITE**. The **MAP_HASSEMAPHORE** flag must have been set when the *mmap* call was executed. The region may be mapped in any type– MAP_FILE, MAP_ANON, or MAP_DEVICE.

RETURN VALUE & ERRORS

mset returns **1** if the lock was acquired; **0** if not. A **-1** is returned if a system call internal to *mset* returned an error. *mclear* returns a positive value or zero if successful; a **-1** is returned if a system call internal to *mclear* returned an error. Note that *mset* is interruptible, and may return **-1** with *errno* set to EINTR if a signal is received during the sleep.

SEE ALSO

mmap(2), msleep(2), tas(3)

NAME

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr – data base subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
    char *dptr;
    int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
    char *file;
    int flags, mode;

void dbm_close(db)
    DBM *db;

datum dbm_fetch(db, key)
    DBM *db;
    datum key;

int dbm_store(db, key, content, flags)
    DBM *db;
    datum key, content;
    int flags;

int dbm_delete(db, key)
    DBM *db;
    datum key;

datum dbm_firstkey(db)
    DBM *db;

datum dbm_nextkey(db)
    DBM *db;

int dbm_error(db)
    DBM *db;

int dbm_clearerr(db)
    DBM *db;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. This package replaces the earlier *dbm*(3x) library, which managed only a single database.

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

Before a database can be accessed, it must be opened by *dbm_open*. This will open and/or create the files *file*.**dir** and *file*.**pag** depending on the flags parameter (see *open*(2)).

Once open, the data stored under a key is accessed by *dbm_fetch* and data is placed under a key by *dbm_store*. The *flags* field can be either **DBM_INSERT** or **DBM_REPLACE. DBM_INSERT** will only insert new entries into the database and will not change an existing entry with the same key. **DBM_REPLACE** will replace an existing entry if it has the same key. A key (and its associated contents) is deleted by *dbm_delete*. A linear pass through all keys in a

NAME

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr – data base subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
    char *dptr;
    int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
    char *file;
    int flags, mode;

void dbm_close(db)
    DBM *db;

datum dbm_fetch(db, key)
    DBM *db;
    datum key;

int dbm_store(db, key, content, flags)
    DBM *db;
    datum key, content;
    int flags;

int dbm_delete(db, key)
    DBM *db;
    datum key;

datum dbm_firstkey(db)
    DBM *db;

datum dbm_nextkey(db)
    DBM *db;

int dbm_error(db)
    DBM *db;

int dbm_clearerr(db)
    DBM *db;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. This package replaces the earlier *dbm*(3x) library, which managed only a single database.

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

Before a database can be accessed, it must be opened by *dbm_open*. This will open and/or create the files *file*.**dir** and *file*.**pag** depending on the flags parameter (see *open*(2)).

Once open, the data stored under a key is accessed by *dbm_fetch* and data is placed under a key by *dbm_store*. The *flags* field can be either **DBM_INSERT** or **DBM_REPLACE**. **DBM_INSERT** will only insert new entries into the database and will not change an existing entry with the same key. **DBM_REPLACE** will replace an existing entry if it has the same key. A key (and its associated contents) is deleted by *dbm_delete*. A linear pass through all keys in a

database may be made, in an (apparently) random order, by use of *dbm_firstkey* and *dbm_nextkey*. *Dbm_firstkey* will return the first key in the database. *Dbm_nextkey* will return the next key in the database. This code will traverse the data base:

for (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))

Dbm_error returns non-zero when an error has occurred reading or writing the database. *Dbm_clearerr* resets the error condition on the named database.

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*. If *dbm_store* called with a *flags* value of **DBM_INSERT** finds an existing entry with the same key it returns 1.

BUGS

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

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls. This storage is not necessarily aligned; stored "longs", for example, should be copied to a properly aligned block of memory before being accessed.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 4096 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Dbm_store* will return an error in the event that a disk block fills with inseparable data.

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

The order of keys presented by *dbm_firstkey* and *dbm_nextkey* depends on a hashing function, not on anything interesting.

SEE ALSO

dbm(3X)

NAME

nfabort – dump core and log it in a notesfile

SYNOPSIS

nfabort (notesfile, message, title, cname, exitcode)
char *notesfile, *message, *title, *cname, exitcode

cc ... –lnfcom

DESCRIPTION

Nfabort provides user programs with a convenient way to generate a core image, move it to a save place, and log the action in a notesfile.

The *notesfile* parameter specifies the notesfile to receive a copy of the string *message* with a line appended detailing the final resting place of the core image. The text is inserted into the notesfile as a base note with *title* taken from the parameter list. *Cname* is the prefix of the pathname of where to place the core image. This is suffixed with ".integer" to yield the full pathname. The integer is generated from the pid of the current process.

After generating and saving the core image and placing the message in the notesfile, *nfabort* terminates the current process with the exit code specified by the *exitcode* parameter.

Nfabort calls *nfcomment* to insert the message into the notesfile.

BUGS

Certain conditions, such as running out of memory, will cause *nfabort* to fail.

Nfabort will fail to log the message if it can't fork a child process.

The final resting place of the core image will not be logged if *nfabort* can't allocate memory for temporary strings.

FILES

/usr/lib/libnfcom.a -lnfcom library

SEE ALSO

malloc(3), nfcomment(3), nfpipe(1), notes(1), popen(3), system(3),
"The Notesfile Reference Manual" in the *ConvexOS Tutorial Papers*

AUTHORS

Ray Essick

NAME

nfcomment – a user interface to the notesfile system

SYNOPSIS

nfcomment (notesfile, text, title, dirflag, anonflag)
char *nfname, *text, *title;

cc ... –lnfcom

DESCRIPTION

Nfcomment provides user programs with the ability to insert notes into a notesfile.

The note is inserted into the notesfile specified by *nfname*. *Text* is the address of the body of the note; this must be null-terminated. If *text* is NULL, the note is gathered from standard input until an EOF is encountered. The note is entered with the title specified by the *title* parameter. If the *dirflag* or *anonflag* parameters are non-zero, the director message is enabled or the note is entered anonymously. These take effect only if the user has the appropriate privileges in the notesfile.

Nfpipe is used to make the actual insertion of the text.

FILES

/usr/lib/libnfcom.a	-lnfcom library

SEE ALSO

nfpipe(1), notes(1), popen(3), system(3),
"The Notesfile Reference Manual" in the *ConvexOS Tutorial Papers*

AUTHORS

Ray Essick

Rob Kolstad

NAME

nice – set program priority

SYNOPSIS

nice(incr)

DESCRIPTION

This interface is obsoleted by setpriority(2).

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

Negative increments are ignored except on behalf of the superuser. The priority is limited to the range -64 (most urgent) to 64 (least).

The priority of a process is passed to a child process by *fork*(2). For a privileged process to return to normal priority from an unknown state, the system call *setpriority*(2) should be used; *nice* should **not** be used.

SEE ALSO

nice(1), setpriority(2), fork(2), renice(8)

NAME

nlist – get entries from name list

SYNOPSIS

#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*(5) for the structure declaration.

This subroutine is useful for examining the system name list kept in the file **/vmunix**. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out(5)

DIAGNOSTICS

nlist returns -1 and all type entries are set to 0 if the file cannot be found or if it is not a valid namelist.

NAME

pathconf, fpathconf – configurable pathname variables

SYNOPSIS

```
#include <unistd.h>

long pathconf(path, name)
char *path;
int name;

long fpathconf(fildes, name)
int fildes, name;
```

DESCRIPTION

The *pathconf()* and *fpathconf()* functions obtain information about configurable limits or options associated with a file or directory.

The values that may be interrogated follow.

_PC_LINK_MAX
: Maximum number of hard links to a file (see *link(2)*).

_PC_MAX_CANON
: Maximum number of bytes in a terminal canonical queue.

_PC_MAX_INPUT
: Minimum number of bytes which will be available in a terminal input queue; hence, the maximum number of bytes an application may require to be typed as input before reading them.

_PC_NAME_MAX
: Maximum number of bytes in a file name, not including the terminating null.

_PC_PATH_MAX
: Maximum number of bytes in a pathname, excluding a terminating null.

_PC_PIPE_BUF Maximum number of bytes that may be atomically written to a pipe.

_PC_CHOWN_RESTRICTED
: If true, the use of *chown(2)* to change a file's ownership is restricted to root, and changing the group ownership is restricted to a group in the caller's supplementary group ID list.

_PC_NO_TRUNC
: If true, pathnames longer than _PC_NAME_MAX generate an error.

_PC_VDISBALE The value to use in disabling special character processing for a terminal; see *tcgetattr(3)* and *tcsetattr(3)*.

RETURN VALUE

Upon successful completion, the value of the requested *name* is returned without changing *errno*. The value returned will be no more restrictive than the corresponding compile-time value from **<limits.h>**.

If *name* is an invalid value, –1 is returned.

If *name* has no limit for the path or file descriptor, the return is –1 and *errno* is unchanged.

ERRORS

pathconf() and *fpathconf()* will fail if one or more of the following are true:

[EINVAL] The value of *name* is invalid.

pathconf() fails if:

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

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG]
The length of *path* exceeds PATH_MAX, or the length of a component is greater than NAME_MAX for a file system with NO_TRUNC in effect.

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

[ELOOP] Too many symbolic links were encountered in translating *path*.

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

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

fpathconf() will fail if one or more of the following are true:

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

SEE ALSO

fstab(5), mtab(5), mount(8)

NOTES

The application is at the mercy of the system administrator's thoroughness in setting up /etc/fstab to accurately describe any non-BSD-style file systems that are remote-mounted.

The values returned by *pathconf()* for _PC_MAX_INPUT and _PC_MAX_CANON are a minimal approximation; it is recommended that the user call the *fpathconf()* function after opening the device to obtain accurate measures of serial device buffer sizes.

NAME

pause – stop until signal

SYNOPSIS

int pause()

DESCRIPTION

pause() never returns normally. It is used to give up control while waiting for a signal from *kill*(2) or an interval timer, such as *alarm(3c)*. Upon termination of a signal handler started during a *pause()*, the *pause()* call will return.

RETURN VALUE

Always returns –1.

ERRORS

Pause always returns:

[EINTR] The call was interrupted.

SEE ALSO

kill(2), select(2), sigpause(2),alarm(3c)

NAME

perror, strerror – system error messages

SYNOPSIS

#include <stdio.h>

void perror(const char *s);

#include <string.h>

char *strerror(int errnum);

DESCRIPTION

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

strerror returns a string containing a short error message related to the argument *errnum*. On ConvexOS, this message describes errors for the values that **errno** could acquire. **errno** can be used as the argument to *strerror*.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- *strerror* is not available in the backward-compatible mode.

SEE ALSO

intro(2), psignal(3)

NAME

plot: openpl, erase, label, line, circle, arc, move, cont, point, linemod, space, closepl – graphics interface

SYNOPSIS

```
openpl()

erase()

label(s)
char s[];

line(x1, y1, x2, y2)

circle(x, y, r)

arc(x, y, x0, y0, x1, y1)

move(x, y)

cont(x, y)

point(x, y)

linemod(s)
char s[];

space(x0, y0, x1, y1)

closepl()
```

DESCRIPTION

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

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

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

–lplot	device-independent graphics stream on standard output for *plot*(1) filters
–l300	GSI 300 terminal
–l300s	GSI 300S terminal
–l450	DASI 450 terminal
–l4014	Tektronix 4014 terminal

SEE ALSO

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

NAME

popen, pclose – initiate I/O to/from a process or pipe

SYNOPSIS

#include <stdio.h>

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

pclose(stream)
FILE *stream;

DESCRIPTION

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

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type "r" command may be used as an input filter, and a type "w" as an output filter.

SEE ALSO

pipe(2), fopen(3S), fclose(3S), system(3), wait(2), sh(1)

DIAGNOSTICS

Popen returns a null pointer if files or processes cannot be created, or the shell cannot be accessed.

Pclose returns –1 if *stream* is not associated with a 'popened' command.

BUGS

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

Popen always calls *sh*, never calls *csh*.

NAME

printf, fprintf, sprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, ... );

int fprintf(FILE *stream, const char *format, ... );

int sprintf(char *s, const char *format, ... );
```

DESCRIPTION

printf places output on the standard output stream *stdout*. *fprintf* places output on the named output *stream*. *sprintf* places output in the string *s*, followed by the character **\0**.

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

Each conversion specification is introduced by the character **%**.

After the **%**, the following appear in order:

- Zero or more *flags* in any order (described later) that modify the meaning of the conversion specification.
- An optional digit string specifying a *field width;* if the converted value has fewer characters than the field width, it will be padded with spaces on the left by default (or right, if the left adjustment flag, described later, has been given) to make up the field width.
- An optional *precision* which consists of a **.** , which serves to separate the precision from the width, followed by a decimal integer. The precision gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum characters to be written from a string in **s** conversions. If only a **.** is specified, a precision of 0 is implied.
- An optional **h** before an integral conversion means that the argument is of type **short int** or **unsigned short int** although the argument will have been promoted according to normal integral promotions. An **h** before a **n** specifies that the argument is a **short int***. The optional **l** specifies that a following integral conversion corresponds to a long 32-bit integer argument. The **l** followed by a **n** specifies that the argument is a **long int ***. The optional **L** means that the following floating-point conversion specifier applies to an argument of type **long double.** The **ll** and **LL** are CONVEX extensions specifying that the following integral conversion applies to **long long int** (64-bit integer variables).
- A character (enumerated later) that specifies the type of conversion.

An asterisk may be used in place of the field width or precision. In this case, an **int** argument is supplied in the argument list. A negative field width argument is taken as a – flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

– The result of the conversion will be left-justified.

\+ The result of the conversion will have a plus or minus sign.

space If the first character of a signed conversion is not signed, or if the conversion results in characters, a space will be prefixed to the result. This flag is ignored if a + flag is also used.

An optional **#** specifying that the value should be converted to an "alternate form". For **o** conversions, the precision of the number is increased to force the first character of the output string to a zero. For **x(X)** conversion, a nonzero result has the string **0x(0X)** prepended 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 only appears in the results of those conversions if a digit follows the decimal point). For **g** and **G** conversions, trailing zeros are not removed from the result.

0 For all numeric conversions, leading zeros following any sign or base are used to pad the field width. If the – flag is also used, the **0** flag is ignored. For integral conversions this flag is ignored if a precision is also specified.

The conversion characters and their meanings are:

d,i The integer argument is converted to a signed decimal. The precision specifies the *minimum* number of digits to appear; if the value is not large enough to use the indicated number of digits, it is padded with leading zeros. The default precision is 1. The result of converting the value zero with a precision of zero is no characters.

o,u,x,X The argument is of type **unsigned int** and is converted to unsigned octal, decimal, or hexadecimal notation. The letters a through f are used if the **x** specifier is used and A through F are used if the **X** specifier is used. The precision specifies the *minimum* number of digits to appear; if the value is not large enough to use the indicated number of digits, it is padded with leading zeros. The default precision is 1. The result of converting the value zero with a precision of zero is no characters.

f The argument of type **double** is converted to decimal notation in the style [–]**ddd.ddd**, where the number of **d**'s after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given. If the precision is **0** and the **#** was not specified, no decimal point is printed. If a decimal point character appears, at least one digit appears before it.

e,E The double argument is converted in the style [–]**d.ddde± dd**, where there is one digit before the decimal point and the number after is equal to the precision specification for the argument. The precision defaults to 6. If the precision is **0** and the **#** was not specified, no decimal point is printed. The exponent will always contain at least two digits. If the value is zero, the exponent is zero. Uppercase **E** causes the **e** in the exponent to be printed in uppercase.

g,G The **double** argument is printed in style **d**, in style **f**, or in style **e**. The precision specifies the number of significant digits. If the precision is zero, it taken as 1. The style depends on the value. The style **e** or **E** will be used only if the exponent resulting from such a conversion is less that -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion. A decimal only appears if it is followed by a digit. Uppercase **G** causes the **e** in the exponent (if any) to be printed in uppercase.

c The argument is converted to an **unsigned char** and is printed.

s The argument is taken to be an array of characters and characters from the array are printed until a null character or until the number of characters indicated by the precision specification is reached; however, if the precision is **0** or missing, all characters up to a null are printed. If the precision is not specified or is greater than the size of the array, the array will contain a null character.

p The argument is a pointer to **void.** The value of the pointer is printed as a hexadecimal integer.

n The argument is a pointer to an integer into which is *written* the number of characters transmitted thus far. No argument is converted.

% Prints a %; no argument is converted.

In no case does a nonexistent or small *field width* cause truncation of a field. Padding takes place only if the specified *field width* exceeds the actual width.

In all of the floating-point output formats, the output is converted to the specified precision by rounding.

RETURNS

These routines return the number of characters transmitted or a negative value if an output error occurred.

EXAMPLES

To print a date and time in the form 'Sunday, July 3, 10:02', where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimals:

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

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- The backward-compatible *sprintf* returns a pointer to its first argument. *printf* and *fprintf* return 0 or **EOF** to indicate status.
- The +, *space*, and **0** flags are not available in the backward-compatible mode.
- **#** is not considered a flag and must appear *after* the precision in the backward-compatible mode.
- The precision specifier is not allowed with integral conversions in the backward-compatible mode.
- The **%i** conversion specifier is unavailable in the backward-compatible mode.
- The **%p** conversion specifier is unavailable in the backward-compatible mode.
- The **%n** conversion specifier is unavailable in the backward-compatible mode.

SEE ALSO

putc(3S), scanf(3S), ecvt(3) va_list(3) vprintf(3s)

BUGS/NOTES

The **%hn** and **%ln** are not implemented.

The *printf* family of routines makes use of *ecvt, gcvt,* and *fcvt.* These conversion routines use an internal buffer which may be overwritten by the *printf* family of routines when floating-point numbers are printed.

NAME

psignal, sys_siglist – system signal messages

SYNOPSIS

psignal(sig, s)
unsigned sig;
char *s;

char *sys_siglist[];

DESCRIPTION

Psignal produces a short message on the standard error file describing the indicated signal. First the argument string *s* is printed, then a colon, then the name of the signal and a new-line. Most usefully, the argument string is the name of the program which incurred the signal. The signal number should be from among those found in *<signal.h>*.

To simplify variant formatting of signal names, the vector of message strings *sys_siglist* is provided; the signal number can be used as an index in this table to get the signal name without the newline. The define NSIG defined in *<signal.h>* is the number of messages provided for in the table; it should be checked because new signals may be added to the system before they are added to the table.

SEE ALSO

sigvec(2), perror(3)

NAME

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

SYNOPSIS

```
#include <stdio.h>

int putc(int c, FILE *stream);

int putchar(int c);

int fputc(int c, FILE *stream);

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

DESCRIPTION

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

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

fputc behaves like *putc*, but is a genuine function rather than a macro.

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

SEE ALSO

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

DIAGNOSTICS

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

BUGS

Because it is implemented as a macro, *putc* treats a *stream* argument with side effects improperly. In particular:

```
putc(c, *f++);
```

does not work sensibly.

To get *lint* to be quiet about references to *putc* and *putchar*, one must actually use the value that these functions return; using **void** will not work. Below is an example of a reference to *putc* that will keep *lint* happy:

```
if (putc ('a', stdout) == EOF) {
        fprintf (stderr, "Could not putc 'a' to stdout.\n");
        exit (1);
}
```

Errors can occur long after the call to *putc*.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

#include <stdio.h>

int puts(const char *s);

int fputs(const char *s, FILE *stream);

DESCRIPTION

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

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

Neither routine copies the terminal null character.

SEE ALSO

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

RETURN VALUE

fputs and *puts* return EOF if a write error occurs, otherwise they return a nonnegative value.

BUGS

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

NAME

qsort, bsearch – quicker sort and search

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
        int compare(const void *x, const void *y));

void bsearch(const  void *key,  const void *base,
        size_t nmemb, size_t size,
        int compare(const void *x, const void *y));
```

DESCRIPTION

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

bsearch searches an array of *nmemb* objects of size *size* pointed to by *base* for *key*. The array is assumed sorted according to the *compare* function, which is used to search for the *key*. *bsearch* returns a pointer to the array element that matches *key* or **NULL** if a match is not found.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- *bsearch* is not available in the backward-compatible mode.

SEE ALSO

sort(1)

NAME

rand, srand – random number generator

SYNOPSIS

#include <stdlib.h>

int rand(void);

void srand(unsigned int seed);

DESCRIPTION

rand uses a non-linear additive feedback random number generator employing a default table of 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random number generator is very large, approximately $16*(2^{31}-1)$.

The generator is reinitialized by calling *srand* with 1 as argument. *rand* can be set to produce a different pseudo-random sequence of numbers by calling *srand* with a different argument.

BACKWARD COMPATIBILITY

When compiling with the *-pcc* flag of the **cc** command, different library routines are linked. *rand* in the backward-compatible libraries is compatible with previous versions of *rand*. It uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$.

NAME

random, srandom, initstate, setstate – better random number generator; routines for changing generators

SYNOPSIS

long random()

srandom(seed)
int seed;

char *initstate(seed, state, n)
unsigned seed;
char *state;
int n;

char *setstate(state)
char *state;

DESCRIPTION

Random uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random number generator is very large, approximately $16*(2^{31}-1)$.

Random/srandom have (almost) the same calling sequence and initialization properties as *rand/srand.* The difference is that *rand*(3) produces a much less random sequence -- in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by *random* are usable. For example, "random()&01" will produce a random binary value.

Unlike *srand, srandom* does not return the old seed; the reason for this is that the amount of state information used is much more than a single word. (Two other routines are provided to deal with restarting/changing random number generators). Like *rand*(3), however, *random* will by default produce a sequence of numbers that can be duplicated by calling *srandom* with *1* as the seed.

The *initstate* routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by *initstate* to decide how sophisticated a random number generator it should use -- the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error). The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. *Initstate* returns a pointer to the previous state information array.

Once a state has been initialized, the *setstate* routine provides for rapid switching between states. *Setstate returns a pointer to the previous state array;* argument state array is used for further random number generation until the next call to *initstate* or *setstate.*

Once a state array has been initialized, it may be restarted at a different point either by calling *initstate* (with the desired seed, the state array, and its size) or by calling both *setstate* (with the state array) and *srandom* (with the desired seed). The advantage of calling both *setstate* and *srandom* is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

AUTHOR

Earl T. Cohen

DIAGNOSTICS

If *initstate* is called with less than 8 bytes of state information, or if *setstate* detects that the state information has been garbled, error messages are printed on the standard error output.

SEE ALSO

rand(3c)

BUGS

About 2/3 the speed of *rand*(3C).

NAME

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

SYNOPSIS

```
rem = rcmd(ahost, inport, locuser, remuser, cmd, fd2p);
char **ahost;
u_short inport;
char *locuser, *remuser, *cmd;
int *fd2p;

s = rresvport(port);
int *port;

ruserok(rhost, superuser, ruser, luser);
char *rhost;
int superuser;
char *ruser, *luser;
```

DESCRIPTION

rcmd is a routine used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. *rresvport* is a routine which returns a descriptor to a socket with an address in the privileged port space. *ruserok* is a routine used by servers to authenticate clients requesting service with *rcmd*. All three functions are present in the same file and are used by the *rshd*(8C) server (among others).

rcmd looks up the host **ahost* using *gethostbyname*(3N), returning –1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the call succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

The protocol is described in detail in *rshd*(8C).

The *rresvport* routine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by *rcmd* and several other routines. Privileged Internet ports are those in the range 0 to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

ruserok takes a remote host's name, as returned by a *gethostbyaddr*(3N) routine, two user names and a flag indicating whether the local user's name is that of the super-user. It then checks the files */etc/hosts.equiv* and, possibly, *.rhosts* in the local user's home directory to see if the request for service is allowed. A 0 is returned if the machine name is listed in the "hosts.equiv" file, or the host and remote user name are found in the ".rhosts" file; otherwise *ruserok* returns –1. If the *superuser* flag is 1, the checking of the "host.equiv" file is bypassed.

SEE ALSO

intro(2), hosts.equiv(5), rlogin(1C), rsh(1C), rexec(3X), rexecd(8C), rlogind(8C), rshd(8C)

DIAGNOSTICS

rcmd returns a valid socket descriptor on success. It returns –1 on error and prints a diagnostic message on the standard error.

rresvport returns a valid, bound socket descriptor on success. It returns –1 on error with the global value *errno* set according to the reason for failure. The error code EAGAIN is overloaded to mean "All network ports in use."

NOTES

rcmd is an optional product; for more information, contact your CONVEX sales representative.

NAME

rcvtir,dcvtid,ircvtr,idcvtd – IEEE/native floating point mode conversion routines.

SYNOPSIS

```
#include <math.h>

float rcvtir(x)
float x;

double dcvtid(x)
double x;

float ircvtr(x)
float x;

double idcvtd(x)
double x;
```

DESCRIPTION

rcvtir returns the value of its single precision native mode input in IEEE mode. A dirty zero (i.e. non-zero fractional part) will be returned as zero.

dcvtid returns the value of its double precision native mode input in IEEE mode. A dirty zero (i.e. non-zero fractional part) will be returned as zero.

ircvtr returns the value of its single precision IEEE mode input in native mode. A zero (dirty, or true) will be returned as zero.

idcvtd returns the value of its double precision IEEE mode input in native mode. A +/- zero (dirty, or true) will be returned as zero.

DIAGNOSTICS

rcvtir and *dcvtid* have the following boundry conditions and return values:

```
underflow -> 0
    Rop -> +Inf
```

ircvtr and *idcvtd* have the following boundry conditions and return values:

```
+/- Inf -> Rop
 +- Nan -> Rop
overflow -> Rop
```

FILES

/usr/lib/libm.a

/usr/lib/libF77.a

BUGS

There are problems with the single precision versions due to the conversion of function arguments to double precision in C. A reserved operand trap will result for *ircvtr* when negative zero (dirty or true) is passed as the argument. A call to *rcvtir* with a reserved operand as argument will result in a reserved operand trap prior to the call, rather than returning a reserved operand.

NAME

re_comp, re_exec – regular expression handler

SYNOPSIS

char *re_comp(s)
char *s;

re_exec(s)
char *s;

DESCRIPTION

Re_comp compiles a string into an internal form suitable for pattern matching. *Re_exec* checks the argument string against the last string passed to *re_comp*.

Re_comp returns 0 if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If *re_comp* is passed 0 or a null string, it returns without changing the currently compiled regular expression.

Re_exec returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and –1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both *re_comp* and *re_exec* may have trailing or embedded newline characters; they are terminated by nulls. The regular expressions recognized are described in the manual entry for *ed*(1), given the above difference.

SEE ALSO

ed(1), ex(1), grep(1)

DIAGNOSTICS

Re_exec returns –1 for an internal error.

Re_comp returns one of the following strings if an error occurs:

No previous regular expression,
Regular expression too long,
unmatched \(,
missing],
too many \(\) pairs,
unmatched \).

(those that do not contain a dot). This is the default.

RES_DNSRCH
If this option is set, the standard host lookup routine *gethostbyname*(3) will search for host names in the current domain and in parent domains; see *hostname*(7).

res_init reads the initialization file to get the default domain name and the Internet address of the initial hosts running the name server. If this line does not exist, the host running the resolver is tried. The control file */etc/use_nameserver* must exist in order to use the name server. *res_init* returns an immediate error if this file does not exist.

res_mkquery makes a standard query message and places it in *buf*. *res_mkquery* will return the size of the query or –1 if the query is larger than *buflen*. *Op* is usually QUERY but can be any of the query types defined in *nameser.h*. *dname* is the domain name. If *dname* consists of a single label and the RES_DEFNAMES flag is enabled (the default), the current domain name will be appended to *dname*. The current domain name is defined by the hostname or is specified in a system file; it can be overridden by the environment variable LOCALDOMAIN. *newrr* is currently unused but is intended for making update messages.

res_send sends a query to name servers and returns an answer. It will call *res_init* if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the message is returned, or –1 if there were errors.

dn_expand expands the compressed domain name *comp_dn* to a full domain name. Expanded names are converted to upper case. *msg* is a pointer to the beginning of the message, *exp_dn* is a pointer to a buffer of size *length* for the result. The size of compressed name is returned or -1 if there was an error.

dn_comp compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or -1 if there were errors. *length is the size of the array pointed to comp_dn*. *dnptrs* is a list of pointers to previously compressed names in the current message. The first pointer points to to the beginning of the message and the list ends with NULL. *lastdnptr* is a pointer to the end of the array pointed to *dnptrs*. A side effect is to update the list of pointers for labels inserted into the message by *dn_comp* as the name is compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

FILES

/etc/resolv.conf see resolver(5)
/etc/use_nameserver name server control file

SEE ALSO

gethostbyname(3), resolver(5), hostname(7), named(8),
RFC882, RFC883, RFC973, RFC974,
Name Server Operations Guide for BIND

NAME

res_mkquery, res_send, res_init, dn_comp, dn_expand – resolver routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

res_mkquery(op, dname, class, type, data, datalen, newrr, buf, buflen)
int op, class, type, datalen, buflen;
char *dname, *data, *buf;
struct rrec *newrr;

res_send(msg, msglen, answer, anslen)
char *msg, *answer;
int msglen, anslen;

res_init()

dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
char *exp_dn, *comp_dn, **dnptrs, **lastdnptr;
int length;

dn_expand(msg, eomorig, comp_dn, exp_dn, length)
char *msg, *eomorig, *comp_dn, *exp_dn;
int length;
```

DESCRIPTION

These routines are used for making, sending and interpreting packets for use with Internet domain name servers. Global information that is used by the resolver routines is kept in the variable *_res*. Most of the values have reasonable defaults and can be ignored. Options stored in *_res.options* are defined in *resolv.h* and are as follows. Options are stored a simple bit mask containing the bitwise "or" of the options enabled.

RES_INIT
: True if the initial name server address and default domain name are initialized (i.e., *res_init* has been called).

RES_DEBUG
: Print debugging messages.

RES_AAONLY
: Accept authoritative answers only. With this option, *res_send* should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_USEVC
: Use TCP connections for queries instead of UDP datagrams.

RES_STAYOPEN
: Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC
: Unused currently (ignore truncation errors, i.e., don't retry with TCP).

RES_RECURSE
: Set the recursion-desired bit in queries. This is the default. (*res_send* does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES
: If set, *res_mkquery* will append the default domain name to single-component names

NAME

restart - restart execution of a process or a process family

SYNOPSIS

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

pid_t restart(path, flags, signo)
char *path;
int options;
int signo;
```

DESCRIPTION

The *restart* library routine restarts a process or a process hierarchy from a checkpoint file.

By default the top level process of the process hierarchy will receive a SIGCONT signal. This can be modified with the *flags* field as described below.

The parameters passed to *restart* are used as follows:

path The file name of the checkpoint file for the root of the process hierarchy.

flags Optional actions to be performed by *restart*. This field is the inclusive OR of the following:

RESTART_DEBUG
When the process is restarted it will resume execution in the stopped state, making the process suitable for resuming execution under debugger control.

RESTART_FORCE
Force restart even if non-restartable conditions exist.

RESTART_SIGFAMILY
Send the signal *signo* to every process in the process hierarchy.

RESTART_SIGROOT
Send the signal *signo* to the top level process.

RESTART_SUSPEND
Restart all the processes in a suspended state.

The RESTART_SIGFAMILY and RESTART_SIGROOT options are mutually exclusive.

signo The signal to send if RESTART_SIGFAMILY or RESTART_SIGROOT are specified in the *flags* field. If neither RESTART_SIGFAMILY nor RESTART_SIGROOT is given the *signo* argument may be omitted or given a value of zero.

restart returns when the process has been successfully restarted. The parent process id of the restarted process is set to the caller's pid and the restarted process is a child of the caller, as if created via *fork*(2).

NOTES

The *restart* library function is implemented by invoking the *restart*(1) program with the appropriate arguments.

Processes may not call *restart* from within a multithreaded region.

RETURN VALUES

restart returns the process id of the top level process if the restart is successful, or -1 if the restart fails.

[EACCESS] Search permission is denied on a component of the checkpoint file *path* argument.

[EAGAIN] The process ID or process group ID of one or more of the processes to be restarted is currently in use in the system.

[EINTR] The *restart*() operation was interrupted by a signal.

[EINVAL] An invalid argument was passed to the function call.

[ENAMETOOLONG] The length of the *path* string exceeds {PATH_MAX}.

[ENOENT] The checkpoint file *path* does not exist.

SEE ALSO

chkpnt(1), restart(1), chkpnt(3), chkpnt(5)

NAME

restart – restart execution of a process or a process family

SYNOPSIS

integer function restart (path, flags, signo)
character*(*) path
integer flags
integer signo

DESCRIPTION

The *restart* library routine restarts a process or process hierarchy from a checkpoint file.

restart returns when the process has been successfully restarted. The parent process id of the restarted process is set to the caller's pid and the restarted process is a child of the caller.

By default the top level process of the process hierarchy will receive a SIGCONT signal. This may be modified with the *flags* field.

The arguments have the following meanings:

path The file name of the checkpoint file for the root of the process hierarchy.

flags Optional actions to be performed by *restart.*

signo Depending on the value of the *flags* argument, the *signo* parameter may be interpreted as an optional signal to send to restarted processes.

See *restart*(3) for a more complete description of the arguments to *restart.*

RETURN VALUE

restart returns the process id of the top level process if successful; otherwise an error code is returned.

FILES

/usr/lib/libc_old.a

SEE ALSO

chkpnt(3), restart(3), chkpnt(3f)

NAME

rexec – return stream to a remote command

SYNOPSIS

rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
u_short inport;
char *user, *passwd, *cmd;
int *fd2p;

DESCRIPTION

Rexec looks up the host **ahost* using *gethostbyname*(3N), returning –1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's *.netrc* file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; it will normally be the value returned from the call "getservbyname("exec", "tcp")" (see *getservent*(3N)). The protocol for connection is described in detail in *rexecd*(8C).

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then a auxiliary channel to a control process will be setup, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

SEE ALSO

rcmd(3X), rexecd(8C)

BUGS

There is no way to specify options to the *socket* call which *rexec* makes.

NOTES

Rexec is an optional product; for more information, contact your CONVEX sales representative.

NAME

scandir, alphasort – scan a directory

SYNOPSIS

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

scandir(dirname, namelist, select, compar)
char *dirname;
struct dirent *(*namelist)[];
int (*select)();
int (*compar)();

alphasort(d1, d2)
struct dirent **d1, **d2;
```

DESCRIPTION

scandir reads the directory *dirname* and builds an array of pointers to directory entries using *malloc*(3). It returns the number of entries in the array and a pointer to the array through *namelist*.

The *select* parameter is a pointer to a user supplied subroutine which is called by *scandir* to select which entries are to be included in the array. The select routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compar* parameter is a pointer to a user supplied subroutine which is passed to *qsort*(3) to sort the completed array. If this pointer is null, the array is not sorted.

alphasort is a routine which can be used for the *compar* parameter to sort the array alphabetically.

The memory allocated for the array can be deallocated with *free* (see *malloc*(3)) by freeing each pointer in the array and the array itself.

BACKWARD COMPATIBILITY

The backward compatible mode (**-pcc**) of the C compiler requires the following header files and declarations:

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

scandir(dirname, namelist, select, compar)
char *dirname;
struct direct *(*namelist)[];
int (*select)();
int (*compar)();

alphasort(d1, d2)
struct direct **d1, **d2;
```

The backward compatible mode of the C compiler predates the POSIX 1003.1 standard. This standard modified the **direct** data structure, requiring the data structure and header files to be renamed.

SEE ALSO

directory(3), malloc(3), qsort(3), dir(5)

DIAGNOSTICS

Returns –1 if the directory cannot be opened for reading or if *malloc*(3) cannot allocate enough memory to hold all the data structures.

NAME

scanf, fscanf, sscanf – formated input conversion

SYNOPSIS

#include <stdio.h>

int scanf(const char *format, ...);

int fscanf(FILE *stream, const char *format, ...);

int sscanf(const char *s, const char *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 which match optional white space in the input.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional maximum field width, an optional size indicator, and a conversion character.

The size indicator is one of the following characters: **h**, **l**, **L**, or **ll**. The **h** that precedes an integral conversion means that the argument is a pointer to a **short int** or an **unsigned short int**. The **l** that precedes an integral conversion means that the argument is a pointer to a **long int** or an **unsigned long int.** The **l** that precedes a numerical real conversion means that the argument is a pointer to a **double**. The **L** that precedes a numerical real conversion means that the argument is a pointer to a **long double**. The indicator **ll** is a CONVEX extension specifying that the following integral conversion is a pointer to a **long long int**.

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

The conversion character indicates the interpretation of the input field; it usually implies the type of pointer required. The following conversion characters are legal:

% A single "%" is expected in the input at this point; no assignment is done.

d An optionally signed decimal integer is expected; the corresponding argument should be an **int *.** The form of the integer is a sequence of decimal digits. The digits are interpreted as an integer in base 10. The form of the integer should match the input for the **strtol** function with the value 10 for the **base** argument.

i An optionally signed decimal integer is expected; the corresponding argument should be an **int *.** If the string of digits begins with a "0" followed by a string of digits, it is interpreted as an octal integer. If the string begins with "0X" or "0x" followed by a string of hexadecimal digits, it is interpreted as a hexadecimal integer. Otherwise, the sequence of decimal digits is interpreted as an integer in base 10. The form of the integer should match the input for the **strtol** function with the value 0 for the **base** argument.

o An optionally signed octal integer is expected; the corresponding argument should be an **unsigned int *.** The form of the integer should match the input for the **strtoul** function

with the value 8 for the **base** argument.

u An optionally signed decimal integer is expected; the corresponding argument should be an **unsigned int ***. The form of the integer should match the input for the **strtoul** function with the value 10 for the **base** argument.

x An optionally signed hexadecimal integer is expected; the corresponding argument should be an **unsigned int ***. The form of the integer should match the input for the **strtoul** function with the value 16 for the **base** argument.

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 space character or a newline.

c A character is expected. The next character in the stream is assigned even if it is a white space character. The corresponding argument should be a character pointer. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

e,f,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 the **subject** argument to the **strtod** function and is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e** followed by an optionally signed integer.

[Indicates a string that is not necessarily delimited by spaces but is composed only of characters in the *scanset*. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the *scanset*. If the first character is a] then it is taken to be part of the scanset and another] delimits the scanset. If the first character is not circumflex (^), the input field is all characters until the first character not in the scanset. If the first character after the left bracket is ^, the input field is all characters *not* in the scanset. The corresponding argument must point to a character array.

p Matches a hexadecimal number without a **0x** prefix which is taken as a pointer to **void**. This is interpreted as an address in hexadecimal form.

n No input is consumed. The argument is a pointer to an integer through which is *written* the number of characters read thus far. Execution of this directive does not increment the assignment count.

RETURNS

These functions return **EOF** if input failure occurs before any conversion. Otherwise, the number of successfully matched and assigned input items is returned. This can be used to decide how many input items were found. These routines return when the format string is exhausted or when the input does not match the expected form.

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
```

assigns to *i* the value 25, *x* the value 5.432, and *name* contains the null-terminated string "thompson". Or,

```
int i; float x; char name[50];
```

```
scanf("%2d%f% *d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

assigns 56 to *i*, 789.0 to *x*, skips "0123", and places the null-terminated string "56" in *name*. The next call to *getchar* returns "a".

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- The backward-compatible mode often continues assigning input items even after a conversion has failed.
- The backward-compatible mode does not completely match the input of the **strtod** function with real number conversion specifiers. For example, the input string "10e+rror" when fed to the format string "%e%c" assigns 'r' to the character argument instead of '+' which is the behavior of the conforming libraries.
- Many uppercase conversion specifiers are available in backward-compatible mode indicating **long int** arguments.
- The **L** indicator is not available in backward-compatible mode.
- The **p** conversion specifier is not available in backward-compatible mode.
- The **n** conversion specifier is not available in backward-compatible mode.
- The **i** conversion specifier is not available in backward-compatible mode.
- Form feed and vertical tab do not match the white-space directive in backward-compatible mode.
- The backward-compatible mode returns **EOF** if end of file is encountered regardless of the assignments made.

SEE ALSO

atof(3), *getc*(3S), *strtod*(3)

DIAGNOSTICS

The *scanf* functions return EOF on end of input, and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf, setbuffer, setlinebuf, setvbuf – assign buffering to a stream

SYNOPSIS

#include <stdio.h>

void setbuf(FILE *stream, char *buf);

int setvbuf(FILE *stream, char *buf, int mode, size_t size);

int setbuffer(FILE *stream, char *buf, int size);

int setlinebuf(FILE *stream);

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as it is written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is encountered or input is read from stdin. *fflush* (see *fclose*(3S)) may be used to force the block out early. Normally all files are block buffered. A buffer is obtained from *malloc*(3) upon the first *getc* or *putc*(3S) on the file. If the standard stream **stdout** refers to a terminal it is line buffered. The standard stream **stderr** is always unbuffered.

setbuf is used after a stream has been opened but before it is read or written. The character array *buf* is used as the stream buffer instead of a buffer automatically allocated. If *buf* is the constant pointer NULL, input/output will be completely unbuffered. Otherwise a buffer of size BUFSIZ is assumed.

setvbuf is used after a stream has been opened but before it is read or written. The *mode* argument determines how the stream will be buffered: **_IOFBF**, **_IOLBF**, or **_IONBF** to indicate full, line, or no buffering. If *buf* is not **NULL**, then *buf* must point to an array of *size* characters to be used as a buffer for the stream. *setvbuf* returns 0 if successful; nonzero if an improper mode is passed as *mode*.

setbuffer, an alternate form of *setbuf*, is used after a stream has been opened but before it is read or written. The character array *buf* whose size is determined by the *size* argument is used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered.

setlinebuf is used to change *stdout* or *stderr* from block buffered or unbuffered to line buffered. Unlike *setbuf* and *setbuffer* it can be used at any time that the stream is active.

A file can be changed from unbuffered or line buffered to block buffered by using *freopen* (see *fopen*(3S)). A file can be changed from block buffered or line buffered to unbuffered by using *freopen* followed by *setbuf* with a buffer argument of NULL.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- *setvbuf* is not available in the backward-compatible mode.

SEE ALSO

fopen(3S), getc(3S), putc(3S), malloc(3), fclose(3S), puts(3S), printf(3S), fread(3S)

BUGS

The standard error stream should be line buffered by default.

NAME

setfpmode – set floating point mode during program execution

SYNOPSIS

#include <convex/fpmode.h>

int setfpmode(value, level)
int value;
int level;

DESCRIPTION

Usually used by dual-mode programs, *setfpmode* will set the floating point mode of a program while it is executing. The valid *values* are FPMODE_IEEE and FPMODE_NATIVE (defined in *<convex/fpmode.h>*), which turn the IEEE bit of the PSW on or off, respectively.

The second argument, *level*, is used to specify where the given change should take affect. If *level* is 0, the entire program's floating point mode will be set to *value*. Only the current (calling) routine will be modified if *level* is 1. And for a *level* greater than 1, only the *level*th ancestor routine will have its mode set to *value*.

Although the *setfpmode* routine will return a negative value if IEEE hardware is needed but not available, it is a good idea for the user to take precautions for his own sake. When setting the floating point mode to FPMODE_IEEE, care should be taken by the user program to verify that the machine can support the mode. The *getsysinfo* system call can be used to obtain this information.

SEE ALSO

getfpmode(3), getsysinfo(2)

DIAGNOSTICS

A value of -1 is returned if an error is encountered, such as an invalid mode or lack of necessary hardware.

CAVEATS

Caution must be taken if a program uses a *setjmp* before it calls *setfpmode*. A call to *setfpmode* will only change the IEEE bits of the PSW's that are saved on the runtime stack. Because one of the items saved in *jmp_buf* by *setjmp* is the PSW, a *longjmp* back to that location may cause a previous floating point mode to be restored, losing the change made by *setfpmode*. A workaround to this problem is to change the IEEE bit in the PSW of the *jmp_buf* immediately following the *setfpmode* call. The PSW is stored in *jmp_buf*[2], and the IEEE bit is defined in *<machine/psw.h>*.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

#include <setjmp.h>

int setjmp(jmp_buf env);

void longjmp(jmp_buf env, int val);

int _setjmp(jmp_buf env);

int _longjmp(jmp_buf env, int val);

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

setjmp saves its stack environment in *env* for later use by *longjmp*. It returns value 0.

longjmp restores the environment saved by the last call of *setjmp*. It then returns in such a way that execution continues as if the call of *setjmp* had just returned the value *val* to the function that invoked *setjmp*, which must not itself have returned in the interim. All accessible data have values as of the time *longjmp* was called. It is not possible for the *longjmp* function to cause the *setjmp* function to return 0; if *val* is 0, the *setjmp* function returns a value of 1.

setjmp and *longjmp* save and restore the signal mask *sigmask*(2), while *_setjmp* and *_longjmp* manipulate only the C stack and registers.

longjmp may not be used to restore the environment saved by *_setjmp*, and *_longjmp* may not be used to restore the environment saved by *setjmp*. An error condition occurs in both of these cases.

SEE ALSO

sigvec(2), sigstack(2), signal(3C)

NAME

setlocale, localeconv – functions for locale manipulation

SYNOPSIS

#include <locale.h>

char *setlocale(int category, const char *locale);

struct lconv *localeconv(void);

DESCRIPTION

To solve problems faced by international users of the C programming language, ANSI introduced the notion of a *locale*. The locale is intended to describe international features that the library routines will subsequently act upon. Some of the international features are the alphabet, collation, currency formatting, date and time. The standard requires the implementation of the "C" locale, which is the minimum environment for translation with respect to locale. CONVEX supports one locale, the "C" locale.

setlocale queries or sets a portion of the program's locale. If *locale* is **NULL,** the value of the *category* for the current locale is returned. This value may subsequently be used by the *setlocale* function to restore that portion of the program's locale. To set a portion of program's locale, setlocale should be called with a supported category and locale. The categories supported are **LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC,** and **LC_MONETARY.** The "C" locale is the only valid locale. Calling setlocale with **LC_ALL** for a *category* indicates that all the categories are affected. If *locale* is "" the "default" locale is specified. The user can set the default locale by setting the environment variables **LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC,** or **LC_MONETARY** to a valid locale or by setting the environment variable **LANG** to a valid locale.

localeconv fills in a structure with values appropriate for formatting numeric quantities in the current locale.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. These routines are not available backward-compatible mode.

RETURNS

setlocale returns the value of the locale set, or the current locale if *locale* was **NULL**

localeconv returns a pointer to structure with the values appropriate for formatting numeric quantities. Since CONVEX only supports the "C" locale, the fields of the structure will contain the values required by the "C" locale.

NOTES

These functions are intended for use on systems implementing several locales and are provided for standard conformance.

SEE ALSO

setenv(3) strxfrm(3) mbstowcs(3)

NAME

setuid, setgid – set user and group ID

SYNOPSIS

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

int  setuid(uid)
uid_t uid;

int  setgid(gid)
gid_t gid;
```

DESCRIPTION

If the caller's effective uid is super-user, *setuid()* sets the real user ID, effective user ID, and saved set-user-ID of the current process to the value of *uid*.

If the caller is not super-user, but *uid* is equal to the real user ID or saved set-user-ID, *setuid()* sets the effective user ID to *uid*; the real user ID remains unchanged.

If the caller's effective gid uid is super-user, *setgid()* sets the real group ID, effective group ID, and saved set-group-ID of the current process to *gid*.

If the caller is not super-user, but *gid* is equal to the real group ID or the saved set-group-ID, the *setgid()* function sets the effective group ID to *gid*; the real group ID remains unchanged.

Any supplementary group IDs of the calling process remain unchanged by calls to *setgid()* or *setuid()*.

RETURN VALUE

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

ERRORS

setuid() fails if:

[EINVAL]	The value of *uid* is invalid.
[EPERM]	The caller is not super-user and *uid* does not match the real user ID or saved set-user-ID.

setgid() fails if:

[EINVAL]	The value of *gid* is invalid.
[EPERM]	The caller is not super-user and *gid* does not match the real group ID or saved set-user-ID.

BACKWARD COMPATIBILITY

seteuid, setruid, setegid, setrgid – set real or effective IDs

```
seteuid(euid)
setruid(ruid)
setegid(egid)
setrgid(rgid)
```

seteuid() (*setegid()*) sets the effective user ID (group ID) of the current process.

setruid() (*setrgid()*) sets the real user ID (group ID) of the current process.

SEE ALSO

setreuid(2), setregid(2), getuid(2), getgid(2)

NAME

signal, raise – simplified software signal facilities

SYNOPSIS

```
#include <signal.h>

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

int raise(signo)
int signo;
```

DESCRIPTION

Signal is a simplified interface to the more general *sigvec*(2) facility. Programs that use *signal* in preference to *sigvec* are more likely to be portable to all UNIX systems. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

raise sends the signal *signo* to the executing process.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see *tty*(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the *signal* call allows signals either to be ignored or to cause an interrupt to a specified location. The following is a list of all signals with names as in the include file *<signal.h>*:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught 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
SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
SIGIO	23•	i/o is possible on a descriptor (see *fcntl*(2))
SIGXCPU	24	cpu time limit exceeded (see *setrlimit*(2))
SIGXFSZ	25	file size limit exceeded (see *setrlimit*(2))
SIGVTALRM	26	virtual time alarm (see *setitimer*(2))
SIGPROF	27	profiling timer alarm (see *setitimer*(2))

SIGWINCH	28•	window size change
SIGLOST	29*	resource lost (see *lockd*(8C))
SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

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

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop. If *func* is SIG_IGN the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. **Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.**

If a caught signal occurs during certain system calls, causing the call to terminate prematurely, the call is automatically restarted. In particular this can occur during a *read*(2) or *write*(2) on a slow device (such as a terminal; but not a file) and during a *wait*(2).

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

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

NOTES

The signal handling routine can be declared:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number, into which the hardware faults and traps are mapped as defined below. *Code* is a parameter which further defines the type of hardware exception which occurred. *Scp* is a pointer to the *sigcontext* structure (defined in <*signal.h*>), used to restore the context from before the signal.

The following defines the mapping of hardware traps to signals and codes. All of these symbols are defined in <*signal.h*>:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Reserved Operand trap	SIGFPE	FPE_RESOP_TRAP
Segmentation Violations:		
Read access violation	SIGSEGV	SEG_READ_TRAP
Write access violation	SIGSEGV	SEG_WRITE_TRAP
Execute access violation	SIGSEGV	SEG_EXEC_TRAP
Invalid segment	SIGSEGV	SEG_INVSDR_TRAP
Invalid page table page	SIGSEGV	SEG_INVPTP_TRAP
Invalid memory reference	SIGSEGV	SEG_INVDATA_TRAP
I/O access violation	SIGSEGV	SEG_IOACC_TRAP
Ring Violations:		

Inward address reference	SIGBUS	BUS_INWADDR_TRAP
Outward ring call	SIGBUS	BUS_OUTCALL_TRAP
Inward ring return	SIGBUS	BUS_INWRTN_TRAP
Invalid syscall gate	SIGBUS	BUS_INVGATE_TRAP
Invalid return frame length	SIGBUS	BUS_INVFRL_TRAP
Illegal instruction:		
Error exit instruction	SIGILL	ILL_ERRXIT_TRAP
Privileged instruction	SIGILL	ILL_PRIVIN_TRAP
Undefined op code	SIGILL	ILL_UNDFOP_TRAP
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	

None of the above exceptions can be ignored using SIG_IGN because of the of pipelined nature of the CONVEX architecture. Indeterminate results will occur if one attempts ignore or return from a hardware-generated exception.

RETURN VALUE

signal returns the previous action on a successful call. Otherwise, –1 is returned and *errno* is set to indicate the error. *raise* returns 0 if successful, otherwise *raise* returns non-zero.

ERRORS

signal will fail and no action will take place if one of the following occur:

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

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

[EINVAL] An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

WARNINGS

Execution of this or related routines in parallel may yield unexpected results.

raise is only available in the ANSI conforming modes.

SEE ALSO

kill(1), kill(2), pattach(2), sigvec(2), sigblock(2), sigsetmask(2), sigpause(2), sigstack(2), setjmp(3), tty(4)

NAME

sigprocmask – block or unblock signals

SYNOPSIS

#include <signal.h>

int sigprocmask(how, set, oset)
int how;
sigset_t *set,*oset;

DESCRIPTION

sigprocmask() is used to examine or change the calling process's signal mask. If the argument *set* is not NULL, it points to a set of signals to be used in changing the currently blocked set.

Values for *how* are

Name	Description
SIG_BLOCK	Union of current signal mask and the signal set pointed to by *set.*
SIG_UNBLOCK	Intersection of current signal mask and the complement of the signal set pointed to by *set.*
SIG_SETMASK	Resulting signal mask is the set pointed to by *set*

If *oset* is not **NULL**, the previous mask is returned in the location pointed to by *oset.*

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

The call will block if any other thread in the process is currently servicing a signal contained in mask. Execution of the call will continue whenever the thread in the signal handler returns or enables the signal via the *sigprocmask()* function call.

RETURN VALUE

On success, zero is returned. On error, –1 is returned and *errno* is set to indicated the cause of failure.

ERRORS

sigprocmask() will fail if one or more of the following is true:

[EINVAL]	The value of *how* is not one of the defined values.
[EINTR]	The calling process is multi-threaded, and the call was interrupted while waiting for another thread to complete processing of a signal in *set.*

BACKWARD COMPATIBILITY

The *sigprocmask()* function replaces *sigblock()* and *sigsetmask().*

In previous releases of the operating system, it was also impossible to block SIGCONT. This is now permitted; the effect is to defer calling any handler installed for SIGCONT until SIGCONT is unblocked; when a SIGCONT is received, however, a stopped process is continued regardless of whether SIGCONT is blocked.

SEE ALSO

kill(2), sigaction(2), sigsuspend(3)

NAME

sigsetjmp, siglongjmp – non-local goto with signal state preservation

SYNOPSIS

#include <setjmp.h>

int sigsetjmp(env,savemask)
sigjmp_buf env;
int savemask;

void siglongjmp(env,val)
sigjmp_buf env;
int val;

DESCRIPTION

The *sigsetjmp()* macro complies with the definition of the *setjmp()* macro in the C standard. If the value of the *savemask* argument is not zero, the *sigsetjmp()* function shall also save the process's current signal mask (see **<signal.h>**) as part of the calling environment.

The *siglongjmp()* function complies with the definition of the *longjmp()* function n the C Standard. If and only if the *env* argument was initialized by a call to the *sigsetjmp()* function with a non-zero *savemask* argument, the *siglongjmp()* function restores the saved signal mask.

SEE ALSO

sigaction(), sigprocmask(), sigsuspend()

NAME

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate signal sets

SYNOPSIS

```
#include <signal.h>

int sigemptyset(set)
sigset_t *set;

int sigfillset(set)
sigset_t *set;

int sigaddset(set, signo)
sigset_t *set;
int signo;

int sigdelset(set,signo)
sigset_t *set;
int signo;

int sigismember(set,signo)
sigset_t *set;
int signo;
```

DESCRIPTION

The *sigsetops* primitives manipulate sets of signals in a portable fashion.

sigemptyset() initializes the signal set pointed to by the argument *set* such that all signals are excluded.

sigfillset() initializes the signal set pointed to by the argument *set* such that all signals are included.

Either *sigfillset()* or *sigemptyset()* must be used to initialize a signal set object.

sigaddset() and *sigdelset()* respectively add to and delete from *set* the signal *signo*.

sigismember() tests whether the indicated signal *signo* is a member of *set*.

NAME

sigsuspend – wait for signal

SYNOPSIS

#include <signal.h>

int sigsuspend(sigmask)
sigset_t *sigmask;

DESCRIPTION

sigsuspend() replaces the caller's signal mask with the set of signals pointed to by the *sigmask* argument. Execution is then suspended until the deliver of a signal whose action is either to execute a signal handler or to terminate the process. If the action is to terminate the process, the *sigsuspend()* function does not return. If the action is to handle the function, *sigsuspend()* returns with the signal mask restore to the set that existed prior to the call for processes that do not specify the _SA_PARALLEL flag in the *sa_flags* member of the *sigaction* structure used to install the handler.

When calling *sigsuspend()*, the *sigmask* argument is usually the empty set to indicate that no signals are now to be blocked.

sigsuspend() always terminates by being interrupted and returns EINTR. The call will block if any other thread in the process is currently running a signal handler for any signal included in sigmask. Execution of the call will continue whenever the thread in the signal handler returns or makes a *sigsetmask()* call to re-enable the signal.

In normal usage, signals are blocked using *sigprocmask*(3) to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using *sigsuspend()* with the mask returned by *sigprocmask*.

NOTES

sigsuspend() replaces *sigpause(2)* and is only available in the extended and strict POSIX modes of CONVEX C.

SEE ALSO

sigprocmask(3), sigaction(2), kill(2)

NAME

sin, cos, tan, asin, acos, atan, atan2, sinf, cosf, tanf, asinf, acosf, atanf, atan2f, ssin, scos, stan, sasin, sacos, satan, satan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin(double x);

double cos(double x);

double tan(double x);

double asin(double x);

double acos(double x);

double atan(double x);

double atan2(double x, double y);

float sinf(float x);

float cosf(float x);

float tanf(float x);

float asinf(float x);

float acosf(float x);

float atanf(float x);

float atan2f(float x, float y);
```

DESCRIPTION

sin, cos, and *tan* return double-precision trigonometric functions of radian arguments; *ssin, scos,* and *stan* return single-precision trigonometric functions of radian arguments.

asin returns the double-precision arc sine in the range $-\pi/2$ to $\pi/2$; *asinf* returns the single-precision arc sine.

acos returns the double-precision arc cosine in the range 0 to π; *acosf* returns the single-precision arc cosine.

atan returns the double-precision arc tangent of x in the range $-\pi/2$ to $\pi/2$; *atanf* returns the single-precision arc tangent of x.

atan2 returns the double-precision arc tangent of x/y in the range $-\pi$ to π; *atan2f* returns the single-precision arc tangent of x/y.

DIAGNOSTICS

On range or domain error, **errno** is set to **ERANGE** or **EDOM.** Functions that could result in a domain error are *acos, asin, atan2, cos, sin, acosf, asinf, atan2f, cosf,* and *sinf.* Functions that could result in a range error are *sinh, cosh, sinhf,* and *coshf.* On overflow, **HUGE_VAL** of the same sign is returned.

The value of the sine and cosine functions for very large arguments is unmeaningful. This is related to the accuracy of the argument reduction. For single precision, the maximum value is $\pi * 2^{23}$. For double precision, the maximum value is $\pi * 2^{52}$. If the argument exceeds the maximum, it is replaced with zero and evaluation continues. The arc sine and arc cosine functions are undefined for values greater than 1. Arguments greater than 1 are replaced with 1 and evaluation continues. The arc tangent functions with two arguments are undefined when both arguments are 0. In this case, $\pi/2$ is returned.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- The names of the floating-point routines are *ssin, scos, stan, sasin, sacos, satan,* and *satan2* in the backward-compatible libraries.
- **errno** may be set to the following values in the **MTH_LRG_SIN, MTH_LRG_COS, MTH_UNDEF_ASIN, MTH_UNDEF_ACOS** or **MTH_UNDEF_ATAN2** in the backward-compatible libraries. In addition to setting **errno** , error messages are printed on domain and range error.

NAME

sinh, cosh, tanh, sinhf, coshf, tanhf, ssinh, scosh, stanh – hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh(double x);

double cosh(double x);

double tanh(double x);

float sinhf(float x);

float coshf(float x);

float tanhf(float x);
```

DESCRIPTION

These functions compute the designated hyperbolic functions for real arguments.

DIAGNOSTICS

The possible errors and corresponding return values are:

[MTH_OVF_SINH], [MTH_OVF_COSH]

Evaluation of hyperbolic sine or cosine would lead to overflow. The maximum $|x|$ for which these functions can be computed is approximately ln(xmax+ln(2)), where xmax is the largest floating-point value. If the argument exceeds the maximum, then the largest-floating point value of the appropriate sign is returned.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- The names of the floating-point routines are *ssinh*, *scosh*, and *stanh*.

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep(seconds)
unsigned seconds;

DESCRIPTION

The *sleep*() function causes the current process to be suspended from execution until either the number of real time seconds specified by the argument *seconds* have elapsed or a signal is delivered to the calling process and its action is to an invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

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

The sleep routine uses the SIGALRM signal, thus temporarily redefining the existing SIGALRM signal handler and temporarily unblocking SIGALRM. If a SIGALRM is pending on entry, *sleep*() will return immediately with the pended signal discarded.

setitimer(2) supports timing with a finer resolution.

RETURN VALUE

If the *sleep*() function returns because the requested time has elapsed, the value returned will be zero. If the *sleep*() function returns due to delivery of a signal, the value returned will be the unslept amount (the requested time minus the time actually slept) in seconds.

SEE ALSO

getitimer(2), sigpause(2)

WARNINGS

Execution of this routine in parallel may produce unexpected results.

NAME

stdarg.h, stdarg, va_list, va_start, va_arg, va_end – variable argument list

SYNOPSIS

#include <stdarg.h>

va_list argList;
va_start(va_list argList, <last parameter>)
va_arg(va_list argList, <type>)
va_end(va_list argList)

DESCRIPTION

This set of macros provides a means of writing portable functions that accept variable argument lists (e.g., printf). Routines having variable argument lists that do not use stdarg are not portable since different machines use different argument passing conventions. (The older varargs(3) mechanism provides a slightly less portable method of writing such routines. It is available on other operating systems but is not part of ANSI C).

The stdarg variable argument list system is for use with functions with prototypes using the "..." syntax to indicate that varying numbers of actual parameters may be present. There must be at least one named parameter.

va_list is a type used to declare a variable which describes the current state of the variable argument list. To use the stdarg.h system you must declare a variable of this type and initialize it with a call to **va_start**.

va_start(*argList,lastParam*) is called to initialize *argList*. The second parameter to va_start must be the last named parameter in the function's prototype.

va_arg(*argList, type*) will return the next argument in the list pointed to by *argList*. The second argument to va_arg is the type the argument is expected to be. The type may not be **short**, **char**, or **float**, or any signed, unsigned or qualified version of those types. Typically the routine determines what type the argument is based on the preceding arguments (e.g., printf's format string).

va_end(*argList*) is used to indicate that argument traversal is complete.

Multiple traversals, each bracketed by **va_start** and **va_end**, are possible, and the same va_list type variable may be reused.

The address of a **va_list** type variable may be passed to another function and **va_arg** used on the variable in that function. The **va_list** type variable must first be initialized by a call to **va_start** in the function whose parameter list is being traversed.

The stdarg macros do not check that the number of calls to va_start matches the number of arguments pushed. The routine using stdarg.h and its caller must agree on a convention for determining how many arguments there are. For example, *execl* expects a 0 argument to signal the end of the list while the format string to *printf* indicates how many arguments there should be.

EXAMPLE

```
#include <stdarg.h>
execl(char * name, ...)
{
        va_list ap;
        char *args[100];
        int argno = 0;

        va_start(ap, name);
        while (args[argno++] = va_arg(ap, char *))
                ;
        va_end(ap);
```

```
        return execv(name, args);
}
```

NAME

stddef.h, NULL, offsetof, ptr_diff_t, size_t, wchar_t – header file contains standard definitions

SYNOPSIS

#include <stddef.h>

NULL;

offsetof(type, member-designator);

ptr_diff_t;

size_t;

wchar_t;

DESCRIPTION

stddef.h contains standard definitions used by the library functions.

ptr_diff_t is the type of the result of subtracting two pointers. **size_t** is the type of the result of the **sizeof** operator.

wchar_t is the type that can hold the largest character set implemented. Currently, CONVEX supports no extended character set but **wchar_t** is provided for standard conformance.

NULL is a pointer constant having special meaning to many library functions.

offsetof expands to **size_t** which is the offset in bytes of *member-designator* from the beginning of its structure. The structure must be of type *type*.

NAME

strlen, index, rindex – string operations

SYNOPSIS

```
#include <string.h>

size_t strlen(s)
const char *s;

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

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

DESCRIPTION

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

strlen returns the number of characters in *s*, which must be terminated by a null character.

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

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

NOTES

rindex and *index* are available only in the extended mode and backward-compatible mode of CONVEX C.

SEE ALSO

stringcpy(3), stringcmp(3), stringsearch(3), bstring(3).

NAME

strcat, strncat – string concatenation functions

SYNOPSIS

#include <string.h>

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

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

DESCRIPTION

strcat appends a copy of string *s2* to the end of string *s1*. *strncat* copies at most *n* characters. The target may not be null-terminated if the length of *s2* is *n* or more. Both return a pointer to the result.

SEE ALSO

stringcpy(3), stringcmp(3), stringsearch(3), bzero(3)

NAME

memcmp, strcmp, strcoll, strncmp, strxfrm – string comparison functions

SYNOPSIS

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);

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

int strcoll(const char *s1, const char *s2);

int strncmp(const char *s1, const char *s2, size_t n);

size_t strxfrm(char *s1, const char *s2, size_t n);
```

DESCRIPTION

The *memcmp* function compares the first *n* characters of the object pointed to by *s1* to the first *n* characters of the object pointed to by *s2*.

strcmp compares the string pointed to by *s1* to the string pointed to by *s2*.

strcoll also compares the string pointed to by *s1* to the string pointed to by *s2*. The string is interpreted as appropriate to the **LC_COLLATE** category of the current locale. Currently, CONVEX only supports the "C" locale.

strncmp compares not more than *n* characters from the strings pointed to by *s1* and *s2*.

strxfrm transforms the string pointed to by *s2* into the string pointed to by *s1*. If two strings are transformed, the comparison by the *strcoll* function on the original strings is the same as the comparsion by the *strcmp* function on the transformed strings. Not more than *n* characters are placed in *s1*.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- *memcmp* is not available in the backward-compatible mode.
- *strcoll* is not available in the backward-compatible mode.
- *strxfrm* is not available in the backward-compatible mode.

RETURNS

The *memcmp, strcmp, strcoll,* and *strncmp* return an integer greater than, equal to, or less than zero according to the results of the comparison. *strxfrm* returns the length of the transformed string (not counting the **NULL** terminator).

NOTES

strcoll and *strxfrm* are provided for standard conformance. They are intended for use in environments with multiple locales.

SEE ALSO

stringcat(3), stringcpy(3), stringsearch(3), bzero(3), setlocale(3)

NAME

memcpy, memmove, memset, strcpy, strncpy – string copy functions

SYNOPSIS

#include <string.h>

void *memcpy(void *dest, const void *src, size_t length);

void *memmove(void *dest, const void *src, size_t length);

void *memset(void *dest, int c, size_t length);

char *strcpy(char *dest, const char *src);

char *strncpy(char *dest, const char *src, size_t length);

DESCRIPTION

memcpy copies *length* characters from *src* to *dest*. If the objects overlap the behavior is undefined.

memmove copies *length* characters from *src* to *dest*. The copying is done as if it were copied first to a temporary block so that overlapping of strings is permitted.

The *memset* function copies the value of *c* (converted to an unsigned char) into each of the first *length* characters of the object pointed to by *dest*.

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

The functions *memcpy*, *memmove*, *memset*, *strcpy* and *strncpy* return the value of *dest*.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- *memcpy* is not available in the backward-compatible mode.
- *memmove* is not available in the backward-compatible mode.
- *memset* is not available in the backward-compatible mode.

SEE ALSO

stringcat(3), stringcmp(3), stringsearch(3), bzero(3)

NAME

memchr, strchr, strrchr, strcspn, strpbrk, strstr, strspn, strtok, index, rindex – string search functions

SYNOPSIS

```
#include <string.h>

void *memchr(const void *s, int c, size_t n);

char *strchr(const char *s, int c);

size_t strcspn(const char *s1, const char *s2);

char *strpbrk(const char *s1, const char *s2);

char *strrchr(const char *s, int c);

size_t strspn(const char *s1, const char *s2);

char *strstr(const char *string, const char *pattern);

char *strtok(char *s1, const char *s2);

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

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

DESCRIPTION

memchr locates the first occurrence of *c* (converted to an *unsigned char*) in the initial *n* characters (each interpreted as *unsigned char*) of the object pointed to by *s*.

strchr locates the first occurrence of *c* (converted to a *char*) in the string pointed to by *s*. The terminating null character is considered to be part of the string.

strcspn computes the length of the maximum initial segment of the string pointed to by *s1* which consists entirely of characters **not** from the string pointed to by *s2*.

strpbrk locates the first occurrence in the string pointed to by *s1* of any character pointed to by *s2*.

strrchr locates the last occurrence of *c* (converted to *unsigned char*) in the string pointed to by *s*. The terminating null character is considered to be part of the string.

strspn computes the length of the maximum initial segment of the string pointed to by *s1* which consists entire of character from the string pointed to by *s2*.

strstr locates the first occurrence of the string pointed to by *pattern* in the string pointed to by *string*. The terminating null character in *pattern* is not included in the pattern-match.

strtok breaks a string, *s1*, into a sequence of tokens, delimited by the characters contained in the string *s2*.

A sequence of calls to the *strtok* function breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *s2*. The first call in the sequence has *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *s2* my be different from call to call.

The first call in the sequence searches the string pointed to by *s1* for the first character that is **not** contained in the current separator string pointed to by *s2*. If no such character is found, then there are no tokens in the string pointed to by *s1* and the *strtok* function returns a null pointer. If such a character is found, it is the start of the first token.

The *strtok* function then searches from there for a character that **is** contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searched for a token will return a null pointer. If such a

character is found, it is overwritten by a null character, which terminates the current token. The *strtok* function saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The *index* function locates the first occurrence of *c* in the string pointed to by *s*.

The *rindex* function locates the last occurrence of *c* in the string pointed to by *s*.

RETURNS

memchr returns a pointer to the located character, or a null pointer if the character does not occur in the object.

strchr returns a pointer to the located character, or a null pointer if the character does not occur in the string.

strcspn returns the length of the segment.

strpbrk returns a pointer to the character, or a null pointer if no character from *s2* occurs in *s1*.

strrchr returns a pointer to the character, or a null pointer if *c* does not occur in the string.

strspn returns the length of the segment.

strstr returns a pointer to the located string, or a null pointer if the string is not found. If *pattern* points to a string with zero length, the function returns *string*.

strtok returns a pointer to the first character of a token, or a null pointer if there is no token. *index* returns a pointer to the located character, or zero if the character does not occur in the string. *rindex* returns a pointer to the located character, or zero if the character does not occur in the string.

BACKWARD COMPATIBILITY

Only *index* and *rindex* may be used in the backward compatible mode. These functions are also available in the extended mode the compiler.

SEE ALSO

stringcmp(3), stringcpy(3), stringcat(3), stringlen(3)

NAME

strtod, strtol, strtoul – string to numeric value conversion routines

SYNOPSIS

#include <stdlib.h>

double strtod(const char *nptr, char **endptr);

long int *strtol(const char *nptr, char **endptr, int base);

unsigned long int *strtoul(const char *nptr, char **endpt, int base);

DESCRIPTION

strtod converts the initial portion of the string in *nptr* to a double. Whitespaces are skipped. The remainder of the string is decomposed into a subject string and a final string containing unrecognized characters. The subject sequence is an optional plus or minus, then a nonempty digit sequence optionally containing a decimal point, followed by an optional exponent. An exponent is an **e** or an **E**, then an optional plus or minus, followed by a nonempty digit sequence. If *endptr* is not **NULL**, a pointer to the final sequence is placed in **endptr*. No floating point suffix is recognized as part of the subject sequence.

strtol converts the initial portion of the string in *nptr* to a long integer. Whitespaces are skipped. The remainder of the string is decomposed into a subject string and a final string containing unrecognized characters. If the base is zero, the subject sequence optionally begins with a plus or minus. Next is a zero optionally followed by a digit sequence indicating an octal number, or a **0x** (or **0X**) followed by a digit sequence indicating a hexadecimal number, or any other digit sequence indicating a decimal number. If the value of **base** is between 2 and 36 the subject sequence is a string represented by digits and letters. The letters **a** (or **A**) through **z** (or **Z**) are ascribed values 10 to 35. If the base is 16 an optional **0x** or **0X** may follow the sign and precede the letters and digits. If *endptr* is not **NULL**, a pointer to the final sequence is placed in **endptr*. No integer suffix is recognized as part of the subject sequence.

strtoul is the same as *strtol* except that the string is interpreted as an unsigned long integer.

RETURNS

strtod returns the value of the subject string, converted to a double. If the subject string is empty or in an unexpected form, zero is returned. If the subject string represents a number outside the representable range of double, plus or minus **HUGE_VAL** is returned. If the correct value would cause underflow, zero is returned.

strtol returns the value of the subject string, converted to a long integer. If the subject string is empty or in an unexpected form, zero is returned. If the subject string represents a number outside the representable range of a long integer, **LONG_MIN** or **LONG_MAX** is returned.

strtoul returns the value of the subject string, converted to a unsigned long integer. If the subject string is empty or in an unexpected form, zero is returned. If the subject string represents a number outside the representable range of a long integer, **ULONG_MAX** is returned.

DIAGNOSTICS

On range error, **errno** is set to **ERANGE**.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. These routines are not available backward-compatible mode.

SEE ALSO

atof(3), atoi(3), atol(3), atoll(3)

NAME

stty, gtty – set and get terminal state (defunct)

SYNOPSIS

```
#include <sgtty.h>

stty(fd, buf)
int fd;
struct sgttyb *buf;

gtty(fd, buf)
int fd;
struct sgttyb *buf;
```

DESCRIPTION

This interface is obsoleted by ioctl(2).

Stty sets the state of the terminal associated with *fd*. *Gtty* retrieves the state of the terminal associated with *fd*. To set the state of a terminal the call must have write permission.

The *stty* call is actually "ioctl(fd, TIOCSETP, buf)", while the *gtty* call is "ioctl(fd, TIOCGETP, buf)". See *ioctl*(2) and *tty*(4) for an explanation.

DIAGNOSTICS

If the call is successful 0 is returned, otherwise –1 is returned and the global variable *errno* contains the reason for the failure.

SEE ALSO

ioctl(2), tty(4)

NAME

swab – swap bytes

SYNOPSIS

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

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to,* exchanging adjacent even and odd bytes. It is useful for carrying binary data between machines that use different byte ordering, such as CONVEX to VAX transfers. *Nbytes* should be even.

NAME

sysconf – get configurable system variables

SYNOPSIS

#include <limits.h>
#include <unistd.h>

long sysconf(name)
int name;

DESCRIPTION

The *sysconf()* function provides values of configurable system variables or options. The *name* argument represents the value to be queried. The permissible values of *name* are found in **<unistd.h>**. The values that may be interrogated follow.

_SC_ARG_MAX Maximum length of arguments and environment for the *exec* functions.

_SC_CHILD_MAX
Maximum number of simultaneous processes per real user id.

_SC_CLK_TCK Number of intervals per second, used to express the value in type *clock_t*.

_SC_NGROUPS_MAX
Maximum number of simultaneous supplementary group IDs per process.

_SC_OPEN_MAX
Maximum number of files that a process may have open at one time.

_SC_JOB_CONTROL
Job control is supported.

_SC_SAVED_IDS
Each process has a saved set-user-ID and a saved set-group-ID.

_SC_VERSION Corresponds to the revision of IEEE Std. P1003.1-1988 supported.

RETURN VALUE

Upon successful completion, the value of the requested *name* is returned. If the requested *name* has no limit, –1 is returned and *errno* is left unchanged. On error, –1 is returned and *errno* is set to indicate the error.

ERRORS

sysconf() will fail if:

[EINVAL] The value of *name* is invalid.

NOTES

The name of CLK_TCK is debatable; X3J11 seems to have changed in favor of CLOCKS_PER_SEC. One might claim that by inference, _SC_CLK_TCK should be renamed _SC_CLOCKS_PER_SEC.

NAME

syslog, openlog, closelog, setlogmask – control system log

SYNOPSIS

#include <syslog.h>

void openlog(const char *ident, int logopt, int facility)

void syslog(int priority, const char *message, ...)

void closelog(void)

int setlogmask(int maskpri)

DESCRIPTION

Syslog arranges to write *message* onto the system log maintained by *syslogd*(8). The message is tagged with *priority*. The message looks like a *printf*(3) string except that **%m** is replaced by the current error message (collected from *errno*). A trailing newline is added if needed. This message will be read by *syslogd*(8) and written to the system console, log files, or forwarded to *syslogd* on another host as appropriate.

Priorities are encoded as a *facility* and a *level*. The facility describes the part of the system generating the message. The level is selected from an ordered list:

LOG_EMERG	A panic condition. This is normally broadcast to all users.
LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, e.g., hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but should possibly be handled specially.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

If *syslog* cannot pass the message to *syslogd*, it will attempt to write the message on */dev/console* if the LOG_CONS option is set (see below).

If special processing is needed, *openlog* can be called to initialize the log file. The parameter *ident* is a string that is prepended to every message. *Logopt* is a bit field indicating logging options. Current values for *logopt* are:

LOG_PID	log the process id with each message: useful for identifying instantiations of daemons.
LOG_CONS	Force writing messages to the console if unable to send it to *syslogd*. This option is safe to use in daemon processes that have no controlling terminal since *syslog* will fork before opening the console.
LOG_NDELAY	Open the connection to *syslogd* immediately. Normally the open is delayed until the first message is logged. Useful for programs that need to manage the order in which file descriptors are allocated.
LOG_NOWAIT	Don't wait for children forked to log messages on the console. This option should be used by processes that enable notification of child termination via SIGCHLD, as *syslog* may otherwise block waiting for a child whose exit status has already been collected.

The *facility* parameter encodes a default facility to be assigned to all messages that do not have an explicit facility encoded:

LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_MAIL	The mail system.
LOG_DAEMON	System daemons, such as *ftpd*(8), *routed*(8), etc.
LOG_AUTH	The authorization system: *login*(1), *su*(1), *getty*(8), etc.
LOG_LPR	The line printer spooling system: *lpr*(1), *lpc*(8), *lpd*(8), etc.
LOG_MIG	The automated file migration system.
LOG_LOCAL0	Reserved for local use. Similarly for LOG_LOCAL1 through LOG_LOCAL7.

Closelog can be used to close the log file.

Setlogmask sets the log priority mask to *maskpri* and returns the previous mask. Calls to *syslog* with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro LOG_MASK(*pri*); the mask for all priorities up to and including *toppri* is given by the macro LOG_UPTO(*toppri*). The default allows all priorities to be logged.

EXAMPLES

```
syslog(LOG_ALERT, "who: internal error 23");

openlog("ftpd", LOG_PID, LOG_DAEMON);
(void)setlogmask(LOG_UPTO(LOG_ERR));
syslog(LOG_INFO, "Connection from host %d", CallingHost);

syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```

SEE ALSO

logger(1), syslogd(8)

NAME

system – issue a shell command

SYNOPSIS

#include <stdlib.h>

int system(const 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 value of the command that was executed.

system may not be called from a multi-threaded process.

SEE ALSO

popen(3), fork(2), execve(2)

DIAGNOSTICS

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

system always returns the value 255 in the extended, conforming, or strict compatibility modes if a *SIGCHLD* signal handler is installed.

NAME

tpmount, tpunmount, tpqueue, tpattr, tplabel, tpunlabel, tpwait, tpstatus, tperror – tape subsystem calls

SYNOPSIS

```
#include <tape.h>

tpmount(mntopt)
struct tp_mntopt *mntopt;

tpunmount(symname, keeponline)
char *symname;
int keeponline;

struct queue_t *
tpqueue()

tpattr(symname, blocksize, recordsize, format, fileident, access, delim,expir_date,uhl,utl)
char *symname;
int blocksize, recordsize;
char format;
char *fileident;
int access;
int delim;
char *expir_date;
char *uhl[8];
char *utl[8];

tplabel(symname, access)
char *symname;
char access;

tpunlabel(symname)
char *symname;

tplist(symname, labselect, flags)
char *symname;
int labselect;
int flags;

tpwait(symname)
char *symname;

tpstatus(cwd, symname)
char *cwd;
char *symname;

tperror(s)
char *s;

char *tp_errlist[];
int tp_nerr;
```

DESCRIPTION

These functions make calls to the tape subsystem. Each of these routines perform the same function as the user commands (see *tpmount*(1), et al). These functions are obtained with the loader option -ltape.

tpmount() performs the same function as *tpmount*(1), except the request is always put into the background. *tpwait*() must be called to wait for the request to finish. The input structure is defined in *tape.h*:

```
struct tp_mntopt {
    int flags;                  /* Mount flags (see TF_*)*/
    int skip;                   /* Number of file to skip from beginning */
    char *label;                /* The label type (ansi, ibm, etc). */
    char *type;                 /* Tape type */
    char *density;              /* Tape density */
    char *speed;                /* Tape speed */
    char *comment;              /* Comment for operator */
    char *sym_name;             /* Name for symbolic link to drive */
    char *vsns;                 /* Volume serial numbers, separated by commas */
    char *dev_name;             /* Drive to allocate */
    struct tp_mntopt *next;     /* Pointer to the next tape in the list of */
                                /* tapes that need mounted simultaneously */
};

/* tpmount flags */
#define TF_MODE                 0007    /* the device mode: block, char, labeled */
#define   TM_DEFAULT            0000    /* use default device mode */
#define   TM_CHAR               0001    /* char special device */
#define   TM_BLOCK              0002    /* block special device */
#define   TM_LABEL              0003    /* labeled */
#define TF_READONLY             00010   /* tape should not have a write ring */
#define TF_READONLY_DEFAULT     00020   /* use default value */
#define TF_KEEPLOCAL            00040   /* keep tape mount on local host */
#define TF_KEEPLOCAL_DEFAULT    00100   /* use default value */
#define TF_REWINDDEV            00200   /* rewind device is desired */
#define TF_REWINDDEV_DEFAULT    00400   /* use default value */
#define TF_BYPASS               01000   /* bypass label processing */
#define TF_BYPASS_DEFAULT       02000   /* use default value */
#define TF_QUEUEALLOC           04000   /* queue the drive alloc */
```

tpqueue() returns the head of a linked list of requests that are queued in the tape subsystem. The file *tape.h* defines the *queue_t* structure:

```
/* list of queued & active tape requests */
struct queue_t {
    char *hostname;             /* host where drive is located */
    char *drive;                /* name of the drive including unit */
                                /* number */
    char *actualdevice;         /* real tape device name */
    char *userdevice;           /* user device name (eg, /dev/lt/u0) */
    int uid;                    /* user uid */
    char *vsn;                  /* tape vsn */
    char *sym_name;             /* symbolic link name */
    struct queue_t *next;       /* pointer to next item in linked list */
};
```

tpunmount() performs the same function as *tpunmount*(1). If keeponline is non-zero, then the tape is not taken offline. This is equivalent to the -k option to *tpunmount*(1).

tpattr() performs the same function as *tpattr*(1).

The blocksize, recordsize, and format values are changed if their value is non-zero. Otherwise, the value is left unchanged. The fileident is changed if its value is not NULL and the string length is greater than zero. The access and delim values are changed if their value is not -1. The file *tape.h* defines several contents that can be used to set the access and delimiter values:

```
/* tpattr access bits */
#define TO_READ          04     /* others can read */
#define TO_WRITE         02     /* others can write */

/* tpattr delimiter options */
#define TD_BLOCKNL       0      /* block-newline delimiter */
#define TD_NEWLINE       1      /* newline delimiter */
#define TD_SYSCALL       2      /* system call delimiter */

/* tpattr format options   */
#define TF_FIXED              'F'  /* Fixed Record  */
#define TF_DEC_VAR            'D'  /* Decimal Encoded Variable length ansi only */
#define TF_BIN_VAR            'V'  /* Binary Encoded Variable length IBM only */
#define TF_BIN_VAR_SPN        'v'  /* Variable length spanned IBM only*/
#define TF_UNDEFIND           'U'  /* Undefined record/block format*/
```

tplabel() performs the same function as *tplabel*(1). The access argument can be either space or non-space. A space indicates that others are allowed to access this tape. Non-space indicates that others cannot access this tape.

tpunlabel() performs the same function as *tpunlabel*(1).

tplist() performs the same function as *tplist*(1). *tplist* returns a socket descriptor which must be read (until EOF) to get the label information. The format of the socket data is the same as the output of *tplist*(1). The file *tape.h* defines several constants that can be used to set the labselect and flags values:

```
/* tplist label select options */
#define TLS_DETAIL     001    /* detailed info on labels */
#define TLS_VOL        002    /* just volume info */
#define TLS_FILE       004    /* just file info */

/* tplist flag options */
#define TLF_RAW        001    /* display raw labels (with newlines) */
```

tpwait() performs the same function as *tpwait*(1).

tpstatus() will check the status of a mount. The cwd parameter is the directory of the symname or NULL. If cwd is NULL, the current working directory is used. symname is the symbolic link name given in the tpmount command, NULL, or TAPE. The return code given is the status of the mount:

-1 the mount is still pending,
0 the mount is successful,
>0 the mount has failed.

The error code returned is one of the TE_* codes defined in tape.h. Once a tpstatus() call is made tpdaemon will send a SIGALRM signal to the process when the mount completes.

tperror() prints a short message on the standard error file describing the last tape error.

RETURN VALUES

For *tpqueue*(), NULL is returned if the call fails. For all other calls, -1 is returned if the call fails. When a call fails, an error code if left in the global structure *tp_error*. *tperror*() can be used to interpret this error code.

SEE ALSO

tpmount(1), tpunmount(1), tpqueue(1), tpattr(1), tplabel(1), tpunlabel(1), tplist(1), tpwait(1)

NAME

tas – indivisibly test and set a memory location

SYNOPSIS

tas(addr)
caddr_t addr;

DESCRIPTION

tas indivisibly tests and sets the memory location *addr*. The return value represents the previous value of the location.

If the return value is zero, you have acquired the lock. If the return value is non-zero, sleep and try again.

SEE ALSO

mset(3), msleep(2)

NAME

tcgetattr, tcsetattr - get/set terminal characteristics

SYNOPSIS

#include <termios.h>

tcgetattr(fd, termios_p)
int fd;
struct termios *termios_p;

tcsetattr(fd, optional_action, termios_p)
int fd;
int optional_actions;
struct termios *termios_p;

DESCRIPTION

tcgetattr() gets the parameters associated with the object referred to by *fd* and stores them in the termios structure referenced by *termios_p*. This function is allowed from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

The *tcsetattr()* function sets the parameters associated with the terminal from the termios structure referenced by *termios_p*.

optional_action determines when the change will occur as follows. If *optional_action* is TCSANOW, the change will occur immediately. If *optional_action* is TCSADRAIN, the change will occur after output written to *fd* has been transmitted to the terminal. This action should be used when changing parameters that affect output. If *optional_action* is TCSAFLUSH, the change will occur after all output has drained, and all input that has been received but not read will be discarded before the change.

TCSANOW, TCSADRAIN, and TCSAFLUSH are defined in **<termios.h>**.

RETURNS

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

ERRORS

If any of the following conditions occur, the function will return -1 and set *errno* to the corresponding value:

[EBADF]	The *fd* argument is not a valid file descriptor.
[EINVAL]	The *optional_action* argument is not a proper value.
[ENOTTY]	The file associated with *fd* is not a terminal.

SEE ALSO

cfgetispeed(3), cfsetispeed(3), cfgetospeed(3), cfsetospeed(3), tcgetpgrp(3), tcsetpgrp(3), termios(4).

NAME

tcgetpgrp, tcsetpgrp - get/set terminal process group

SYNOPSIS

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

pid_t tcgetpgrp(fd)
int fd;

tcsetpgrp(fd, pgrp_id)
int fd;
pid_t pgrp_id;
```

DESCRIPTION

tcgetpgrp() returns the process group ID of the foreground process group associated with the terminal. The file associated with *fd* either must be the controlling terminal of the calling process or must be the master side of a pty device pair.

The *tcsetpgrp*() function sets the foreground process group ID associated with the terminal to *pgrp_id*. The file associated with *fd* must be the controlling terminal of the calling process, and the controlling terminal must currently be associated with the session of the calling process. The value of *pgrp_id* must match a process group ID of a process in the same session as the calling process.

RETURNS

tcgetpgrp() returns the process group ID of the foreground process group if successful; otherwise a value of –1 is returned and *errno* is set to indicate the error.

tcsetpgrp() returns a value of zero upon successful completion; otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, the function will return –1 and set *errno* to the corresponding value:

[EBADF]	The *fd* argument is not a valid file descriptor.
[EINVAL]	The value of the *pgrp_id* argument is invalid.
[ENOTTY]	The file associated with *fd* is neither the controlling terminal of the calling process nor the master side of a pty device pair (see *pty*(4)).
[EPERM]	The value of the *pgrp_id* does not match the process group ID of a process in the same session as the calling process.

SEE ALSO

intro(2), cfgetospeed(3), cfsetospeed(3), tcgetattr(3), tcsetattr(3), termios(4), pty(4), tty(4).

NAME

tcsendbreak, tcdrain, tcflush, tcflow - terminal line control functions

SYNOPSIS

```
#include <termios.h>

tcsendbreak(fd, duration)
int fd;
int duration;

tcdrain(fd)
int fd;

tcflush(fd, queue_selector)
int fd;
int queue_selector;

tcflow(fd, action)
int fd;
int action;
```

DESCRIPTION

If the terminal is using asynchronous serial data transmission, the *tcsendbreak()* function will transmit a break for a specific duration. If *duration* is zero, the break is transmitted for 0.25 seconds. If *duration* is non-zero it will be interpreted as the duration of the break in 0.1 second increments.

The *tcdrain()* function will wait until all output written to the object referred to by *fd* has been transmitted to the terminal.

The *tcflush()* function will discard data written to the object referred to by *fd* but not transmitted or data received but not read, depending on the value of *queue_selector*. If *queue_selector* is TCIFLUSH, data received but not read will be flushed. If *queue_selector* is TCOFLUSH, data written but not transmitted will be flushed. If *queue_selector* is TCIOFLUSH, both data received but not read and data written but not transmitted will be flushed.

The *tcflow()* function will suspend transmission or reception of data on the object referred to by *fd*, depending on the value of *action*. If *action* is TCOOFF, it will suspend output. If *action* is TCOON, it will restart suspended output. If *action* is TCIOFF, the system will transmit a STOP character, which is intended to cause the terminal device to stop transmitting data to the system. If *action* is TCION, the system will transmit a START character, which is intended to cause the terminal device to start transmitting data to the system.

Values of *queue_selector* and *action* are defined in **<termios.h>**.

RETURNS

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

ERRORS

If any of the following conditions occur, the function will return -1 and set *errno* to the corresponding value:

[EBADF]	The *fd* argument is not a valid file descriptor.
[EINTR]	The signal interrupted the *tcdrain()* function.
[EINVAL]	The *queue_selector* or *action* argument is not a proper value.
[ENOTTY]	The file associated with *fd* is not a terminal.

SEE ALSO

tcsetattr(3).

NAME

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

SYNOPSIS

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

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

tgetnum(id)
char *id;

tgetflag(id)
char *id;

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

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

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

DESCRIPTION

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

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

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

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

Tputs decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by *stty*(3). The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null (**^@**) is inappropriate.

FILES

/usr/lib/libtermcap.a	–ltermcap library
/etc/termcap	data base

SEE ALSO

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

AUTHOR

William Joy

NAME

time - get system time

SYNOPSIS

#include <time.h>

time_t time(time_t *tloc);

DESCRIPTION

The *time()* function returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. If *tloc* is nonnull, the return value is also stored in the place to which *tloc* points.

RETURN VALUE

The *time()* function always succeeds and returns the value described above.

SEE ALSO

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

NAME

times – get process times

SYNOPSIS

#include <sys/times.h>

clock_t times(buffer)
struct tms *buffer;

DESCRIPTION

times() returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in **CLK_TCK**ths of a second.

The members of *struct tms* are:

tms_utime	User CPU time.
tms_stime	System CPU time.
tms_cutime	User CPU time of terminated child processes.
tms_cstime	System CPU time of terminated child processes.

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

RETURN VALUE

Upon success, the elapsed real time in **CLK_TCK**ths of a second, since system startup time is returned. This point does not change from one invocation of *times()* within the process to another. The return value may overflow the possible range of type *clock_t*. If the times function fails, a value of ((*clock_t*)–1) is returned, and *errno* is set to indicate the error.

ERRORS

It is unlikely that *times()* will fail. Any possible failures come from *getrusage(2)*.

BACKWARD COMPATIBILITY

Previous versions of *times()* filled the *struct tms* members with units measured in 60ths of a second. **CLK_TCK**ths are not necessarily the same.

Previous versions of *times()* always returned zero or –1.

SEE ALSO

time(1), getrusage(2), wait(2), time(3c)

NAME

tmpfile, tmpnam – create a temporary file or generate a unique file name.

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void);

char *tmpnam(char *buffer);
```

DESCRIPTION

tmpfile creates a temporary file on */tmp*. This file is opened in mode "w+". The file is removed when it is closed; this occurs explicitly by calling *fclose* or implicitly by calling *exit*. If the program terminates abnormally, the file may not be removed.

tmpnam generates a string that is a unique file name. If *buffer* is not NULL, the string will be stored in *buffer;* otherwise it will be stored in a static buffer internal to *tmpnam*. Subsequent calls to *tmpnam* will overwrite this static buffer. If *buffer* is not NULL, it must identify an array of at least L_tmpnam characters. Each of the first TMP_MAX calls to *tmpnam* will generate different strings.

RETURNS

tmpfile returns a pointer to the FILE structure for the open file. If the file cannot be opened, a null pointer is returned.

tmpnam returns a pointer to the string.

SEE ALSO

fopen(3s).

NAME

ttyname, isatty, ttyslot – find name of a terminal

SYNOPSIS

char *ttyname(filedes)
int filedes;

isatty(filedes)
int filedes;

ttyslot()

DESCRIPTION

ttyname returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *filedes* (this is a system file descriptor and has nothing to do with the standard I/O FILE typedef).

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

ttyslot returns the number of the slot in the */etc/utmp* file corresponding to the current user, 0 if no such slot is found.

FILES

/dev/*
/etc/ttys
/etc/utmp

SEE ALSO

ioctl(2), ttys(5), utmp(5)

DIAGNOSTICS

ttyname returns a null pointer (0) if *filedes* 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

tzset – set time zone information

SYNOPSIS

#include <time.h>

void tzset()

DESCRIPTION

The *tzset()* function uses the value of the environment variable **TZ** to set time conversion information used by *localtime(), ctime(), strftime(),* and *mktime().* If **TZ** is absent from the environment, implementation-defined default time zone information is used.

The *tzset()* function sets the external variable *tzname:*

extern char *tzname[2] = { "*std*", "*dst*" };

where *std* and *dst* are the standard and summer time zone names defined form the environment variable **TZ**.

The value of **TZ** has the following form (spaces have been inserted for clarity):

std offset dst offset, rule

Where:

std and *dst* are three or more byte sequences that are the designation for the standard (*std*) or summer (*dst*) time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon (:), digits, comma(,), minus(-), plus(+), and ASCII NULL are allowed.

offset indicates the value one must add to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

hh[:mm[:ss]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst,* summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour shall be between zero and 24, and the minutes (and seconds) - if present - between zero and 59. Out of range values will cause unpredictable behavior. If preceded by a '-', the time zone is east of the Prime Meridian; otherwise it is west (which may be indicated by an optional preceding '+').

Rule indicated when to change to and back from summer time. The *rule* has the form:

date/time,date/time

where the first *date* describes when the change from standard to summer occurs, and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made. The format of *date* shall be one of the following:

J*n*, the Julian day *n* (one based). Because leap days are not counted, it is impossible to refer to February 29.

n, The zero-based Julian day. Because leap days are counted, it is possible to refer to February 29.

M*m.n.d*, day *d* of week *n* of month *m*. Week *n* is the *n*th week in which day *d* appears. Week 5 is always the last week. Sunday is day zero.

The *time* has the same format as *offset* except that no leading sign ('-' or '+') is allowed. The default, if *time* is not given, is 02:00:00.

NAME

ungetc – push character back into input stream

SYNOPSIS

#include <stdio.h>

int ungetc(int c, FILE *stream)

DESCRIPTION

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

One character of pushback is guaranteed from the stream. Attempts to push EOF are rejected.

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

SEE ALSO

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

DIAGNOSTICS

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

NAME

utime – set file times

SYNOPSIS

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

int utime(file, timep)
char *file;
struct utimbuf *timep;
```

DESCRIPTION

The *utime()* function uses the *actime* and *modtime* members of the *utimbuf* structure pointed to by *timep* to set the recorded times for *file*. If *timep* is NULL, the current time is used.

The caller's effective uid must match the owner of the file, or the caller must be the super-user. The "inode-changed" time of the file is set to the current time.

RETURNS

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

ERRORS

utime() fails if:

[EACCES]	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]	A component of the path prefix denies search permission.
[EFAULT]	The *file* argument points outside the process's allocated address space.
[EINVAL]	The pathname contained a character with the high-order bit set.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[ENAMETOOLONG]	The pathname was too long.
[ENIENT]	The pathname argument pointed to an empty string.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The process is not super-user and not the owner of the file.
[EROFS]	The file system containing the file is mounted read-only.

SEE ALSO

utimes(2), stat(2)

NAME

valloc – aligned memory allocator

SYNOPSIS

char *valloc(size)
unsigned size;

DESCRIPTION

Valloc allocates *size* bytes aligned on a page boundary. It is implemented by calling *malloc*(3) with a slightly larger request, saving the true beginning of the block allocated, and returning a properly aligned pointer.

DIAGNOSTICS

Valloc returns a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block.

BUGS

Vfree isn't implemented.

NAME

varargs – variable argument list

SYNOPSIS

#include <varargs.h>

function(**va_alist**)
va_dcl
va_list *pvar*;
va_start(*pvar*);
f = **va_arg**(*pvar*, *type*);
va_end(*pvar*);

DESCRIPTION

This set of macros provides a means of writing portable procedures that accept variable argument lists. Routines having variable argument lists (such as *printf*(3)) that do not use varargs are inherently nonportable, since different machines use different argument passing conventions.

va_alist is used in a function header to declare a variable argument list.

va_dcl is a declaration for **va_alist**. Note that there is no semicolon after **va_dcl.**

va_list is a type which can be used for the variable *pvar*, which is used to traverse the list. One such variable must always be declared.

va_start(pvar) is called to initialize *pvar* to the beginning of the list.

va_arg(*pvar*, *type*) 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, since it cannot be determined at runtime.

va_end(*pvar*) is used to finish up.

Multiple traversals, each bracketed by **va_start** ... **va_end,** are possible.

EXAMPLE

```
#include <varargs.h>
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[100];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while (args[argno++] = va_arg(ap, char *))
                ;
        va_end(ap);
        return execv(file, args);
}
```

BUGS

It is up to the calling routine to determine how many arguments there are, since it is not possible to determine this from the stack frame. For example, *execl* passes a 0 to signal the end of the list. *Printf* can tell how many arguments are supposed to be there by the format.

NAME

vlimit – control maximum system resource consumption

SYNOPSIS

#include <sys/vlimit.h>

vlimit(resource, value)
int resource, value;

DESCRIPTION

This facility is superseded by getrlimit(2).

Limits the consumption by the current process and each process it creates to not individually exceed *value* on the specified *resource*. If *value* is specified as –1, then the current limit is returned and the limit is unchanged. The resources which are currently controllable are:

LIM_NORAISE A pseudo-limit; if set non-zero then the limits may not be raised. Only the super-user may remove the *noraise* restriction.

LIM_CPU the maximum number of cpu-seconds to be used by each process

LIM_FSIZE the largest single file which can be created

LIM_DATA the maximum growth of the data+stack region via *sbrk*(2) beyond the end of the program text

LIM_STACK the maximum size of the automatically-extended stack region

LIM_CORE the size of the largest core dump that will be created.

LIM_MAXRSS a soft limit for the amount of physical memory (in bytes) to be given to the program. If memory is tight, the system will prefer to take memory from processes which are exceeding their declared LIM_MAXRSS.

Because this information is stored in the per-process information this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to *csh*(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way; a *break* call fails if the data space limit is reached, or the process is killed when the stack limit is reached (since the stack cannot be extended, there is no way to send a signal!).

A file i/o operation which would create a file which is too large will cause a signal SIGXFSZ to be generated, this normally terminates the process, but may be caught. When the cpu time limit is exceeded, a signal SIGXCPU is sent to the offending process; to allow it time to process the signal it is given 5 seconds grace by raising the cpu time limit.

SEE ALSO

csh(1), getrlimit(2)

BUGS

If LIM_NORAISE is set, then no grace should be given when the cpu time limit is exceeded.

There should be *limit* and *unlimit* commands in *sh*(1) as well as in *csh*.

The options and specifications of this system call and even the call itself are subject to change. It may be extended or replaced by other facilities in future versions of the system.

NAME

vprintf, vfprintf, vsprintf – stdarg style formatted output conversion

SYNOPSIS

#include <stdio.h>

#include <stdarg.h>

int vprintf(const char *format, va_list arg);

int vfprintf(FILE *stream, const char *format, va_list arg);

int vsprintf(char *s, const char *format, va_list arg);

DESCRIPTION

vprintf, vfprintf, and *vsprintf* are alternate forms of *printf, fprintf,* and *sprintf. vprintf, vfprintf,* and *vsprintf* require a single argument of type *va_list* rather than a variable number of arguments. This argument must be initialized with the *va_start* macro provided in stdarg.h.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- None of these routines are available in the backward-compatible mode.

SEE ALSO

stdarg.h(3) printf(3s)

NAME

vtimes – get information about resource utilization

SYNOPSIS

vtimes(par_vm, ch_vm)
struct vtimes *par_vm, *ch_vm;

DESCRIPTION

This facility is superseded by getrusage(2).

Vtimes returns accounting information for the current process and for the terminated child processes of the current process. Either *par_vm* or *ch_vm* or both may be 0, in which case only the information for the pointers which are non-zero is returned.

After the call, each buffer contains information as defined by the contents of the include file */usr/include/sys/vtimes.h:*

```
struct vtimes {
        int     vm_utime;               /* user time (*HZ) */
        int     vm_stime;               /* system time (*HZ) */
        /* divide next two by utime+stime to get averages */
        unsigned vm_idsrss;             /* integral of d+s rss */
        unsigned vm_ixrss;              /* integral of text rss */
        int     vm_maxrss;              /* maximum rss */
        int     vm_majflt;              /* major page faults */
        int     vm_minflt;              /* minor page faults */
        int     vm_nswap;               /* number of swaps */
        int     vm_inblk;               /* block reads */
        int     vm_oublk;               /* block writes */
};
```

The *vm_utime* and *vm_stime* fields give the user and system time respectively in 60ths of a second (or 50ths if that is the frequency of wall current in your locality.) The *vm_idrss* and *vm_ixrss* measure memory usage. They are computed by integrating the number of memory pages in use each over cpu time. They are reported as though computed discretely, adding the current memory usage (in 4096 byte pages) each time the clock ticks. If a process used 5 core pages over 1 cpu-second for its data and stack, then *vm_idsrss* would have the value 5*60, where *vm_utime+vm_stime* would be the 60. *Vm_idsrss* integrates data and stack segment usage, while *vm_ixrss* integrates text segment usage. *Vm_maxrss* reports the maximum instantaneous sum of the text+data+stack core-resident page count.

The *vm_majflt* field gives the number of page faults which resulted in disk activity; the *vm_minflt* field gives the number of page faults incurred in simulation of reference bits; *vm_nswap* is the number of swaps which occurred. The number of file system input/output events are reported in *vm_inblk* and *vm_oublk* These numbers account only for real i/o; data supplied by the caching mechanism is charged only to the first process to read or write the data.

SEE ALSO

getrusage(2), wait(2)

BUGS

This call is peculiar to the Berkeley version of UNIX. The options and specifications of this system call are subject to change. It may be extended to include additional information in future versions of the system. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

Section 4
Special files/driver functions/ networking support

NAME

intro – introduction to special files and hardware support

DESCRIPTION

This section describes the special files, related driver functions, and networking support available in the system.

DEVICE SUPPORT

This section describes the hardware supported on the Convex computer. Software support for these devices comes in two forms. A hardware device may be supported with a character or block *device driver*. Block and character devices are accessed through files in the file system of a special type; c.f. *mknod*(8).

A hardware device is identified to the system at configuration time and the appropriate device driver is then compiled into the system. When the resultant system is booted, the autoconfiguration facilities in the system probe for the device and enable the software support for it.

The autoconfiguration system is described in *autoconf*(4). A list of the supported devices is given below.

SEE ALSO

autoconf(4), ioconfig(4)

LIST OF DEVICES

The devices listed below are supported in this incarnation of the system. Devices are indicated by their functional interface.

ca(4)	Multibus/VMEbus Asynchronous communications interface
da(4)	Multibus Mass storage disk interface
dm(4)	Multibus DMA interface to Raster Technology Model/One 80 (Optional)
ex(4)	Excelan Ethernet network interface
hy(4)	Multibus HYPERchannel network interface
ik(4)	Multibus DR11-W network interface
pa(4)	Multibus/VMEbus Line printer interface
pb(4)	Multibus Versatec plotter interface
ta(4)	Multibus/VMEbus Tape drive interface
dd(4)	VME Dual SMD/ESDI Mass storage disk interface
du(4)	IDC IPI-2 Mass storage disk interface

NAME

networking – introduction to networking facilities

SYNOPSIS

```
#include <sys/socket.h>
#include <net/route.h>
#include <net/if.h>
```

DESCRIPTION

This section briefly describes the networking facilities available in the system. Documentation in this part of section 4 is broken up into three areas: *protocol families* (domains), *protocols*, and *network interfaces*. Entries describing a protocol family are marked "4F," while entries describing protocol use are marked "4P." Hardware support for network interfaces are found among the standard "4" entries.

All network protocols are associated with a specific *protocol family*. A protocol family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol family may support multiple methods of addressing, though the current protocol implementations do not. A protocol family is normally comprised of a number of protocols, one per *socket*(2) type. It is not required that a protocol family support all socket types. A protocol family may contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in *socket*(2). A specific protocol may be accessed either by creating a socket of the appropriate type and protocol family, or by requesting the protocol explicitly when creating a socket. Protocols normally accept only one type of address format, usually determined by the addressing structure inherent in the design of the protocol family/network architecture. Certain semantics of the basic socket abstractions are protocol specific. All protocols are expected to support the basic model for their particular socket type, but may, in addition, provide non-standard facilities or extensions to a mechanism. For example, a protocol supporting the SOCK_STREAM abstraction may allow more than one byte of out-of-band data to be transmitted per out-of-band message.

A network interface is similar to a device interface. Network interfaces comprise the lowest layer of the networking subsystem, interacting with the actual transport hardware. An interface may support one or more protocol families and/or address formats. The SYNOPSIS section of each network interface entry gives a sample specification of the related drivers for use in providing a system description to the *ioconfig* file on the SPU. The DIAGNOSTICS section lists messages which may appear on the console and/or in the system error log, */usr/adm/messages* (see *syslogd*(8)), due to errors in device operation.

PROTOCOLS

The system currently includes the DARPA Internet protocols and the Xerox Network Systems(tm) protocols. Raw socket interfaces are provided to the IP protocol layer of the DARPA Internet, to the IMP link layer (1822), and to the IDP protocol of Xerox NS. The DARPA Internet protocols are currently the only supported protocols. Consult the appropriate manual pages in this section for more information regarding the support for each protocol family.

ADDRESSING

Associated with each protocol family is an address format. The following address formats are used by the system (and additional formats are defined for possible future implementation):

```
#define AF_UNIX      1    /* local to host (pipes, portals) */
#define AF_INET      2    /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK   3    /* arpanet imp addresses */
#define AF_PUP       4    /* pup protocols: e.g. BSP */
#define AF_NS        6    /* Xerox NS protocols */
```

```
#define  AF_HYLINK        15      /* NSC Hyperchannel */
```

The address types, AF_IMPLINK, AF_PUP, AF_NS, and AF_HYLINK are not currently supported.

ROUTING

The network facilities provided limited packet routing. A simple set of data structures comprise a "routing table" used in selecting the appropriate network interface when transmitting packets. This table contains a single entry for each route to a specific network or host. A user process, the routing daemon, maintains this data base with the aid of two socket-specific *ioctl*(2) commands, SIOCADDRT and SIOCDELRT. The commands allow the addition and deletion of a single routing table entry, respectively. Routing table manipulations may only be carried out by super-user.

A routing table entry has the following form, as defined in *<net/route.h>*;

```
struct rtentry {
        u_long      rt_hash;
        struct      sockaddr rt_dst;
        struct      sockaddr rt_gateway;
        short       rt_flags;
        short       rt_refcnt;
        u_long      rt_use;
        struct      ifnet *rt_ifp;
};
```

with *rt_flags* defined from,

```
#define  RTF_UP          0x1     /* route usable */
#define  RTF_GATEWAY     0x2     /* destination is a gateway */
#define  RTF_HOST        0x4     /* host entry (net otherwise) */
#define  RTF_DYNAMIC     0x10    /* created dynamically (by redirect) */
```

Routing table entries come in three flavors: for a specific host, for all hosts on a specific network, for any destination not matched by entries of the first two types (a wildcard route). When the system is booted and addresses are assigned to the network interfaces, each protocol family installs a routing table entry for each interface when it is ready for traffic. Normally the protocol specifies the route through each interface as a "direct" connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface is requested to address the packet to the gateway listed in the routing entry (i.e. the packet is forwarded).

Routing table entries installed by a user process may not specify the hash, reference count, use, or interface fields; these are filled in by the routing routines. If a route is in use when it is deleted (*rt_refcnt* is non-zero), the routing entry will be marked down and removed from the routing table, but the resources associated with it will not be reclaimed until all references to it are released. The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existent entry, or ENOBUFS if insufficient resources were available to install a new route. User processes read the routing tables through the */dev/kmem* device. The *rt_use* field contains the number of packets sent along the route.

When routing a packet, the kernel will first attempt to find a route to the destination host. Failing that, a search is made for a route to the network of the destination. Finally, any route to a default ("wildcard") gateway is chosen. If multiple routes are present in the table, the first route found will be used. If no entry is found, the destination is declared to be unreachable.

A wildcard routing entry is specified with a zero destination address value. Wildcard routes are used only when the system fails to find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing

traffic.

INTERFACES

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface, *lo*(4), do not.

The following *ioctl* calls may be used to manipulate network interfaces. The *ioctl* is made on a socket (typically of type SOCK_DGRAM) in the desired domain. Unless specified otherwise, the request takes an *ifrequest* structure as its parameter. This structure has the form

```
struct  ifreq {
        char    ifr_name[16];           /* name of interface (e.g. "ec0") */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                struct  sockaddr ifru_broadaddr;
                short   ifru_flags;
                int     ifru_metric;
        } ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr         /* address */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr     /* other end of p-to-p link */
#define ifr_broadaddr   ifr_ifru.ifru_broadaddr   /* broadcast address */
#define ifr_flags ifr_ifru.ifru_flags       /* flags */
#define ifr_metric      ifr_ifru.ifru_metric      /* routing metric */
};
```

SIOCSIFADDR
: Set interface address for protocol family. Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR
: Get interface address for protocol family.

SIOCSIFDSTADDR
: Set point to point address for protocol family and interface.

SIOCGIFDSTADDR
: Get point to point address for protocol family and interface.

SIOCSIFBRDADDR
: Set broadcast address for protocol family and interface.

SIOCGIFBRDADDR
: Get broadcast address for protocol family and interface.

SIOCSIFFLAGS
: Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified; some interfaces may be reset so that incoming packets are no longer received. When marked up again, the interface is reinitialized.

SIOCGIFFLAGS
: Get interface flags.

SIOCSIFMETRIC
: Set interface routing metric. The metric is used only by user-level routers.

SIOCGIFMETRIC
: Get interface metric.

SIOCGIFCONF
: Get interface configuration list. This request takes an *ifconf* structure (see below) as a

value-result parameter. The *ifc_len* field should be initially set to the size of the buffer pointed to by *ifc_buf*. On return it will contain the length, in bytes, of the configuration list.

```
/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration
 * for machine (useful for programs which
 * must know all networks accessible).
 */
struct  ifconf {
        int     ifc_len;        /* size of associated buffer */
        union {
                caddr_t ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
#define ifc_buf  ifc_ifcu.ifcu_buf /* buffer address */
#define ifc_req  ifc_ifcu.ifcu_req /* array of structures returned */
};
```

SEE ALSO

socket(2), config(8), ifconfig(8), ioctl(2), intro(4), routed(8C)

NOTES

Networking is an optional product; for more information, contact your CONVEX sales representative.

NAME

arp – Address Resolution Protocol

SYNOPSIS

pseudo-device ether

DESCRIPTION

ARP is a protocol used to dynamically map between DARPA Internet and 10Mb/s Ethernet addresses. It is used by both of the 10Mb/s Ethernet interface drivers, ex(4) and ve(4). It is not specific to Internet protocols or to 10Mb/s Ethernet, but this implementation currently supports only that combination.

ARP caches Internet-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending message is then transmitted. ARP will queue at most one packet while waiting for a mapping request to be responded to; only the most recently "transmitted" packet is kept.

To facilitate communications with systems which do not use ARP, *ioctl*s are provided to enter and delete entries in the Internet-to-Ethernet tables. Usage:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
struct arpreq arpreq;

ioctl(s, SIOCSARP, (caddr_t)&arpreq);
ioctl(s, SIOCGARP, (caddr_t)&arpreq);
ioctl(s, SIOCDARP, (caddr_t)&arpreq);
```

Each ioctl takes the same structure as an argument. SIOCSARP sets an ARP entry, SIOCGARP gets an ARP entry, and SIOCDARP deletes an ARP entry. These ioctls may be applied to any socket descriptor *s*, but only by the super-user. The *arpreq* structure contains:

```
/*
 * ARP ioctl request
 */
struct arpreq {
        struct sockaddr         arp_pa; /* protocol address */
        struct sockaddr         arp_ha; /* hardware address */
        int                     arp_flags;/* flags */
};
/*  arp_flags field values */
#define ATF_COM                 0x02    /* completed entry  */
#define  ATF_PERM 0x04          /* permanent entry */
#define  ATF_PUBL 0x08          /* publish (respond for other host) */
#define  ATF_USETRAILERS 0x10   /* send trailer packets to host */
```

The address family for the *arp_pa* sockaddr must be AF_INET; for the *arp_ha* sockaddr it must be AF_UNSPEC. The only flag bits which may be written are ATF_PERM, ATF_PUBL and ATF_USETRAILERS. ATF_PERM causes the entry to be permanent if the ioctl call succeeds. The peculiar nature of the ARP tables may cause the ioctl to fail if more than 8 (permanent) Internet host addresses hash to the same slot. ATF_PUBL specifies that the ARP code should respond to ARP requests for the indicated host coming from other machines. This allows a host to act as an "ARP server," which may be useful in convincing an ARP-only machine to talk to a non-ARP machine.

ARP is also used to negotiate the use of trailer IP encapsulations; trailers are an alternate encapsulation used to allow efficient packet alignment for large packets despite variable-sized headers. Hosts which wish to receive trailer encapsulations so indicate by sending gratuitous ARP translation replies along with replies to IP requests; they are also sent in reply to IP translation replies. The negotiation is thus fully symmetrical, in that either or both hosts may request trailers. The ATF_USETRAILERS flag is used to record the receipt of such a reply, and enables the transmission of trailer packets to that host.

ARP watches passively for hosts impersonating the local host (i.e. a host which responds to an ARP mapping request for the local host's address).

DIAGNOSTICS

duplicate IP address!! sent from ethernet address: %x:%x:%x:%x:%x:%x. ARP has discovered another host on the local network which responds to mapping requests for its own Internet address.

SEE ALSO

ex(4), ve(4), inet(4F), arp(8C), ifconfig(8C)
"An Ethernet Address Resolution Protocol," RFC826, Dave Plummer, Network Information Center, SRI.
"Trailer Encapsulations," RFC893, S.J. Leffler and M.J. Karels, Network Information Center, SRI.

BUGS

ARP packets on the Ethernet use only 42 bytes of data; however, the smallest legal Ethernet packet is 60 bytes (not including CRC). Some systems may not enforce the minimum packet size, others will.

NOTES

Arp is an optional product; for more information, contact your CONVEX sales representative.

NAME

autoconf – diagnostics from the autoconfiguration code

DESCRIPTION

When the SPU OS bootstraps, the SPU OS file */ioconfig* is used to specify the controllers, drives, and other devices present and their hardware parameters. Devices are probed to see if their control-status registers respond. If not, they are ignored. Each device prints its configuration parameters on the console; these messages are summarized in the SYNOPSIS paragraph of each device.

SEE ALSO

intro(4)

NAME

ca – terminal multiplexor

SYNOPSIS

ctlr ACM-001 csr 0x%x int %d
unit %d type TTY

ctlr ACM-201 csr 0x%x int %d
unit %d type TTY
unit 16 type PRT-CEN

ctlr ACM-201 csr 0x%x int %d
unit %d type TTY
unit 16 type PRT-DAT

DESCRIPTION

The *ca* device provides 16 lines of asynchronous serial line support with full modem control. ACM-001 is the Multibus controller and ACM-201 is the VMEbus controller. The VMEbus controller also provides a line printer interface on port 16 as described in *pa*(4).

Each tty line attached to a serial line port behaves as described in *tty*(4). Transmission speeds for each line may independently be set to run at any of 16 rates; see *tty*(4) for the encoding.

FILES

/dev/tty[0-9a-f][0-9a-f]

SEE ALSO

tty(4), pa(4)

DIAGNOSTICS

Notations:

%d	expands to a decimal number
%x	expands to a hexadecimal number
%s	expands to a text string

It is possible to get many error messages from the ca driver. Many are for *impossible* error conditions which can *never* occur in released systems. Nearly all of the *impossible* class of messages will immediately be followed by a CCU crash. No attempt will be made to fully document all possible messages. If you need help interpreting an error message that is not in the lists which follow, please contact the CONVEX Technical Assistance Center.

The following messages may appear as the system is being booted:

ca: tty controller limit of %d exceeded

or

ca: "number_tty_controllers" tuned only to %d
The file */ioconfig* contains too many ACM controller entries for the system being booted. If you see this message, you are probably attempting to violate the licensing agreements for maximum number of users. Contact the CONVEX Technical Assistance Center for upgrade information.

ca: ACM-001 Multibus %d controller is not present at CSR %x

or

ca: ACM-201 VMEbus %d controller is not present at CSR %x
The file */ioconfig* contains an ACM controller entry for a controller that does not exist at the specified CSR address.

ca: ACM-201 VMEbus %d CSR %x failed its self test

or

ca: ACM-201 VMEbus %d CSR %x eprom not responding
The controller is broken. Contact CONVEX Field Service.

ca: last probe failed, attach ABORT

or

ca: unable to send attach message, error=%d
These errors usually indicate errors in the */ioconfig* file. They could also occur due to software or hardware problems in the communication system between the CPU and CCU.

ca: illegal ACM-001 unit %d type %s

or

ca: illegal ACM-201 unit %d type %s
The file */ioconfig* has an incorrect entry. Either the unit number is bad, it must be in range 0 - 15 for ACM-001 or 0 - 16 for ACM-201 where unit 16 is either "PRT-CEN" or "PRT-DAT", or the type is bad. Currently, only type "TTY" is officially recognized unit type for units 0 - 15. If this error occurs, the system should be shut down. The error in */ioconfig* should be corrected, and the system should be rebooted.

ca: ACM-001 Multibus %d CSR %x ignored attempt to redefine unit %d

or

ca: ACM-201 VMEbus %d CSR %x ignored attempt to redefine unit %d
A duplicate entry for unit **%d** was found in */ioconfig*. The system should be shut down, the error in */ioconfig* should be corrected, and the system rebooted.

The following messages may occur at any time:

tty overrun: Multibus %d CSR %x Port(s) %x %x %x

or

tty overrun: VMEbus %d CSR %x Port(s) %x (%d) %x (%d) %x (%d)
Incoming characters for the specified port(s) are arriving faster than the controller can accept them. Some incoming data has been discarded. In the message from the vme controller the port number is followed by the number of discarded characters detected by the controller. Check for "ringing" terminal lines or other hardware problems. The *tty* picture of the *syspic(8)* program may be helpful in diagnosing the problem.

tty input overflow: VMEbus %d CSR %x Port(s) %x (%d) %x (%d) %x (%d)
Incoming characters for the specified port(s) are arriving from the controller faster than the driver can accept them. The number of incoming characters discarded by the controller follows the port number. Check for "ringing" terminal lines or other hardware problems. The *tty* picture of the *syspic(8)* program may be helpful in diagnosing the problem.

ca: ACM-001 Multibus %d CSR %x port %x command error %x

or

ca: ACM-001 Multibus %d CSR %x illegal request code %x

or

ca: ACM-201 VMEbus %d CSR %x illegal request code %x

The controller has received an illegal command. In many cases, the system will continue with no apparent problems. In other cases, the terminal line in question will hang until the system is rebooted. These errors should be reported to CONVEX, along with any information about the terminal activity that was occurring on the terminal line in question at the time of the error. These errors are generally not cause for alarm, but should be reported at your convenience.

Driver bugs, controller hardware problems or firmware problems may cause the following messages to be displayed on the system console:

ca: ACM-001 Multibus %d CSR %x appears hung

Under normal circumstances, the ACM-001 uses an on-board interval timer to generate periodic interrupts. A period of 30 seconds has elapsed since the last controller interrupt. The controller is assumed to be hung or crashed. The controller will be automatically reset. The driver will attempt to reconfigure the controller and resume tty operations with as little disruption as possible.

ca: ACM-001 Multibus %d CSR %x appears to be broken
Please ask CONVEX Field Service to run dev4300

The controller has reported a hardware problem with a particular port, or has not gone ready within a reasonable period after a board reset command. It is very likely that the controller or USART assembly is broken. CONVEX Field Service should run diagnostics on the controller as soon as convenient.

ca: ACM-001 Multibus %d CSR %x hung or crashed. Resetting controller!

The driver is attempting to issue a command to the controller, and has found that the controller is unwilling to accept the command. The driver will initiate the recovery process mentioned above in the "*appears hung*" message.

ca: ACM-001 Multibus %d CSR %x hung. Status %x PC %x

The driver is attempting to recover from a hung/crashed condition. The PC and status information may be valuable debugging aids to CONVEX personnel, but may otherwise be ignored.

ca: ACM-001 Multibus %d CSR %x revived

The driver has successfully recovered from a fatal controller error, subject to the limitations described in the *TIOCRESET* description in *tty(4)*.

ca: ACM-001 Multibus %d CSR %x recovery attempt failed
ca: controller disabled until reboot

A fatal controller error occurred while trying to recover from a previous fatal controller error. The controller is reset and its interrupt is disabled. The system will continue to run, but the specified controller will be unusable until the next reboot. It is likely that this error indicates a hardware failure. If a reboot does not fix the problem, notify CONVEX Field Service.

ca: ACM-001 Multibus %d CSR %x reset by user

or

ca: ACM-201 VMEbus %d CSR %x reset by user

The specified controller has been reset with the *TIOCRESET* ioctl. See *tty(4)*. Recovery will be attempted, using the procedures mentioned above in the description of "*appears hung*".

NAME

ccu – channel control unit pseudo-device

SYNOPSIS

/dev/ccuN

DESCRIPTION

/dev/ccu*n* is a special file that exists solely to support the adbccu(8) debugger used to debug CCU device drivers by communicating with the kernel of each CCUs via MBS messages to do debugging commands such as single step, set breakpoint, etc.

It provides only the open(2) and close(2) driver entry points primarily to insure that a CCU will be restored to sanity if the debugger is killed, while running. Though the SIGKILL signal cannot be caught, the kernel always closes any open file descriptors, thus ensuring that the /dev/ccuN driver can send a *cleanup* message to the CCU

Only the root user may run adbccu when it opens a */dev/ccu* device.

However, adbccu can be used to examine crashdump files without being root, because the /dev/ccuN device file is not opened.

It may be used, for example, to examine (and even to patch) the CCU program and data.

NOTES

The SPU also has a /dev/ccu device file that is used by the adbccu(8) utility that runs on the SPU. They are commonly used only by device driver and kernel developers.

SEE ALSO

adbccu(8)

BUGS

Since the CCU is halted whenever the *ccu* file is open, it is usually not possible to debug CCUs that are controlling critical devices, such as the user's tty or root file system disk drive.

Since the kernel in the CCU does all breakpoint operations, there are places in the CCU program where breakpoints must not be set. If a breakpoint is set in the message interrupt or clock interrupt code, for example, the CCU will probably crash or hang.

NAME

cons – console interface

DESCRIPTION

The console is available to the processor through the Service Processor (SPU). The console works like any other UNIX terminal except that when the CONVEX computer is in LOCAL or REMOTE MAINTENANCE mode, control-P puts the console in the local console mode (an SPU OS shell with the prompt "(spu)>". Terminating the shell with control-D returns the normal console. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

FILES

/dev/console

SEE ALSO

tty(4), reboot(8)

DIAGNOSTICS

None.

BUGS

The console ignores attempts to change its baud rate.

NAME

ct – cartridge tape interface

DESCRIPTION

This is a simple interface to the Service Processor Unit (SPU) cartridge tape.

DIAGNOSTICS

None.

NAME

da – mass storage disk interface

SYNOPSIS

ctlr DKC-001 csr 0x%x int %d
unit %d type DKD-001 (Fujitsu Eagle M2351A)

ctlr DKC-001 csr 0x%x int %d
unit %d type DKD-002 (CDC 9766)

ctlr DKC-001 csr 0x%x int %d
unit %d type DKD-005 (NEC D2352)

ctlr DKC-001 csr 0x%x int %d
unit %d type DKD-008 (NEC D2363)

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, etc. Minor device numbers are always modulo 256. Therefore, minor numbers for drive 32 wrap back to 0. The major device number for da drives 0 through 31 is 5. Drives 32 through 63 have a major device number of 48. Major da numbers are the same for both block and character devices. The standard device names begin with "da", followed by the drive number, and a letter (**a-h** for partitions **0-7** respectively).

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call generally results in exactly one I/O operation. Therefore, raw I/O is considerably more efficient when the transfer size is very large. The names of the raw files conventionally begin with an extra 'r.'

DISK PARTITIONS

The size of each disk partition is a function of the total size of the disk. The *c* partition always occupies the entire disk, minus 2 or 3 cylinders which are reserved for bad block remapping and diagnostic purposes. The *g* partition occupies the same space as the total space occupied by the *d*, *e*, and *f* partitions. More detailed information about disk partitions may be found in *Managing ConvexOS: Configuration Guide*, and in the file */etc/disktab*.

FILES

/dev/da[0-63][a-h]	block files
/dev/rda[0-63][a-h]	raw files

DIAGNOSTICS

Notations:
%d expands to a decimal number
%x expands to a hexadecimal number
%s expands to an error message string

da%d: controller time out.

A disk operation failed to interrupt in a reasonable amount of time. Operation will be retried.

da%d: %s:%s [cyl=%d hd=%d sect=%d cnt=%d].

An error occurred at the specified disk address. Operation may be retried.

da%d: can't read bad track table.

None of the five copies of the bad sector forwarding table at the end of the disk could be read.

da%d: unrecoverable disk error.

The previous error was unrecoverable after a maximum number of retries.

NAME

dd – VME Dual SMD/ESDI Mass storage disk interface

SYNOPSIS

ctlr DKC-204 csr 0x%x int %d
unit %d type DKD-206 (NEC D2352)

ctlr DKC-204 csr 0x%x int %d
unit %d type DKD-208 (NEC D2363)

ctlr DKC-204 csr 0x%x int %d
unit %d type DKD-281 (SEAGATE 9720)

ctlr DKC-203 csr 0x%x int %d
unit %d type DKD-214 (HITACHI 514-38)

DESCRIPTION

DKC-203 is the ESDI controller(4201) and DKC-204 is for the SMD controller(4200). Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, etc. Minor device numbers are always modulo 256. Therefore, minor numbers for drive 32 wrap back to 0. The major device number for dd drives 0 through 31 is 5. Drives 32 through 63 have a major device number of 48. Major dd numbers are the same for both block and character devices. The standard device names begin with "dd", followed by the drive number, and a letter (**a-h** for partitions **0-7** respectively).

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call generally results in exactly one I/O operation. Therefore, raw I/O is considerably more efficient when the transfer size is very large. The names of the raw files conventionally begin with an extra 'r.'

DISK PARTITIONS

The size of each disk partition is a function of the total size of the disk. The *c* partition always occupies the entire disk, minus the cylinders which are reserved for bad block remapping and diagnostic purposes. The *g* partition occupies the same space as the total space occupied by the *d*, *e*, and *f* partitions. More detailed information about disk partitions may be found in *Managing ConvexOS: Configuration Guide*, and in the file */etc/disktab*.

FILES

/dev/dd[0-63][a-h]	block files
/dev/rdd[0-63][a-h]	raw files

DIAGNOSTICS

Notations:
%d expands to a decimal number
%b expands to a "number<string,string>" output
%s expands to a character string

dd: dev=%d %s:%s [chs=%d/%d/%d]

dd??: %s:%s [chs=%d/%d/%d]

dd: dev=%d EXCP:%b retry=%d [chs=%d/%d/%d]

dd??: EXCP:%b retry=%d [chs=%d/%d/%d]

An error occurred at the specified disk address. Operation may be retried. The errors

with EXCP indicate that the command completed properly, but with error recovery applied as indicated by the %b field.

NAME

dm – Raster Technologies Model One/80 driver

SYNOPSIS

ctlr GPI-001 csr 0x%x int %d
unit %d type GPD-001

DESCRIPTION

The *dm* device supports a parallel interface to a Raster Technologies Model One/80 graphics controller through an IKON Corporation Model 10077 DR11-W Emulator card. The controller number is specified by the minor device number of the /dev entry for the device. Two *ioctl* functions are available.

IKIOCRESET does not require an argument pointer, and will cause a reset of the One/80.

IKIOCTIMER is passed a pointer to an integer containing the time in seconds that the driver will use as a timeout value for DMA operations to the One/80. The default timeout value is 60 seconds.

The *ioctl* commands are defined in *<dev_iop/ikreg.h>*, and can be accessed by including the following include statements in your source code:

```
#include <sys/types.h>
#include <dev_iop/ikreg.h>
#include <sys/ioctl.h>
```

NOTES

The *dm* device is an optional product; for more information, contact your CONVEX sales representative.

NAME

drum – paging device

DESCRIPTION

This file refers to the paging device in use by the system. This may actually be a subdevice of one of the disk drivers, but in a system with paging interleaved across multiple disk drives it provides an indirect driver for the multiple drives.

FILES

/dev/drum

BUGS

Reads from the drum are not allowed across the interleaving boundaries. Since these only occur every .5Mbytes or so, and since the system never allocates blocks across the boundary, this is usually not a problem.

NAME

du – Integrated Disk Controller IPI-2 Logical Level for Disk special file

SYNOPSIS

idc %d
ipi %d
drvr DKC-IP2
unit %d type DKD-502 (Imprimis 6Mb/sec parallel head drive)

DESCRIPTION

The Integrated Disk Controller (IDC) is a PBUS channel control unit designed specifically for very high performance IPI-2 disk units. Each controller supports four IPI-1 ports, numbered 0 through 3. Each IPI port supports 8 slaves, numbered 0 through 7, so the */ioconfig* file entries must specify the idc number, ipi port, driver type and slave number.

The 8.0 Release of ConvexOS supports 32 bit device numbers, with the most significant 12 bits allocated to the major number and the least significant 20 bits allocated to the minor number. The DU file minor number is divided into physical unit number and logical subunit (or more commonly called partition). The subunit (partition) is specified by the lower 8 bits (7-0) and the physical unit is specified in the upper 12 bits. Subunit numbers 0 and 255 are special subunits, not present in the da or dd special files, and are reserved for future use by special diagnostic utilities. These two files are called the *zero* partition and *ctl* device.

Subunit numbers 1 through 8 correspond to partitions *a* through *h*. (Note: this differs from VME and Multibus where the *a* through *h* partitions are numbered 0 through 7, due to the fact that only 3 bits were avialible for the subunit when minor numbers were only 8 bits long). The standard device names begin with "du", followed by the drive number, and a letter (**a-h** for partitions **1-8** respectively). The shell script, */dev/MAKEDEV*, has been modified to create the IDC special files. The 8.O release of ConvexOS is shipped with a / (root) partition containing the special file entries for disk unit 0. The user can create others by becoming root and executing the script.

For example,

MAKEDEV du1 du2 du3

will create the special files for three more IDC units.

The block files access the disk via the system's normal filesystem buffer cache and may be read and written without regard to physical disk records.

Unlike the DA and DD disks, both the geometry and logical partition tables are stored on-disk and read into the DU driver data structures at boot time. This is accomplished using a proprietary disk format. This new format employs a special cylinder, called the topology cylinder, containing a set of data structures initialized during disk formatting. The standard partitions, called *a* through *h* are the default partitions and have the same proportions as their counterparts on VME and Multibus disks. E.g, *a* is the first 5%, *b* the next 20%, etc.

There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. The 'raw' device permits I/O **only** in multiples of the physcially addressible units. Since the IDC disks employ 2048-byte physical sectors, this means that reads and writes must be done by application programs in multiples of 2048. If a partial-sector read or write is attempted, the driver will return the error number for EFAULT (bad address). All ConvexOS disk utilities such as *mkfs(8)*, that access the raw device have been modified to accommodate this restriction. **Note that this means the minimum frag size on IDC partitions is 2K.** This is reflected in the */etc/disktab* defaults for DKD-502, the first disk product released with the IDC. Note that the partition sizes (e.g. :pa#24600:) are in *physical block* units (physical sector), whereas *mkfs(8)* creates filesytems whose sizes are counted in *logical block* units. A *logical block* is always 512 bytes. A single read or write call generally results in exactly one I/O

operation. Therefore, raw I/O is considerably more efficient when the transfer size is very large. In particular, due to the way the IDC disks are formatted, very large sequential (>>64K) transfers result in no lost revolutions when the head seeks from cylinder to cylinder.

DISK PARTITIONS

The *default* size of each disk partition is a function of the total size of the disk. The *c* partition occupies the entire disk, minus the cylinders which are reserved for topology, manufacturer's defect list and diagnostic purposes. The *g* partition occupies the same space as the total space occupied by the *d*, *e*, and *f* partitions. More detailed information about disk partitions may be found in the *Managing ConvexOS: Configuration Guide*. A subsequent release of ConvexOS will provide a disk label editor utility, permitting the system manager to redefine the sizes and number of du partitions. That utility will employ the two special subunits unique to the DU device, the zero and control partitions. These are also raw devices, but they are NOT data partitions. The special file names are /dev/rduN-0 and /dev/rduN-ctl, where N represents the disk number [0-511].

DO NOT ATTEMPT TO USE duN-0 or duN-ctl, THEY ARE INTENDED FOR USE BY SPECIAL, PRIVILEGED DIAGNOSTIC SOFTWARE. MISUSE OF THESE SPECIAL SUBUNITS CAN COMPLETELY DESTROY ANY DATA ON THE DISK, SUCH THAT IT WILL BE IMPOSSIBLE TO RECOVER.

FILES

/dev/MAKEDEV	script to create the /dev/[r]du nodes
/dev/du[0-511][a-h]	block files
/dev/rdu[0-511][a-h]	raw files
/dev/rdu[0-511]-0	the 'zero' diagnostic subunit
/dev/rdu[0-511]-ctl	the 'control' diagnostic subunit

NAME

ex – Excelan 10 Mb/s Ethernet network interface

SYNOPSIS

ctlr LAN-001 csr0x%x int %d
unit %d type ex

ctlr LAN-007 csr0x%x int %d
unit %d type ex

DESCRIPTION

The *ex* interface provides access to a 10 Mb/s Ethernet network through an Excelan controller used as a link-layer interface. LAN-001 is the Excelan 201 Multibus controller. LAN-007 is the Excelan 302 VMEbus controller.

Each of the host's network addresses is specified at boot time with an SIOCSIFADDR ioctl. The *ex* interface employs the address resolution protocol described in *arp*(4P) to dynamically map between Internet and Ethernet addresses on the local network.

The interface normally tries to use a "**trailer**" encapsulation to minimize copying data on input and output. The use of trailers is negotiated with ARP. This negotiation may be disabled, on a per-interface basis, by setting the IFF_NOTRAILERS flag with an SIOCSIFFLAGS ioctl. This is typically done by using ifconfig(8C).

When the SIOCSIFFLAGS ioctl is used to mark the interface **down**, a hardware reset of the Excelan controller will be attempted.

When the SIOCSIFFLAGS ioctl is used to mark the interface **up**, a software initialization of the Excelan controller will be attempted, the controller placed on-line and connected to the Ethernet.

When the SIOCSIFADDR ioctl is used, an implicit hardware reset and software initialization is performed. Additionally, the physical ethernet address of the interface will be printed to the console.

Version 6 of the Ethernet firmware does not support the reporting of information on transmit collisions.

SEE ALSO

intro(4n), inet(4F), arp(4P), ifconfig(8)

NOTES

Ex is an optional product; for more information, contact your CONVEX sales representative.

NAME

hy – HYPERchannel driver or network interface

SYNOPSIS

ctlr LAN-002 csr 0x%x int %d /* Multibus device */
unit %d

ctlr LAN-004 csr 0x%x int %d /* Multibus network interface */
unit %d type hy

ctlr LAN-204 csr 0x%x int %d /* VME network interface */
unit %d type hy

DESCRIPTION

Hy provides a device driver or network interface for the Network Systems Corporation (NSC) A400 and NB400 HYPERchannel adapters. LAN-002 is the device driver for the IKON 10077NSC multibus controller. LAN-004 is the network interface for the IKON 10077NSC multibus controller. LAN-204 is the network interface for the IKON 10090 VME controller.

The *hy* device is a character special device with major device number 15. The standard system calls *open, close, read,* and *write* are supported, as well as ten *ioctl* functions.

Any number of processes may have the *hy* device open at a given time. In addition, simultaneous *read* and *write* system calls will exhibit full duplex behavior. The *open* system call does not cause any transmission or any change of state of the NSC adapter.

The *close* system call does not cause any transmission, port markdowns, or any other change of state of the NSC adapter.

The *write* system call will transmit a HYPERchannel message proper and, optionally, associated data. The buffer address supplied in the system call must be a message proper buffer followed immediately by the associated data buffer. If no associated data is to be transmitted, the byte count must be equal to or less than the largest message proper (64 bytes). If associated data is to be transmitted, the byte count must be greater than the largest message proper (64 bytes), and the associated data bit must be set in the control field of the message proper.

The *read* system call will input a HYPERchannel message proper and any associated data. The message proper will be placed at the beginning of the buffer supplied with the system call, and any associated data will begin at the offset of the maximum message proper (64 bytes). A returned byte count less than or equal to the maximum message proper indicates that no associated data was sent, and a byte count greater than the maximum message proper indicates accompanying associated data.

Ten *ioctl* commands are supported. These commands are defined in <*dev_iop/hyioctl.h*>.

```
#define HYIOCSTATUS _Ior('h', 1, struct hy_status)
#define HYIOCGETSTATUS _Ior('h', 2, struct hy_status)
#define HYIOCGETSTATISTICSr_Ior('h', 3, struct hy_stat)
#define HYIOCTIMERVAL _Iow('h', 4, int)
#define HYIOCRETRYVAL _Iow('h', 5, int)
#define HYIOCDATAQUERY _Ior('h', 6, int)
#define HYIOCCLEAR _Io('h', 7)
#define HYIOCDOWN _Iow('h', 8, int)
#define HYIOCDOWNREROUTE _Iow('h', 9, int)
#define HYIOCRESET _Io('h', 10)
/*
```

```
 * Structure of Statistics Record (counters)
 */
struct hy_stat {
        u_char  hyc_msgcnt[12];         /* # messages transmitted */
        u_short hyc_cancels;            /* # canceled operations */
        u_short hyc_aborts;             /* # aborted operations */
        u_char  hyc_retransmits[12];    /* # frames retransmitted */
        u_char  hyc_atype[3];           /* adapter type and revision level */
        u_char  hyc_uaddr;              /* adapter unit number */
};
/*
 * Structure of the Status Record
 */
struct hy_status {
        u_char hys_gen_status;          /* general status byte */
        u_char hys_last_fcn;            /* last function code issued */
        u_char hys_resp_trunk;          /* trunk response byte */
        u_char hys_status_trunk;        /* trunk status byte */
        u_char hys_recd_resp;           /* received response byte */
        u_char hys_error;               /* error code */
        u_char hys_caddr;               /* compressed add of 1st msg on chain */
        u_char hys_pad;                 /* not used */
};
```

HYIOCSTATUS returns the contents of the previous adapter status. Status is sampled by the driver on every abnormal termination of an adapter function, and that status may be retrieved with this *ioctl*. Note that an adapter status function is not executed on behalf of this *ioctl*, only the previous status is returned.

HYIOCGETSTATUS also returns adapter status, however, an adapter status function is executed before returning the results. If this *ioctl* is used immediately after an I/O error, the status returned will be the status of the previous status command, not the status of the function that caused the error.

HYIOCGETSTATISTICS returns the 32-byte HYPERchannel statistics buffer.

HYIOCTIMERVAL sets the value of the read timeout in seconds (default is "no timeout").

HYIOCRETRYVAL sets the value of the retry count used to retry transmissions.

HYIOCDATAQUERY returns the value TRUE if data is available to be read from the adapter. The value FALSE is returned if no data is available.

HYIOCRESET will issue a hardware reset to the adapter. This should only be issued if the adapter becomes hung.

HYIOCCLEAR will issue a clear adapter command to the A400.

HYIOCDOWN will issue a port markdown to the A400. A pointer to the port number is the third *ioctl* parameter.

HYIOCDOWNREROUTE will issue a port markdown and reroute to the A400. A pointer to the port number is the third *ioctl* parameter.

The *hy* network interface supports the Internet protocol family. The implementation of this interface complies with RFC-1044.

The host's Internet address is specified at boot time with an SIOCSIFADDR *ioctl* (see *ifconfig* (8)). If *hyroute (8c)* is not run to map adapter address to Internet address, the specified address must be a class A Internet address (a.b.c.d), where byte **a** supplies the network number, byte **b** must be zero, byte **c** is the Network Systems adapter address, and byte **d** is the adapter port number.

SEE ALSO

intro(4n), inet(4f), ifconfig(8), hyroute(8c), hystat(1c)

NOTES

The network interface drivers are included in the CONVEX Internet Services product.

NAME

icmp – Internet Control Message Protocol

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_RAW, proto);

DESCRIPTION

ICMP is the error and control message protocol used by IP and the Internet protocol family. It may be accessed through a "raw socket" for network monitoring and diagnostic functions. The *proto* parameter to the socket call to create an ICMP socket is obtained from *getprotobyname*(3N). ICMP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls, though the *connect*(2) call may also be used to fix the destination for future packets (in which case the *read*(2) or *recv*(2) and *write*(2) or *send*(2) system calls may be used).

Outgoing packets automatically have an IP header prepended to them (based on the destination address). Incoming packets are received with the IP header and options intact.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;

[ENOBUFS] when the system runs out of memory for an internal data structure;

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists.

SEE ALSO

send(2), recv(2), intro(4N), inet(4F), ip(4P)

NOTES

Icmp is an optional product; for more information, contact your CONVEX sales representative.

NAME

inet – Internet protocol family

SYNOPSIS

#include <sys/types.h>
#include <netinet/in.h>

DESCRIPTION

The Internet protocol family is a collection of protocols layered atop the *Internet Protocol* (IP) transport layer, and utilizing the Internet address format. The Internet family provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types; the SOCK_RAW interface provides access to the IP protocol.

ADDRESSING

Internet addresses are four byte quantities, stored in network standard format (on the VAX these are word and byte reversed). The include file *<netinet/in.h>* defines this address as a discriminated union.

Sockets bound to the Internet protocol family utilize the following addressing structure,

```
struct sockaddr_in {
        short       sin_family;
        u_short     sin_port;
        struct      in_addr sin_addr;
        char        sin_zero[8];
};
```

Sockets may be created with the local address INADDR_ANY to effect "wildcard" matching on incoming messages. The address in a *connect*(2) or *sendto*(2) call may be given as INADDR_ANY to mean "this host." The distinguished address INADDR_BROADCAST is allowed as a shorthand for the broadcast address on the primary network if the first network configured supports broadcast.

PROTOCOLS

The Internet protocol family is comprised of the IP transport protocol, Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). TCP is used to support the SOCK_STREAM abstraction while UDP is used to support the SOCK_DGRAM abstraction. A raw interface to IP is available by creating an Internet socket of type SOCK_RAW. The ICMP message protocol is accessible from a raw socket.

The 32-bit Internet address contains both network and host parts. It is frequency-encoded; the most-significant bit is clear in Class A addresses, in which the high-order 8 bits are the network number. Class B addresses use the high-order 16 bits as the network field, and Class C addresses have a 24-bit network part. Sites with a cluster of local networks and a connection to the DARPA Internet may chose to use a single network number for the cluster; this is done by using subnet addressing. The local (host) portion of the address is further subdivided into subnet and host parts. Within a subnet, each subnet appears to be an individual network; externally, the entire cluster appears to be a single, uniform network requiring only a single routing entry. Subnet addressing is enabled and examined by the following *ioctl*(2) commands on a datagram socket in the Internet domain; they have the same form as the SIOCIFADDR command (see *intro*(4N)).

SIOCSIFNETMASK Set interface network mask. The network mask defines the network part of the address; if it contains more of the address than the address type would indicate, then subnets are in use.

SIOCGIFNETMASK Get interface network mask.

SEE ALSO

ioctl(2), socket(2), intro(4N), tcp(4P), udp(4P), ip(4P), icmp(4P)

CAVEAT

The Internet protocol support is subject to change as the Internet protocols develop. Users should not depend on details of the current implementation, but rather the services exported.

NOTES

Inet is an optional product; for more information, contact your CONVEX sales representative.

NAME

ioconfig – System I/O configuration description file

SYNOPSIS

The ioconfig file resides in the root of the SPU winchester disk.

DESCRIPTION

The *ioconfig* file contains the description of all the CCU, controller, and peripheral definitions for a given system configuration. It resides in the **SPU** root partition and is used by the operating system and diagnostics for device configuration information. The *ioconfig* file uses the following formats:

For IOP type ccus

iop *ccu_slot_number*
mbus *chassis_number*
ctlr *type* **csr 0x***address* **int** *interrupt_number*
unit *unit no.* **type** *unit_name*

For VIOP type ccus

viop *ccu_slot_number*
vme *chassis_number*
ctlr *type* **csr 0x***address* **int** *interrupt_number*
unit *unit no.* **type** *unit_name*

For 3480-compatible tape, the unit entry is :
unit *unit no.* **subunit** *subunit no.* **type** *unit_name*

For HSP type ccus

hsp *ccu_slot_number*
drvr *type* **csr 0x***address* **int** *interrupt_number*
chnl *channel no.* **type** *unit_name*

For IDC type ccus

idc *ccu_slot_number*
ipi *port_number*
drvr DKC-IP2
unit *unit no.* **type** *unit_name*

ccu_slot_number
A digit which corresponds to the slot number in which a CCU (iop, viop, hsp, or idc) is physically located. In C1 class machines, this will be 3 through 7, inclusive; in C210 class machines, it is 0 through 3.

chassis_number
A number between 0 and 3 for **IOP** ccus, which specifies the multibus chassis connected to the given IOP. For **VIOP** ccus, it is 0 or 1 as the VIOP supports at most two VMEbus chasses. **HSP** type ccus do not have controller chasses and therefore have no chassis_number.

type Type of controller residing in given chassis. For HSP, a user-supplied device controller type. For IDC, a disk device type.

address Control and status register (csr) address in hexadecimal format, corresponding to the

board strapping for the specific controller type.

interrupt_number
interrupt number associated with the controller.

unit_name
Device unit specification; the disk type name as specified in the /etc/disktab file.

port_number
The IDC port number, in the range 0 to 3.

unit_no The device unit number. Its range is device-specific. For 3480-compatible tape, the range is 0 to 6 which stands for the SCSI ID of the formatter.

subunit_no
The device subunit number. Its range is device-specific. For 3480-compatible tape , the range is 0 to 3 which stands for the LOGICAL UNIT NUMBER of the 3480-compatible tape drive.

Below is an example of an *ioconfig* file which supports the following: one disk, one LAN interface, and a 16-line terminal controller on Multibus 0 connected to IOP 3; two disks, one LAN interface and one 3480-compatible formatter attached with two 3480-compatible tape drives on VMEbus 0 connected to VIOP 5; two IPI disks on two different ports in a IDC in CCU 6.

```
iop 3
  mbus 0
    ctlr DKC-001 csr 0xdc0 int 2
      unit 0 type DKD-001
    ctlr LAN-001 csr 0xc40 int 1
      unit 0 type ex
    ctlr ACM-001 csr 0x020 int 7
      unit 0 type TTY
      unit 1 type TTY
      unit 2 type TTY
      unit 3 type TTY
      unit 4 type TTY
      unit 5 type TTY
      unit 6 type TTY
      unit 7 type TTY
      unit 8 type TTY
      unit 9 type TTY
      unit 10 type TTY
      unit 11 type TTY
      unit 12 type TTY
      unit 13 type TTY
      unit 14 type TTY
      unit 15 type TTY
viop 5
  vme 0
    ctlr DKC-203 csr 0xc00 int 1
      unit 0 type DKD-214
      unit 1 type DKD-214
    ctlr LAN-007 csr 0x001 int 2
      unit 0 type ve
    ctlr MTC-202 csr 0xee00 int 3
      unit 0 subunit 0 type MTD-207
      unit 0 subunit 1 type MTD-207
idc 6
  ipi 0
    drvr DKC-IP2
      unit 0 type DKD-502
  ipi 1
    drvr DKC-IP2
      unit 0 type DKD-502
```

SEE ALSO

intro(4), info(4)

NAME

iostats – I/O statistics

SYNOPSIS

#include <sys/ioctl.h>

ioctl(d, request, statbuf)
int d, request;
char statbuf[256];

DESCRIPTION

The I/O drivers and CCUs collect various statistics. The *ioctl* system call is used to retrieve these data from drivers that support the two requests IOC_GET_STATS and IOC_CLEAR_STATS.

In order to read the statistics, the device control file corresponding to the unit in question must be opened for reading. For example, to read the statistics for the IDC disk unit */dev/du0*, a program must open the */dev/rdu0-ctl* special file for reading and perform an IOC_GET_STATS *ioctl* In order to clear the counters, the device control file must be opened for writing. Clearing device statistics is a privileged operation, permitted only for root.

The statistics buffer is 256 bytes in size, and the first 25 statistics entries (including name and type) are the same for each class. The actual number statistics collected by a specific driver may vary, depending upon class and/or driver support. The disk class drivers, for example, collect 25 statistics per disk unit. Of these 25, 23 are counted in the class driver and 2 (seek counters) are counted in the CCU driver.

In order to provide a uniform interface to partial implementations (statistic subsets), the first element in the statistics structure is a 64 bit bitmask that specifies which elements are valid. Each bit in this mask corresponds to a 4-byte word in the 256 byte statistics buffer. When a bit is set by the driver, it means that the corresponding word is a valid statistics element, counter or partial counter. When a bit is clear, that statistics word is not being updated in the driver. Some elements (such as the device name and byte counters) are 8 bytes long. Two bits will be set in the bitmask for these elements.

The second element in the statistics buffer is an eight byte null-terminated ASCII string, containing the base name of the device special file, e.g., *dd0*, *ccu2*, and *mt0*. The third element is a number that identifies the statistics format (structure) of the remaining elements.

The device statistics format numbers are (defined today, but not limited to):

RESERVED 0, DISK 1, TAPE 2, TTY 3, ETHERNET 4, HYPERCHANNEL 5, ULTRA 6, FDDI 7, CCU 8

At this time, only the disk class statistics format is defined (See next page).

I/O statistics counters are cumulative. However, they may be reset to zero at any time by issuing the IOC_CLEAR_STATS command. All counters are simultaneously reset; no selective reset of individual counters is supported. The bitmask, special file and statistics format fields are not reset.

In order to reduce source-level interdependencies between the utilities, kernel and I/O system, the statistics structure is ***not*** defined in a shared #include file data structure. Instead, a structure is defined for each device type in memory-image form. The drivers and various programs are then free to define their own data structures that map to this statistics image.

All elements are unsigned 32 bit integers except for the two byte-counters that are 64 bits and the special file name which is an 8 byte character array.

Note that the counters use C Series native byte-ordering (big-endian).

The statistics structure for disk devices is defined below

IOCTL Command Definitions

The two new ioctl commands are defined as

#define IOC_CLEAR_STATS _Io('c', 0)

and

#define IOC_GET_STATS (IOC_OUT | _IOC_TYPE('c')|1| ((256&_IOCPARM_MASK)<<16))

ERRORS

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

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

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

[EPERM] Only superusers may clear (reset to zero) the unit statistics.

SEE ALSO

ioctl(2), da(4), dd(4), du(4)

STATISTICS

The preceding figure defined the format of the statistics structure for disk devices. This section explains each of the fields. Each counter is a 32 bit unsigned integer, except where noted. Note that the first three fields of the structure are identical for all classes of device and are used as a 'self-description' of the statistics buffer to facilitate separate release of software.

Stats Bitmap — This 64 bit field is used to indicate valid statistics fields. Each bit corresponds to a 4-byte word in the 256 byte statistics buffer. Bit 0 (lsb) corresponds to the first word; bit 1 corresponds to the second word, and so on. Since some elements require more than four bytes, these elements will correspond to more than one bit. If a defined statistic is not recorded by the driver for that unit, then its corresponding bit(s) in the bitmap will be zero. If the statistic is valid, the bit(s) will be 1. Since the valid map itself is always used, bits 0 and 1 are always set. A program can test the bits in the bitmap of the returned statistics buffer before attempting to process any statistics fields or counters. This field is intended to be used by utilities to permit the same utility to support drivers that do not update all the statistics defined for a particular device.

Special File — This eight-byte field contains the base-name portion of the special file name for the device. For example, for */dev/rmt16*, the base name is "mt0". The base name portion of the VME disk file */dev/rdd0h* is "dd0". The name is stored as an ASCII, null-terminated string, convenient for printing with the %s *printf()* format by a utility. This means names will not be longer than 7 characters; most are 3 or 4 characters, depending on the unit number.

Format — The statistics in the preceding structure are suitable for various classes of device. Currently, only the disk driver supports these statistics ioctl commands. However, it is possible that each class will have a different statistics structure when support is added to other drivers. Using a structure format identifier, permits release of new driver and syspic without the co-requisite re-release of previous drivers.

Operations — These statistics accumulate the number of input and output requests the kernel makes to the device driver. The counter is per-physical-unit; each partition is not counted separately. When a write request is 'pulled' back from the driver, the counter for output operations is *not* decremented. Instead, the canceled output operations counter (see below) is incremented.

Bytes — This 64 bit unsigned integer counts the number of bytes successfully transferred to (write) or from (read) this physical unit.

Queue Timers
: These timers accumulate time that disk requests spend in the CPU side device driver queue. The request is time-stamped when the kernel gives it to the driver. When the driver removes it from the queue, immediately prior to sending the request to the CCU driver, the time that the request sat in the queue is added. Time spent in the queue for requests that are pulled back is not subtracted from the accumulator. For the stripe driver, these counters represent the TOTAL time that the request takes to complete. Since the stripe driver splits up its requests and hands them off to the real disk driver(s), these counters are somewhat redundant and less informative than those maintained in the real disk drivers.

CCU Timers
: These timers accumulates time for I/O requests for the interval from the instant when the request is enqueued (sent) to the CCU driver until its completion reply message arrives back in the CPU side driver.

Retries
: These counters are incremented when the CCU driver reports in its completion status that retries were required. Some CCU (channel) drivers do not report this fact in the status message, so the counter is of minimal value as an aid in system monitoring. When drivers are modified to use the standard Common Message Interface, it may prove more useful.

Canceled
: When the kernel cancels (or pulls back) a disk request, this counter is incremented. In effect, this counts I/O operations that have been synchronously stopped at the request of the I/O system client.

Aborted
: This counter is currently not supported. In effect, this counts I/O operations that have been asynchronously stopped as a result of some internal I/O system event or series of events, such as reloading or restarting a device driver.

Completion Counters
: The Common Message Interface (CMI) for CPU-to-CCU driver messages defines six levels of completion status. Not all drivers currently use CMI and those that are using it do not fully support it, particularly with respect to reporting severity levels. The six severity level completion counters keep track of how many of each of the various types of completion events have occurred. Input events are not separated from output events, though this might be an enhancement that could be made if it was valuable.

Number of Seeks
: This counter accumulates the number of seek operations executed by the CCU driver. Currently the seek counter is not provided and will always report 0 seeks.

Sum of Seek Deltas
: The distance of each seek is accumulated in this counter. It too, is not currently implemented.

STATISTICS FORMAT

The next page illustrates the format for disk statistics buffers.

```
struct io_statistics {
        unsigned long long      valid;          /* bitmask          */
        char                    devname[8];     /* logical unit name */
                                                /* du0, mt0, dd5, da2*/
        unsigned int            format;         /* statistics format */
        unsigned long int       in_ops;
        unsigned long long int  in_bytes;
        unsigned long long int  in_q_msec;      /* delta-t in queue */
        unsigned long long int  in_ccu_msec;    /* detla-t in ccu   */
        unsigned long           in_retries;     /* soft-errors */
        unsigned long int       out_ops;
        unsigned long long int  out_bytes;
        unsigned long long int  out_q_msec;     /* delta-t in queue */
        unsigned long long int  out_ccu_msec;   /* detla-t in ccu   */
        unsigned long           out_retries;    /* soft-errors */
        unsigned int            ctl_ops;
        unsigned int            cancelled;
        unsigned int            aborted;
        unsigned int            normal;         /* internal stuff */
        unsigned int            info;           /*       .        */
        unsigned int            warning;        /*       .        */
        unsigned int            error;          /*       .        */
        unsigned int            severe;         /*       .        */
        unsigned int            fatal;          /* not for syspic */
        unsigned int            nseeks;         /* not supported  */
        unsigned int            seeksum;        /* yet.           */
        unsigned int            spares[32];
        };
```

NAME

ip – Internet Protocol

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_RAW, proto);

DESCRIPTION

IP is the transport layer protocol used by the Internet protocol family. Options may be set at the IP level when using higher-level protocols that are based on IP (such as TCP and UDP). It may also be accessed through a "raw socket" when developing new protocols, or special purpose applications.

A single generic option is supported at the IP level, IP_OPTIONS, that may be used to provide IP options to be transmitted in the IP header of each outgoing packet. Options are set with *setsockopt*(2) and examined with *getsockopt*(2). The format of IP options to be sent is that specified by the IP protocol specification, with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use. A subsequent getsockopt() request will return an option list which reflects this adjustment. IP options may be used with any socket type in the Internet family.

Raw IP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls, though the *connect*(2) call may also be used to fix the destination for future packets (in which case the *read*(2) or *recv*(2) and *write*(2) or *send*(2) system calls may be used).

If *proto* is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If *proto* is non-zero, that protocol number will be used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number the socket is created with). Incoming packets are received with IP header and options intact.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;

[ENOBUFS] when the system runs out of memory for an internal data structure;

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists.

The following errors specific to IP may occur when setting or getting IP options:

[EINVAL] An unknown socket option name was given.

[EINVAL] The IP option field was improperly formed; an option field was shorter than the minimum value or longer than the option buffer provided.

SEE ALSO

getsockopt(2), send(2), recv(2), intro(4N), icmp(4P), inet(4F)

NOTES

Ip is an optional product; for more information, contact your CONVEX sales representative.

NAME

lo – software loopback network interface

SYNOPSIS

pseudo-device loop

DESCRIPTION

The *loop* interface is a software loopback mechanism which may be used for performance analysis, software testing, and/or local communication. As with other network interfaces, the loopback interface must have network addresses assigned for each address family with which it is to be used. These addresses may be set or changed with the SIOCSIFADDR ioctl. The loopback interface should be the last interface configured, as protocols may use the order of configuration as an indication of priority. The loopback should **never** be configured first unless no hardware interfaces exist.

DIAGNOSTICS

lo%d: can't handle af%d. The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

SEE ALSO

intro(4N), inet(3N), ifconfig(8C)

BUGS

Previous versions of the system enabled the loopback interface automatically, using a nonstandard Internet address (127.1). Use of that address is now discouraged; a reserved host address for the local network should be used instead.

NAME

mem, kmem – main memory

DESCRIPTION

Mem is a special file that is an image of the main memory of the computer. It may be used, for example, to examine (and even to patch) the system.

Byte addresses in *mem* are interpreted as physical memory addresses. References to non-existent locations cause errors to be returned.

The file *kmem* is the same as *mem* except that kernel virtual memory rather than physical memory is accessed.

FILES

/dev/mem
/dev/kmem

NAME

mtio – UNIX magtape interface

DESCRIPTION

The files *mt0, ..., mt247* refer to the UNIX reel tape drives. (UNIX is a registered trademark of UNIX System Laboratories, Inc.) CONVEX supports up to 32 physical magnetic reel tape drives. Since most installations confine tape use to physical devices 0 through 3, the discussion will cover only those drives. For a complete list of the tape naming conventions, see the tape system chapter of *Managing ConvexOS: Configuration Guide.* The following description applies to any of the transport/controller pairs. The files *mt0, ..., mt7* are 800bpi, *mt8, ..., mt15* are 1600bpi, and *mt16, ..., mt23* are 6250bpi. The files *mt0, ..., mt3, mt8, ..., mt11,* and *mt16, ..., mt19* are rewound when closed; the others are not. The following table summarizes the characteristics of the magtape drives 0 through 3.

Density	**Rewind?**	**unit0**	**unit1**	**unit2**	**unit3**
800 bpi	rewind	mt0	mt1	mt2	mt3
800 bpi	no rewind	mt4	mt5	mt6	mt7
1600 bpi	rewind	mt8	mt9	mt10	mt11
1600 bpi	no rewind	mt12	mt13	mt14	mt15
6250 bpi	rewind	mt16	mt17	mt18	mt19
6250 bpi	no rewind	mt20	mt21	mt22	mt23

In addition to these reel tape device, there are 32 cartridge tape, and Digital Audio Tape device files. These files have "tc0" through "tc31" (for cartridge tape) and "dat0" through "dat31" (for Digital Audio Tape) as a component of their names. The names of these devices have the following form:

/dev/[r]{tc|dat}**?**[n][u]

The following characters in this format have special significance:

r Raw tape device; absent for block mode tape.

? Unit number

n No rewind after file is closed; if **n** is absent, tape will be rewound.

u Data buffering disabled at the drive; if **u** is absent, buffering is enabled. Digital Audio Tape does not support this option.

Note that the CONVEX 3480-compatible cartridge tape drive is equipped with an internal data buffer. This buffer is used only if a device file that enables buffering is used to mount the drive. If buffering is used, the data transfer rate may be increased significantly over non-buffered rates. However, buffering also makes data loss more likely if power is shut off to the drive while writing is in progress. The Convex Digital Audio Tape (DAT) drive also is equipped with an internal data buffer. However, it is not possible to turn off this feature.

For cartridge tape and DAT, the maximum record size is 128 kbytes. Usually, the larger the record size, the better the data transfer rate. However, transferring 128 kytes for each read or write does not yield the best transfer rate. In addition, the record length should be an even number of bytes. This is because if the record size used in the read or write is an odd number, data is transferred to or from the tape one byte at a time; if the record size used in read or write is an even number, data is transferred 4 bytes at a time.

If a file open for writing is closed, two end-of-file markers are written. If the tape is a no-rewind-on-close device, it is positioned with the head between the two tape marks.

If a file open for reading is closed and a no-rewind-on-close device is used, the tape position does not change.

If a file open for reading and writing is closed, and if the last tape operation wrote to tape, two end-of-file markers are written. (The tape is positioned with the head between the two tape marks if the tape is a no-rewind-on-close device.) If none of these conditions are met, the user program is responsible for explicitly writing an end-of-file mark on the tape. For example, if a file open for reading and writing is closed and the last tape operation was a backward space record (bsr), **no** end-of-file markers are written to tape. In this example, the user program should have written two end-of-files after the last write to tape.

A standard tape consists of a series of records terminated by an end-of-file mark. Though this is inefficient, a tape can be treated much like an ordinary file. Seeks have their usual meaning and it is possible to read or write a byte at a time. Writing in very small units is inadvisable, however, because it tends to create too many record gaps; thus, much of the tape will be wasted on record gaps instead of data.

The *mt, tc* or *dat* 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, ..., rmt23, rtc0, ..., rtc23, rdat0, ..., rdat23* but the same minor-device considerations as for the regular files still apply. A number of other ioctl operations are available on raw magnetic tape. The following definitions are from *<sys/mtio.h>*:

```
/*
 * Structures and definitions for mag tape io control commands
 */

/* structure for MTIOCTOP - mag tape op command */
struct      mtop    {
    short   mt_op;          /* operations defined below */
    long    mt_count;   /* how many of them */
};

/* 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 MTGAP           8       /* write erase gap */
#define MTCLR           9       /* drive clear */
#define MTBLANK         10      /* erase the remaining medium; support
                                   for the cartridge tape only */
/*
 * Structure for MTIOCGET - mag tape get status command
 *
 *  Note mt_blkno and mt_fileno will be invalid following errors that
 *  that cause the tape to be lost. These will be reset correctly
 *  following a rewind.
```

```
*/

struct      mtget    {
    short   mt_type;    /* type of magtape device */
    short   mt_dsreg;   /* "drive status" register */
    short   mt_erreg;   /* "error" register */
    long    mt_resid;   /* residual count */
    long    mt_size;    /* maximum record size */
    long    mt_fileno;  /* file number of current position */
    long    mt_blkno;   /* block number of current position */
};

/*
 * Constants for mt_type byte
 */
#define     MT_ISTA         1       /* for 9-track tape */
#define     MT_ISTC         2       /* for cartridge tape */
#define     MT_ISDAT    3           /* for digital audio tape */

/*
 * Constants for mt_dsreg
 * Controller may not implement all status indications
 */

#define MT_ONL          0x0001      /* Online Status */
#define MT_RDY          0x0002      /* Ready Status */
#define MT_REW          0x0004      /* Rewinding Status */
#define MT_TM           0x0008      /* Tape Mark Status */
#define MT_FPT          0x0010      /* Write Protect Status */
#define MT_OVR          0x0020      /* Block Long on Read */
#define MT_BOT          0x0040      /* Beginning of Tape */
#define MT_EOT          0x0080      /* End of Tape */
```

The *MT_ONL* bit is set when the drive is able to accept commands. Tape drives can generally be place on-line/off-line via a switch on the drive. The *MT_RDY* bit is set when a tape is loaded. The *MT_REW* bit is set when a rewind command is in progress. The *MT_TM* bit is set after a tape operation causes the drive head to move over a file mark. If the operation moved the tape forward and crossed a tape mark, the *MT_TM* bit indicates that the head is immediately after a tape mark. If the operation moved the tape backward and crossed a tape mark, the *MT_TM* bit indicates that the head is immediately before a tape mark. The *MT_FPT* bit is set when the tape volume is physically write protected. The *MT_OVR* bit is set after a read operation where the tape record was larger than the read buffer. The *MT_BOT* bit is set when the tape is positioned at the beginning. The *MT_EOT* bit is set when the tape is positioned at or beyond the end of media warning.

```
#define MT_DSREG_BITS "\10\8EOT\7BOT\6OVR\5FPT\4\TM\3REW\2\RDY\1ONL"

/*
 * Constants for mt_erreg
 */

#define MT_LOST         0x0001      /* Tape Position Lost */
#define MT_DRVABT       0x0002      /* Driver aborted the command */
```

```
#define MT_ERREG_BITS "\10\2DRVABT\1LOST"

/* mag tape io control commands */
#define MTIOCTOP        _IOW(m, 1, struct mtop)/* do a mag tape op */
#define MTIOCGET        _IOR(m, 2, struct mtget)/* get tape status */
#define MTIOCLSTAT      _IOR(m, 3, struct mtget)/* faster "" "" "" */
                                                /* (MTIOCGET with  */
                                                /* no drive clear) */
#define MTIOCRTY        _IO(m, 2)       /* retry tape errors (default) */
#define MTIOCNRTY       _IO(m, 3)       /* don't retry tape errors  */
#define MTIOCEOT        _IO(m, 4)       /* EOT aborts commands (default) */
#define MTIOCNEOT       _IO(m, 5)       /* EOT is ignored */
#define MTIOCEOTCOM     _IO(m, 123)     /* EOT Processing is complete */
#define MTIOC100IPS     _IO(m, 124)     /* select 100ips */
#define MTIOC50IPS      _IO(m, 125)     /* select 50ips */

#ifndef KERNEL
#define DEFTAPE         "/dev/rmt12"
#endif
```

The correct way to see if a tape unit supports 100ips is issuing the speed select ioctl(2) for 100ips with MTIOC100IPS. If it fails, 100ips is not supported.

The cartridge tape device will ignore MTIOC100IPS and MTIOC50IPS.

The meaning of MTIOCEOT and MTIOCNEOT change slightly when labelled tapes are used. With labelled tapes, MTIOCEOT requests the end of each volume (EOT marker or EOV labels) be passed to the application. MTIOCNEOT requests that the intermediate end of volume indications be hidden from the application. With unlabeled tapes, MTIOCEOT tells the tape driver to observe the EOT marker. Reads and writes are not allowed to proceed past the EOT. With unlabeled tapes, MTIOCNEOT requests the tape driver to ignore the EOT mark. Reads and writes can occur past the EOT marker. It is the application's responsibility not to spin the tape off of the reel when ignoring the EOT marker.

The default behavior for labelled tapes is to ignore intermediate volume switches. The default behavior for unlabeled tapes is to be notified the end of the tape is reached.

When using a multivolume labelled tape set, MTIOCEOT will request end of volume notification. This means the application will be notified of the end of each intermediate volume in the volume set. EOV will be indicated by a short byte count being returned from the read(2) or write(2) system call, when reading, the MT_EOT bit of the mt_dsreg will also be set. Each read(2) or write(2) system call following the EOV notification will return a zero byte count until the MTIOCEOTCOM ioctl is issued. Following EOV notification, the application should issue the MTIOCEOTCOM ioctl to signal that it is ready for the next tape to be mounted. If there are no volumes left in the volume set, the ioctl will fail.

The MTIOCETCOM ioctl is undefined for unlabeled tapes. When used with an unlabeled tape, it will aways fail.

Other ioctls may be available that are tape drive specific. Refer to man pages in the SEE ALSO section.

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, provided it is no greater than the buffer size. If the record is greater than the buffer size, the record is truncated, the MT_OVR bit is set in the drive status, and the read is successful.

In raw tape, I/O seeks are ignored. A **0** byte count is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

FILES

/dev/mt?
/dev/rmt?
/dev/tc?
/dev/rtc?
/dev/dat?
/dev/rdat?

SEE ALSO

mt(1), tar(1), ta(4), tc(4)

BUGS

Using the tape as a non-labeled, block tape device without the control of tpdaemon may crash the system due to the lack of flow control for read or write requests in the device driver. Accessing the tape via asynchronous I/O under the same condition may also cause the same problem.

NAME

null – data sink

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

NAME

pa – line printer driver

SYNOPSIS

ctlr PRC-001 csr 0x%x int %d
unit PRT-001 type PRT-001

ctlr ACM-201 csr 0x%x int %d
unit 16 type PRT-CEN

ctlr ACM-201 csr 0x%x int %d
unit 16 type PRT-DAT

DESCRIPTION

The *pa* device provides a line printer interface. PRC-001 is the Multibus controller. Port 16 of the VMEbus tty controller ACM-201 is the VMEbus printer interface. The Multibus controller provides an interface to a Printronix line printer using the Centronix interface, while the VMEbus controller provides either a Centronix or Dataproducts line printer interface. Both interfaces use the same physical port, 16. This is selected by specifying a unit type of "PRT-CEN" or "PRT-DAT" respectively.

The unit number of the printer is specified by the minor device after removing the low 3 bits, which act as per-device parameters. Currently the two low bits of the minor device number are interpreted.

If the lowest bit is set, the device is treated as having a 64-character set, rather than a full 96-character set. In the resulting half-ASCII mode, lower case letters are turned into upper case and the characters in the left-hand column below become the ones in the corresponding right-hand column:

```
{ (
} )
` '

| !
~ ^
```

If the second of the three lowest bits in the minor device number is set, then the driver is placed in raw mode. This mode eliminates all output processing.

If the driver is not in raw mode it correctly interprets carriage returns, backspaces, tabs, and form feeds. Lines longer than the maximum page width of 132 are truncated.

FILES

/dev/lp

SEE ALSO

lpr(1), ca(4)

DIAGNOSTICS

pa%d: printer not ready

The printer is off line or out of paper. This message will print with a frequency decreasing to once an hour while the condition persists and characters remain to be printed.

pa%d: printer paper error

The printer is out of paper or a paper jam condition exist. This message will print with a frequency decreasing to once an hour while the condition persists and characters remain to be printed.

NAME

pb – Ikon controller for Versatec plotters

SYNOPSIS

ctlr VER-001 csr 0x%x int %d
unit %d type PLT-001

DESCRIPTION

This provides an interface to Versatec compatible plotters using an Ikon Multibus interface. Currently, only one plotter may be connected to each such controller. The plotter may be operated in print mode, plot mode, or simultaneous print plot mode. (See the Versatec plotter manual for a description of these modes.) In print mode, if canonical processing is selected, tabs will be expanded to blanks (aligning to every eight columns), and lowercase characters may optionally be translated to uppercase (to support uppercase only printers). In plot mode, canonical processing will send the Versatec remote end-of-line command after every write system call, so that the user can issue one write call per plotter line of dots.

The initial mode is set by the minor device number. The least significant bit (bit 0) should be set for plot mode (e.g. minor device 1), and clear for print mode (e.g. minor device 0). Bit 1 should be set only if simultaneous print/plot mode is desired (e.g. minor device 2). Bit 2 should be set only if canonical processing is desired (e.g. minor device 4 or 5), and bit 3 should be set if lowercase is to be translated to uppercase. The upper four bits of the minor device number select any of several possible Ikon controller cards per system (your system probably has been compiled in maximum configuration).

For normal printer operation, most users will select minor device 4, which enables canonical processing and sets the plotter to print mode. For normal plot operation, most users will select minor device 5, which enables canonical processing and sets the plotter to plot mode. In this mode, each write system call represents one row of dots to be output to the plotter.

The plotter mode can be changed by using *ioctl* calls. The commands TIOCLSET, TIOCLBIS, and TIOCLBIC can be used to set or clear the mode bits PBPLOT, PBSPP, PBCANONICAL, and PBUPPER (see *machine/pb1.h* and *tty*(4)). The *ioctl* command TIOCLGET will return the current mode to the user. The command TIOCSET can be used to send a remote command to the Versatec plotter, any of: CMDCLEAR, CMDFF, CMDEOT, CMDEOL to send a clear, form feed, end of transmission, and end-of-line respectively. The command TIOCGET will return the current plotter status, which is any of: OFFLINE | NOPAPER | NOTREADY.

Only one process may have this file open at a time. It is write-only.

FILES

/dev/pb? (normally character special devices with a major device number of 14)

BUGS

In every mode but canonical print, transfers are limited to 4096 bytes or less on a single write call.

DIAGNOSTICS

pb: Versatec unit ? offline.
pb: Versatec unit ? out of paper.
pb: Versatec unit ? not ready.

NAME

pi – process interface

SYNOPSIS

#include <sys/pcntl.h>
#include <sys/ioctl.h>

DESCRIPTION

This section describes a process control interface which is used to implement the ConvexOS process debuggers. Using this interface, the execution of a process can be controlled by another process (e.g. a debugger) by using the *pattach(2)* system call to "attach" to the process to be controlled. *pattach* returns a descriptor that may be used to read/write a process's address space, or control the execution of the process through ioctl calls.

A process may also be opened for reading and/or writing regardless of exclusive execution to permit machine-dependent utilities, such as *ps*, to read information contained within a given process's address space. Information may be read from and written to the process's address space using the standard *read(2)* , *write(2)* and *lseek(2)* system calls. Other functions, such as reading and writing registers and single-stepping, may be done using the *ioctl(2)* system call. Arguments to *ioctl* include the process descriptor, operation to be performed, and a pointer to the process control structure defined in *<sys/pcntl.h>* The process control structure is divided into input and output parameters. The input parameters in the structure are supplied by the caller and are prefixed with **pi_** . The output parameters are updated by the kernel, and are prefixed with **pi_o** .

In general, ioctls on a process descriptor returned by *pattach* attempt to read/write *pi_nbytes* bytes of data from the process referenced by the descriptor into the buffer pointed to by *pi_buffer*. *Pi_offset* is used in all the register read/write operations. It is the byte offset in the appropriate register structure to the first byte to be transferred to/from *pi_buffer*. This offset permits reading/writing a single byte within any register set.

The following *ioctl* operations are supported on processes which have been opened for read access.

PIGETCWD — Return a path to the process's current working directory as a NULL-terminated string in *pi_buffer*. The length of the buffer is given in *pi_nbytes*, and is typically MAXPATHLEN from *<sys/param.h>*.

PIGETFLAGS — Read the process flags word *p_flags* as from the proc structure as defined *<sys/proc.h>* and returns its value in *pi_opflags*.

PIGETP — Read the process proc structure as defined in *<sys/proc.h>* into the user defined buffer *pi_buffer*. Input parameter *pi_nbytes* must contain the sizeof(struct proc). Note: This structure and its size are dependent on the current version of ConvexOS.

PIGETROOT — Return a path to the process's root directory as a NULL-terminated string in The length of the buffer is given in *pi_nbytes*, and is typically MAXPATHLEN from *<sys/param.h>*.

PIGETT — Read the process thread structure for the thread id specified in *pi_thread* into the user defined buffer *pi_buffer*. Input parameter *pi_nbytes* must contain the sizeof(struct thread). The format of the thread structure is found in *<sys/thread.h>*. Note: This structure and its size are dependent on the current version of ConvexOS.

PIGETU — Read the process user structure as defined in *<sys/user.h>* into the user defined buffer *pi_buffer*. Input parameter *pi_nbytes* must contain the sizeof(struct user). Note: This structure and its size are dependent on the current version of ConvexOS.

PIRDSDRS — Read the process segment descriptor registers as defined in

<machine/sdr.h> into user defined buffer *pi_buffer*. Input parameter *pi_nbytes* must contain the sizeof(struct sdr).

PISETRWTID Set the thread id to be used in all thread memory read/write requests to the process to *pi_thread*. The default thread id is **0**.

The following *ioctl* operations are supported on processes which are opened for exclusive access. These requests are generally used to implement process and thread debugging. Once a process is attached, it must be stopped as indicated by the *wait3* system call prior to execution of the following *ioctl* operations.

PIXCONTINUE Enable the thread specified by *pi_thread* to continue execution at program counter *pi_pc* with signal *pi_signo*. *pi_signo* should contain a signal to be passed to the thread when execution resumes at *pi_pc*. *pi_signo* is usually either a **0**, which indicates that the signal that caused the thread to stop should be ignored, or a signal for the thread to handle. If *pi_pc* is a **1**, execution resumes from where the thread stopped, otherwise it is taken as the absolute pc at which to resume thread execution.

PIXDETACH This request is used to specify that the process currently attached be released to resume execution at process close time. If detach is not specified, the process is terminated on close. The default is terminate on close.

PIXGETSIGACTION Return the process action associated with the signal number specified in *pi_signo*. The signal action is returned in *pi_osigact* and is described in *sigvec(2)*.

PIXGETSUBCODE Read thread state information pertaining to the thread specified in *pi_thread*. The state of the thread is returned in *pi_otstate*. A state of PI_TGONE, indicates the thread is not currently in execution. A state of PI_TALIVE indicates the thread is executing, and that its reason for stopping is indicated in *pi_osigno* and *pi_osigcode*. If *pi_osigno* is zero, the thread has been stopped simply because the process stopped, otherwise it indicates the signal which caused the thread to stop. On systems which support an independent thread time, *pi_ottimer* contains the threads 64 bit usec timer.

PIXGETTHCOUNT Read the maximum thread count for the process into *pi_othdcnt*. This request is used to determine the number of execution streams within a process.

PIXINHERIT This request can be used to specify, via *PI_SETINHERIT* in *pi_inherit*, that all children of the process are considered attached prior to execution of their first instruction. When inherit is set, any fork request by the current process will cause a SIGTRAP signal with subcode TRP_FORK_TRAP to be sent to controlling process. The default is no inherit. The inherit property is propagated to the forked children. Inherit may be cleared by issuing a *PIXINHERIT* with *pi_inherit* set to *PI_CLEARINHERIT*.

PIXRDCREGS Read the entire communication register set and lock bits for the process. *pi_offset* must be zero, and *pi_nbytes* must be sizeof(struct creg_ctx) as defined in *<machine/creg.h>*.

PIXRDREGS Read the scalar register set as defined in *<machine/reg.h>* as (struct syscall_context).

PIXRDVREGS Read the entire vector register set as defined in *<machine/reg.h>* as (struct vecst).

PIXRUN — Resume execution of the process. All threads which have been continued or single stepped will resume execution. All other PI_ALIVE threads will be suspended from execution. This request will fail if an attempt is made to run a process in which none of the threads have been continued or stepped.

PIXSTEP — Enable the thread to single–step. This request works in the same manner as PIXCONTINUE, except that the thread will stop with a *pi_osigno* of SIGTRAP, and a *pi_osigcode* of TRP_TRACE_TRAP after one instruction has been executed.

PIXTERMINATE — Terminate the process.

PIXWRCREGS — Write the ring 4 communication register set and lock bits for the process. *pi_offset* must be zero, and *pi_nbytes* must be sizeof(struct creg_ctx) as defined in *<machine/creg.h>*. Only the ring 4 registers and lock bits are actually written.

PIXWRREGS — Write the register set as defined in *<machine/reg.h>* as (struct syscall_context).

PIXWRVREGS — Write the vector register set as defined in *<machine/reg.h>* as (structure vecst).

RETURN VALUE

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

ERRORS

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

[EBADF] Invalid process file descriptor.

[EINVAL] Invalid ioctl request

[ERSCH] Process no longer exists, or a exclusive request was made on a non-stopped process.

[EFAULT] Specified buffer address, thread id, byte count, signal number, or user pc is out of range.

SEE ALSO

adb(1), *csd*(1), *close*(2), *ioctl*(2), *lseek*(2), *pattach*(2), *read*(2), *write*(2)

NAME

pty – pseudo terminal driver

SYNOPSIS

pseudo-device pty

DESCRIPTION

The *pty* driver provides support for a device-pair termed a *psuedo-terminal.* A pseudo terminal is a pair of character devices, a *master* device and a *slave* device. The slave device provides processes an interface identical to that described in *tty*(4). However, whereas all other devices which provide the interface described in *tty*(4) have a hardware of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the pseudo terminal. That is, anything written on the master device is given to the slave device as input and anything written on the slave device is presented as input on the master device.

The master device is an exclusive device, that is, only one process in the system may have the master side open at a time. Attempts by other processes to open the device will fail with EIO returned in *errno.* When a process closes the master device, access to the slave device is revoked in the manner of *vhangup*(2), causing I/O for processes attached to the slave device to fail. The slave device must be opened by a process that is a descendant of the master process (and the master process must still exist) or the slave device open will fail with EACCES returned in *errno.*

In configuring, if no optional "count" is given in the specification, 16 pseudo terminal pairs are configured.

The following *ioctl* calls apply only to pseudo terminals:

TIOCSTOP

Stops output to a terminal (e.g. like typing ^S). Takes no parameter.

TIOCSTART

Restarts output (stopped by TIOCSTOP or by typing ^S). Takes no parameter.

TIOCPKT

Enable/disable *packet* mode. Packet mode is enabled by specifying (by reference) a nonzero parameter and disabled by specifying (by reference) a zero parameter. When applied to the master side of a pseudo terminal, each subsequent *read* from the terminal will return data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as TIOCPKT_DATA), or a single byte reflecting control status information. In the latter case, the byte is an inclusive-or of zero or more of the bits:

TIOCPKT_FLUSHREAD

whenever the read queue for the terminal is flushed.

TIOCPKT_FLUSHWRITE

whenever the write queue for the terminal is flushed.

TIOCPKT_STOP

whenever output to the terminal is stopped a la ^S.

TIOCPKT_START

whenever output to the terminal is restarted.

TIOCPKT_DOSTOP

whenever *t stopc* is ^S and *t startc* is ^Q.

TIOCPKT_NOSTOP

whenever the start and stop characters are not ^S/^Q.

This mode is used by *rlogin*(1c) and *rlogin*(8C) to implement a remote-echoed, locally ^S/^Q flow-controlled remote login with proper back-flushing of output; it can be used by other similar programs.

TIOCREMOTE

A mode for the master half of a pseudo terminal, independent of TIOCPKT. This mode causes input to the pseudo terminal to be flow controlled and not input edited (regardless of the terminal mode). Each write to the control terminal produces a record boundary for the process reading the terminal. In normal usage, a write of data is like the data typed as a line on the terminal; a write of 0 bytes is like typing an end-of-file character. TIOCREMOTE can be used when doing remote line editing in a window manager, or whenever flow controlled input is required.

FILES

/dev/pty[p-tonmlkjihgfe][0-9a-f] master pseudo terminals
/dev/tty[p-tonmlkjihgfe][0-9a-f] slave pseudo terminals

DIAGNOSTICS

None.

BUGS

It is not possible to send an EOT.

NOTES

Pty is an optional product; for more information, contact your CONVEX sales representative.

/etc/disktab Disk geometry and fs partition data.
/etc/mkfs To actually build the file system

SEE ALSO

st(4), disktab(5), streconfig(5), stripecap(5),getst(8), putst(8), mvst(8), rmst(8), qst(8), convst(8), mkfs(8), fsirand(8), newfs(8)

CAUTIONS

As a courtesy to other drivers and the *dump (8)* utility, raw I/O transfers whose length is not a multiple of the stripe sector size will be correctly handled, but raw I/O transfers which do not start on a stripe sector boundary will return an error.

NAME

st – stripe (disk) device interface

SYNOPSIS

st?

DESCRIPTION

The stripe interface allows users to combine various block devices into one logical device. Normally, this is used only for disk devices, allowing several disk partitions to be logically treated as a single partition. Benefits of disk *striping* include elimination of the restriction that a file system reside in a single partition (or physical disk), enhanced throughput due to the parallelism available from having multiple disk devices and CCUs working on the same file system, and the possibility of better disk-drive load balancing.

The "striped file system" is supported by the *Virtual Volume Manager (VVM).* The *VVM* driver provides redundant and non-redundant stripe capability. Redundant stripes protect against data loss from disk failure by either data mirroring or use of parity. From a user standpoint, redundant stripes will operate identically to non-redundant stripes. The difference is in the action and output of some of the *VVM* utilities (i.e. the output of *getst* is enhanced for redundant stripes, and supports the *-H* option, which gives information regarding Hot Spares - see *getst(8)*).

There are up to 256 stripes available. Each is referenced with a different minor device number (0-255). The actual number the system will allow is set by the boot parameter *stripe_devices*, which is by default set to 16. By convention, the special files */dev/rst0* through */dev/rst255* are used to access the raw stripe devices. The special files */dev/st0* through */dev/st255* are used to access the block stripe devices. Each stripe can combine up to 128 other block devices, such as disk partitions. There is no requirement that the block devices be of the same size or physical layout, although such parameters may affect the efficiency of disk transfers.

Before a stripe can be accessed, a table that specifies the layout of the stripe must be loaded into the memory of the ConvexOS kernel. (In particular, the stripe table must be loaded before *mkfs(8)*, *fsck(8)*, or *mount(8)* can be invoked on the stripe, and should never be changed while the stripe is mounted, or while the device is being accessed.) Normally, the script */etc/rc* and *init(8)* invoke *putst(8)* in order to load all the stripe tables at reboot time. Alternately, the table loading may be done manually with the program *putst*. The layout of this table is specified in the file */usr/include/sys/stripe.h*, and includes the major/minor device numbers of the component block devices, information on the number and size of the component blocks, and the stripe block size. Normal users should make the stripe table using *newst(8)*, which automatically handles all the low-level details. Make sure the stripe table is loaded by an invocation of *putst* in */etc/initrc* at every reboot.

The *VVM* system incorporates a daemon - the *vvmdaemon*, which listens for messages from the kernel regarding failed disks. If a disk in a redundant stripe should fail, the *vvmdaemon* will invoke the *mvst(8)* utility to begin reconstruction. Any problems encountered during reconstruction will cause the generation of error messages where appropriate. Should the system crash during reconstruction of a failed disk, at reboot time the *vvmdaemon* will detect this condition and restart the reconstruction. This is accomplished through a status file */etc/streconfig*, which retains data regarding outstanding *mvst(8)* activity until reconstruction is complete.

The *VVM* utilities offer a complete set of operations to create and manipulate stripes and Hot Spares, there should be no need for users to manually modify either */etc/stripecap* or */etc/streconfig*, which is highly discouraged.

FILES

/dev/rda??	Raw devices for the disk driver.
/dev/rst?	Raw stripe devices.
/etc/stripecap	Database of the stripe descriptors.
/etc/streconfig	Database of outstanding *mvst* processes.

NAME

ta – Nine–Track Magnetic Tape Device Driver

SYNOPSIS

ctlr MTC-001 csr 0x%x int %d
unit %d type MTD-001 **(STK 2920 Series model 2921)**

ctlr MTC-001 csr 0x%x int %d
unit %d type MTD-002 **(STK 1960 Series models 1963 and 1968)**

ctlr MTC-001 csr 0x%x int %d
unit %d type MTD-003 **(Fujitsu M2436 Series)**

ctlr MTC-001 csr 0x%x int %d
unit %d type MTD-004 **(STK 2920 Series model 2922)**

ctlr MTC-201 csr 0x%x int %d
unit %d type MTD-201 **(STK 2920 Series model 2921)**

ctlr MTC-201 csr 0x%x int %d
unit %d type MTD-202 **(STK 1960 Series models 1963 and 1968)**

ctlr MTC-201 csr 0x%x int %d
unit %d type MTD-203 **(Fujitsu M2436 Series)**

ctlr MTC-201 csr 0x%x int %d
unit %d type MTD-204 **(STK 2920 Series model 2922)**

DESCRIPTION

ta provides a nine–track tape device driver that implements the standard tape interface as described in *mtio*(4). MTC-001 is the Multibus nine–track tape controller, and MTC-201 is the VMEbus nine–track tape controller. The STK 2921 tape drive supports 1,600 and 6,250 bits per inch (bpi) and runs at 50 inches per second (ips). The STK 1960 Series (STK 1963 and STK 1968) tape drives support 800, 1,600, and 6,250 bpi and runs at 125 ips. The Fujitsu M2436 Series tape drive supports 1,600 and 6,250 bpi and runs at 200 ips. The STK 2922, the last supported tape drive, supports 1,600 and 6,250 bpi and runs at 50 and 100 ips. Either controller can support the following configurations: four STK 2920 Series tape drives, eight STK 1960 Series tape drives, or eight Fujitsu M2436 Series tape drives. The default number of *ta* tape drives supported by the *ta* device driver is eight; the maximum number of tape drives the *ta* device driver can support is thirty-two. The number of *ta* tape drives supported is determined at system generation time and can be changed with *sysgen*(8). The ConvexOS *ta* device driver supports two different tape read/write interfaces: *character tape* and *block tape*. The behavior of the read and write operations and existence of the the ioctl operations in the character tape interface are the only differences between the interfaces. The interface used is determined by which device file is opened; see the FILES section.

Character Tape Interface

The *character tape* interface allows variable-length read and write operations. Valid record sizes range from one byte to sixteen megabytes (16,777,216 bytes). An attempt to read or write a size outside this range returns an error. Tape marks are placed between records to designate the end of a file. If, during a read, a tape mark is encountered, the read returns zero. A further read will read the first record of the next file. The logical end-of-tape is designated by two consecutive tape marks. The logical end-of-tape can be identified by two consecutive reads returning zero. Seek operations, *lseek*(2), are ignored in the *character* mode; ioctls are provided in the *mtio*(4)

interface for tape positioning. The *character tape* interface is the preferred method when using nine-track tape.

Block Tape Interface

The *block tape* interface only allows fixed-length read and write operations. Valid record sizes range from one byte to 65,536 bytes. The default record size is 65,536 bytes. The record size of block tape transfers is referred to as the *block record size*. The block record size can be changed with the *fcntl* command _F_SETBLKSIZE; see *fcntl*(2) for details. An attempt to read or write a size less than one byte returns an error. An attempt to read or write a size larger than the block record size will be broken up into multiple records. For example, if the block record size has been set to 4,096 bytes and a write request for 12,288 bytes is issued, the *ta* device driver will write three records, each 4,096 bytes in length. As another example, if the block record size has been set to 8,192 bytes and a write request for 8,193 is issued, the *ta* device driver will write one record 8,192 bytes long, then attempt to *read* the next record, 8,192 bytes. The first byte of the read record will be replaced with the last byte of the 8,193 write request and the new 8,192 byte long record will be written to the tape. In general for block write operations, if the requested transfer size is not a multiple of the block record size the last record of the transfer will be written using a read-modify-write algorithm. Also since it is likely the read operation of the read-modify-write operation will fail it is advisable to always read and write sizes that are a multiple of the block record size. If, during a read, a tape mark is encountered, the read returns zero. A further read will return an error. Thus it is not possible, as it is with the *character tape* interface, to read consecutive files without closing and reopening the nine-track tape device.

Open Operation

The *open* operation grants access to a tape drive. The *ta* device driver guarantees that only one process will be able to successfully open a tape drive. After a process has opened a tape drive, however, the process can *fork*(2) to create new processes having access to the same tape drive. While a process has a tape drive open, further requests to open this same tape drive will fail.

Close Operation

The *close* operation releases access of a tape drive acquired by the *open* operation. The *ta* device driver automatically places two consecutive tape marks on the tape at the current tape position to designate the logical end-of-tape when the tape drive is closed if and only if either of the following criteria are met:

- The tape drive was opened for write only, see *open*(2), and the boot-time tunable parameter `ta_force_EOF_on_close` is non-zero, the default. It is possible to satisfy this condition without writing to the tape.
- The tape drive was opened for writing and the last tape movement operation executed was a write operation. Given this condition, it is possible to write to the tape and *not* have tape marks automatically placed on the tape by the *ta* device driver. This can be done by repositioning the tape before closing but after the last write operation. In this case the user application is responsible for placing two consecutive tape marks on the tape to designate the logical end-of-tape.

If a rewind tape device is being closed, the tape will be rewound after tape marks are possibly written, see criteria above. If tape marks are written during the close and the drive being closed is not a rewind device, the *ta* device driver will position the tape drive head between the two tape marks.

Ioctl Operation

The following *ioctl* operations, as defined in *mtio*(4), are available only on character tape devices: MTIOCTOP, MTIOCGET, MTIOCLSTAT, MTIOCRTY, MTIOCNRTY, MTIOCEOT, and MTIOCNEOT. The supported MTIOCTOP operations are write tape mark, forward space a file, backward space a file, forward space a record, backward space a record, rewind, rewind and unload tape, no operation, erase gap, and drive clear.

Device-specific Interface

The *ta* device driver adds two extensions to the *mtio*(4) interface for the STK 2922, nine-track tape drive (MTD-004 and MTD-204). This tape drive has the unique capability, as compared to other *ta*-supported tape drives, to change the rate of inches per second (ips) at which tapes can be processed. The available rates on the STK 2922 nine-track tape drive are 50 and 100 ips. The *ta* device driver has extended the *mtio*(4) interface by adding two ioctls, one to select 50 ips and another to select 100 ips operation.

Selecting 50 Inches per Second Operation

The default speed of the STK 2922 tape drive is typically 50 ips, which is selectable on the IF Card in the STK 2922 tape drive. The ioctl for selecting 50 ips is MTIOC50IPS. The C code fragment below demonstrates the use of this ioctl.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/mtio.h>

int fd = 0;

if ((fd = open("/dev/rmt20", O_RDWR)) == -1) {
   perror("open");
   exit(1);
}
if (ioctl(fd, MTIOC50IPS, (char *)0) == -1) {
   perror("MTIOC50IPS");
   exit(1);
}
```

Selecting 100 Inches per Second Operation

The STK 2922 is a *streaming* tape drive when in the 100 ips mode. To effectively use the 100 ips mode, read and write operations must be streamed to the tape drive. In other words, the next read or write request must be issued to the tape drive before the tape travels to the latest point in the inter-record gap, the space between records, so that a reposition cycle is not required. The time required to traverse the inter-record gap to this position is the reinstruct time. The length of the reinstruct time differs for different densities but is always less than five milliseconds. If the reinstruct time is not met, the tape drive must reposition the tape, and the time required to reposition the tape is approximately fifty-three milliseconds. Thus, if the tape drive is in the streaming mode and read and write operations are not streamed to the tape drive, performance can be significantly impacted, especially when using small record sizes. The ioctl for selecting 100 ips is MTIOC100IPS. The C code fragment below demonstrates the use of this ioctl.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/mtio.h>

int fd = 0;

if ((fd = open("/dev/rmt20", O_RDWR)) == -1) {
   perror("open");
   exit(1);
```

```
}
if (ioctl(fd, MTIOC100IPS, (char *)0) == -1) {
  perror("MTIOC100IPS");
  exit(1);
}
```

TUNABLE PARAMETERS

There are two boot-time tunable parameters supported by the *ta* device driver. These are `ta_force_EOF_on_close` and `number_ta_iop_wndw`. These parameters are discussed in detail in the *Kernel Boot-Time Parameters* section of the *ConvexOS System Managers Guide.*

FILES

The *ta* device driver supports up to thirty-two physical nine-track tape drives. Each physical tape drive can have twelve different device file names. Since most installations confine tape use to physical tape devices zero through three, this section covers only those drives. Table 1 contains the names of the twelve different device file names for physical tape drives zero through three.

Table 1

MODE	DENSITY	REWIND	UNIT 0	UNIT 1	UNIT 2	UNIT 3
Block	800 bpi	Yes	mt0	mt1	mt2	mt3
Block	800 bpi	No	mt4	mt5	mt6	mt7
Block	1,600 bpi	Yes	mt8	mt9	mt10	mt11
Block	1,600 bpi	No	mt12	mt13	mt14	mt15
Block	6,250 bpi	Yes	mt16	mt17	mt18	mt19
Block	6,250 bpi	No	mt20	mt21	mt22	mt23
Character	800 bpi	Yes	rmt0	rmt1	rmt2	rmt3
Character	800 bpi	No	rmt4	rmt5	rmt6	rmt7
Character	1,600 bpi	Yes	rmt8	rmt9	rmt10	rmt11
Character	1,600 bpi	No	rmt12	rmt13	rmt14	rmt15
Character	6,250 bpi	Yes	rmt16	rmt17	rmt18	rmt19
Character	6,250 bpi	No	rmt20	rmt21	rmt22	rmt23

In general, the following expression, using operators from the C programming language, determines the *mt* and *rmt* numbers of nine-track tape device file names:

$$(((unit\ /\ 4) * 32) + (unit\ \%\ 4) + (density * 8) + (norwd * 4))$$

where

unit is the number of the physical nine-track tape unit.
density is zero for 800 bpi, one for 1,600 bpi, and two for 6,250 bpi.
norwd is zero for a rewind device and one for a no-rewind device.

ERRORS

Any of the nine-track tape operations: open, close, read, write, or ioctl, will fail if one or more of the following is true:

[EACCES] The mounted tape is a *labeled* tape and the tape system has refused to grant the user access to the tape. See the *ConvexOS Tape System Guide* for more information about labeled tapes.

[EIO] The tape drive is not online.

[EIO] The tape drive is not ready.

[EIO] The *ta* device driver was not able to allocate an internal message resource.

[EIO] The *ta* device driver was not able to communicate with the underlying CCU tape device driver.

Open Errors
In addition to the errors returned by the open system call, see *open*(2), the open operation will fail if one or more of the following is true:

[ENXIO] The tape drive is currently open by another process.

[ENXIO] The tape drive was not successfully booted. Refer to the **DIAGNOSTIC** section below to determine the cause of the boot failure.

[EIO] An attempt was made to open the tape drive for writing, but the mounted tape does not contain a write ring.

[EIO] The *ta* device driver was unable to determine the status of the tape drive. A diagnostic error message will be printed on the console; refer to the **DIAGNOSTIC** section for interpretation of this message.

Close Errors
In addition to the errors returned by the close system call, see *close*(2), the close operation will fail if one or more of the following is true:

[EIO] The *ta* device driver could not successfully place the tape marks on the tape. A diagnostic error message will be printed on the console; refer to the **DIAGNOSTIC** section for interpretation of this message.

[EIO] The *ta* device driver could not successfully backspace between the two tape marks just written. A diagnostic error message will be printed on the console; refer to the **DIAGNOSTIC** section for interpretation of this message.

[EIO] The *ta* device driver could not successfully start a rewind operation. A diagnostic error message will be printed on the console; refer to the **DIAGNOSTIC** section for interpretation of this message.

Read Errors
In addition to the errors returned by the read system call, see *read*(2), the read operation will fail if one or more of the following is true:

[ENXIO] *Block Tape Interface Only:* An attempt was made to read a non-existent record, for example, trying to read record seven of a file containing only five records.

[EIO] *Block Tape Interface Only:* The *ta* device driver could not position the tape at the record requested. A diagnostic error message will be printed on the console; refer to the **DIAGNOSTIC** section for interpretation of this message.

[EINVAL] *Character Tape Interface Only:* The requested transfer size is larger than the maximum transfer size supported by the *ta* device driver.

[EIO] An attempt was made to start a read operation with the tape positioned past the physical end-of-tape mark when read operations past the physical end-of-tape mark are not enabled. Read operations past the physical end-of-tape mark are enabled with the ioctl `MTIOCNEOT`.

[EIO] A tape drive error or a media error was detected. A diagnostic error message will be printed on the console; refer to the **DIAGNOSTIC** section for interpretation of this message.

Write Errors
In addition to the errors returned by the write system called, see *write*(2), the write operation will fail if one or more of the following is true:

[EIO] *Block Tape Interface Only:* The *ta* device driver could not position the tape at the record requested. A diagnostic error message will be printed on the console; refer to the DIAGNOSTIC section for interpretation of this message.

[EINVAL] *Character Tape Interface Only:* The requested transfer size is larger than the maximum transfer size supported by the tape drive.

[EIO] An attempt was made to start a write operation with the tape positioned past the physical end-of-tape mark when write operations past the physical end-of-tape mark are not enabled. Write operations past the physical end-of-tape mark are enabled with the ioctl `MTIOCNEOT`.

[EIO] A tape drive error or a media error was detected. A diagnostic error message will be printed on the console; refer to the DIAGNOSTIC section for interpretation of this message.

Ioctl Errors

In addition to the errors returned by the ioctl system call, see *ioctl*(2), the ioctl operation will fail if one or more of the following is true:

[EINVAL] Requested a `MTIOCTOP` ioctl with the count field, `mt_count`, of the `mtop` structure less than one.

[ENXIO] The requested ioctl command is either invalid or not supported by the *ta* device driver.

[ENXIO] An attempt was made to issue either the `MTIOC50IPS` or the `MTIOC100IPS` ioctl on a tape driver other than a STK 2922.

[EIO] An attempt was made to start a command with the tape physically positioned at the beginning of the tape and the command is not valid when the tape is positioned at the beginning of the tape.

[EIO] An attempt was made to start a command with the tape positioned past the physical end-of-tape mark, the end-of-tape mark is not currently ignored, and the command is not valid when the tape is positioned past the end-of-tape mark. Ioctl operations past the physical end-of-tape mark are enabled with the ioctl `MTIOCNEOT`.

[EIO] A forward space a record command or backward space a record command was issued and on completion of the command the tape was positioned immediately before or after a tape mark, respectively.

[EIO] A tape drive error or a media error was detected. A diagnostic error message will be printed on the console; refer to the DIAGNOSTIC section for interpretation of this message.

SEE ALSO

mt(1), tar(1), tpmount(1), mtio(4), tc(4), dump(8), sysgen(8), tpconfig(8)

DIAGNOSTICS

The following list explains the possible error messages that can be issued by the *ta* device driver.

Notations

%d Expands to a decimal number
%x Expands to a hexadecimal number
%b Expands to a hexadecimal number followed by a description of the bits set in the number
%s Expands to a text string

Some of the error messages contain multiple occurrences of the same notation. For example, the error message `ta%d: command %s not issued, command %s in execution level` contains two text strings, the `%s` notation. In these types of error messages it is not implied that each

occurrence of the notation must expand to the same value. Using the above example, the following is a *valid* error message: `ta%d: command NOP not issued, command WRITE in execution level.`

Boot-Time Diagnostics
The following messages may appear as the system is being booted:

`ta: maximum number of ta tape drives exceeded.`
The maximum number of *ta* tape drives has been reached. Either the system must be re-generated with *sysgen*(8) to add more *ta* tape drives, or the number of *ta* tape drives in the *ioconfig* file must be decreased.

`ta: not enough windows available.`
The *ta* device driver on the CCU could not allocate enough CCU windows to operate. The *ta* device driver allocates eight to thirty-four Multibus CCU windows for each Multibus nine–track tape controller; this number is controlled by the boot-time tunable parameter `number_ta_iop_wndw`. The *ta* device driver allocates thirty-four VMEbus CCU windows for each VMEbus nine–track tape controller. The configuration for the CCU printing this message, as specified in the *ioconfig* file, is too large. Some of the controllers or units located on this CCU must be removed from this CCU.

`ta: CCU local memory allocation failed.`
The *ta* device driver could not allocate enough local memory on the CCU. The *ta* device driver allocates thirteen pages of local CCU memory for each Multibus nine–track tape controller. The *ta* device driver allocates twelve pages of local CCU memory for each VMEbus nine–track tape controller. The configuration for the CCU printing this message, as specified in *ioconfig* file, is too large. Some of the controllers or units located on this CCU must be removed from this CCU.

Run-Time Diagnostics
The following messages may occur at any time following booting:

`ta%d: cmd: %s csb: %b esb: %b fsb: %b mux3: %b`
`ta%d: mux0: %x mux1: %x mux2: %x mux3: %x`
This message is printed when tape drive %d fails to successfully complete a request from the tape device driver. Interpretation of this message is as follows:

cmd Defines the *command* that was issued to the tape drive. Table 2 list the commands.

Table 2

MNEMONIC	COMMAND DESCRIPTION
NOP	No operation
CLEAR	Drive clear
DIAG	Diagnostic mode set
SENSE	Sense drive status
RD FWD	Read forward a record
RD BWD	Read backward a record
WRITE	Write data record
LOOP	Loop write-to-read
BS FILE	Backspace a tape mark

MNEMONIC	COMMAND DESCRIPTION
BS BLK	Backspace a data record
FS FILE	Forward space a tape mark
FS BLK	Forward space a data record
WRITE TM	Write tape mark
ERASE	Erase gap
REWIND	Rewind to load point
UNLOAD	Rewind and unload

csb Defines the *controller status byte* that was generated by the tape controller. The value of the **csb** can be 0x00 through 0xFF. Table 3 shows the bit definitions of the **csb**.

Table 3

BIT	MNEMONIC	DESCRIPTION
7	EXEC	A command is currently in execution.
6	PEND	A command has been loaded into the pending level of the controller.
5	REQ	The controller is currently requesting an interrupt.
4	NXT	Indicates the next transfer address parameter will be loaded from the *A* buffer. During normal transfers, this bit is always set, as the *B* buffer is only used for *chain mode* transfers. During *chain mode* transfers, this bit will toggle each time the execution level is reloaded.
3	ENA	Interrupts are enabled. An interrupt will be requested upon completion of an executing command.
2	LOOP	The controller is executing in diagnostic loop-back mode. DMA transfers will be transferred to the controller fifo but will not be transferred to the selected tape drive.
1	PAR	The controller will force parity errors.
0	CHAIN	The controller is executing in *chain mode*.

esb Defines the *error status byte* that was generated by the tape controller. The value of the **esb** can be 0x00 through 0xFF. Table 4 shows the bit definitions of the **esb**.

Table 4

BIT	MNEMONIC	DESCRIPTION
7-6	DENSITY	Density last sent to the tape formatter; not necessarily the density currently in use. Values can be: 00 = PE (1,600 bpi); 01 = GCR (6,250 bpi); or 10 = NRZ1 (800 bpi).
5	TM	Tape mark detected. The tape head has just passed over a tape mark.
4	Multibus Only LAST_BYTE_ INVALID	The last byte written to memory was invalid.
	VMEbus Only READ_UNDER_ RUN	During a read operation, the number of bytes requested was less than the number of bytes in the record on the tape.

BIT	MNEMONIC	DESCRIPTION
3	ABORT	Command aborted; the pending command was aborted due to an error during previous command.
2	Multibus Only MBUS_BUS_ERR	Bus timeout error during attempted DMA transfer.
	VMEbus Only VME_BUS_ERR	Bus timeout error during attempted DMA transfer.
1	RD_PAR	Read parity error. The data byte from tape formatter was parity incorrect.
0	FMT_ERR	Tape error. See the **fsb** and **mux1** to determine the cause.

fsb Defines the *formatter status byte* that was generated by the tape formatter. The value of the **fsb** can be 0x00 through 0xFF. Table 5 shows the bit definitions of the **fsb**.

Table 5

BIT	MNEMONIC	DESCRIPTION
7	CMD_INC	Command incomplete; tape formatter and tape drive are unable to finish the command.
6	CMDREJECTED	Command reject; the previous command was not initiated. See **mux2** to determine *Reject Code.*
5	OVERRUN	Data overrun during a read or write operation.
4	DATA_CHK	Data check. Examine mux bytes to determine reason.
3	PROM	PROM error. The tape formatter parity checker detected an error during command execution or, if an STK 2920 Series tape drive is used, the formatter detected a PROM checksum error during power-up.
2	CORR	Correctable error has been detected.
1	BUS_ERR	Data bus parity error. This is an uncorrectable data error between tape drive and tape formatter.
0	Multibus revision J or greater only READ_UNDER_RUN	During a read operation, the number of bytes requested was less than the number of bytes in the record on the tape.
	Multibus revision less than J or VMEbus Not Used	Reserved.

mux0 Defines the contents of the tape formatter's *dead track register.* Each bit in **mux0** equates to a specific track on the tape, as shown below in Table 6.

Table 6

MUX0 Bit	8	7	6	5	4	3	2	1	0
Dead Track	P	7	6	5	4	3	2	1	0

mux1 Defines the contents of the tape formatter's *read/write error register.* Each bit in **mux1** defines a particular type of read or write error condition. Table 7 shows the bit descriptions of **mux1**.

Table 7

MUX1 Bit	DESCRIPTION
8 *	Cyclic redundancy check error. If an STK 1960 Series tape drive is used, this can be asserted during GCR, PE, or NRZ1 read or write operations. If an STK 2920 Series tape drive is used, this can be asserted during GCR read/write or PE write operations.
7	Write tape mark check; the tape formatter was unable to write a valid tape mark. See **NOTE 1** for additional information.
6	Uncorrectable error. See **NOTE 2** for additional information.
5 *	Partial record; the inter-record gap was detected before the end of data.
4	Multiple track error in PE or GCR mode, or longitudinal redundancy check error if in NRZ1 mode on an STK 1960 Series tape drive. See **NOTE 3** for additional information.
3	Not used.
2 *	End data check. When the operation is a PE or GCR read or write, the end-of-data characters were not detected or the preamble or postamble did not meet format requirements. If the operation was a NRZ1 read or write on an STK 1960 Series tape drive, a vertical redundancy check error was present. The vertical redundancy check is asserted whenever at least one byte of data, the CRC character, or the longitudinal redundancy check character is parity incorrect.
1 *	Velocity error is asserted whenever the tape speed is outside acceptable limits.
0	Diagnostics mode latch. The tape formatter is in diagnostics mode.

* *Data check*, **fsb** bit 4, will also be set.

NOTE 1 *Write tape mark check*, **mux1** bit 7, status will be accompanied by the assertion of *data check*, **fsb** bit 4, or *command reject*, **fsb** bit 6, which will further define the error, as shown below in Table 8.

Table 8

DENSITY IN USE	ASSERTED BITS			DESCRIPTION
	MUX1 7	FSB 4	FSB 6	
PE or GCR	X	X		The tape mark is readable as such, but does it **not** conform to ANSI standards.
PE, GCR, or NRZ1	X		X	The tape mark written is **not** readable as such. Noise is left on the tape.
NRZ1	X	X		The tape mark is **not** readable as such. The data left on the tape appears as an illegally-formatted record.

NOTE 2 *Uncorrectable error*, **mux1** bit 6, status will be accompanied by the assertion of *data check*, **fsb** bit 4, during PE, GCR, or NRZ1 read or write operations. Also, if the NRZ1 mode is being used, bits 8, 7, and 6 of the *dead track register*, **mux0**, will be asserted.

NOTE 3 *Multiple track error*, **mux1** bit 4, is asserted when two or more tracks are detected in error. *Data check*, **fsb** bit 4, will also be asserted if the error is detected during a PE read or write operation or any GCR write operation. If the operation is a GCR read, *data check* will only be asserted if the errors are not correctable.

mux2 The most significant bit of **mux2**, bit eight, is the digital tachometer line from the tape drive for use by certain diagnostic routines. Bits 7, 6, and 5 are for diagnostic mode use only. Bits 4 through 0 define **reject codes** as shown below in Table 9. The valid range of bits 4 through 0 is 0x01 through 0x1F.

Table 9

CODE	REJECT CODE DESCRIPTION
0x01	The addressed tape unit is not ready.
0x02 †	The tape formatter control unit has detected an internal firmware parity error.
0x03 †	TRAK responses to the first TREQs not received within 75 milliseconds during a write.
0x04 †	2920 Series - Reserved. **1960 Series Only** - Unimplemented word detected in microprogram.
0x05	File protect status when a write-type command was attempted.
0x06 †	The addressed tape unit did not go to erase status only.
0x07	Illegal command sequence.
0x08 †	The addressed tape unit did not go to read status.
0x09 †	The addressed tape unit was unable to read a PE or GCR identification burst.
0x0A †	2920 Series - Reserved. **1960 Series Only** - The addressed tape unit did not drop write-inhibit status.
0x0B	2920 Series - Reserved. **1960 Series Only** - The addressed tape unit not on-line.
0x0C †	The addressed tape unit did not go to write status after a write command.
0x0D †	2920 Series - Reserved. **1960 Series Only** - During backward motion after writing the identification burst, BOT was not reached in the expected distance.
0x0E †	2920 Series - Reserved. **1960 Series Only** - The addressed tape unit did not go to backward status.
0x0F †	Noise, possibly data, was detected during an erase gap write following a read command.
0x10	1960 Series - Reserved. **2920 Series Only** - No data is in the buffer for a read in cache data mode.
0x11 †	The addressed tape unit has reset its ready status.
0x12 †	2920 Series - Reserved. **1960 Series Only** - PE of GCR identification burst written on wrong track(s).
0x13	A backward-type command, except rewind or unload, was given and tape was already at BOT.
0x14 †	The ARA burst portion of the GCR identification burst did not have all nine tracks active.
0x15 †	An inter-record gap longer than 25 feet (PE/NRZI) or 15 feet (GCR) was detected during a read or space command.
0x16	1960 Series - Reserved. **2920 Series Only** - More than one speed change request with no intervening motion.

CODE	REJECT CODE DESCRIPTION
0x17 †	2920 Series - Reserved. **1960 Series Only** - The addressed tape unit failed to reset ready during a rewind.
0x18 †	The addressed tape unit does not indicate being in the selected density.
0x19	1960 Series - Reserved. **2920 Series Only** - Loop write-to-read attempted with tape loaded and not at BOT.
0x1A †	The addressed tape unit failed to initiate tape motion.
0x1B †	During read-after-write, data was detected in the inter-record gap.
0x1C †	2920 Series - Reserved. **1960 Series Only** - No inter-record gap was detected following the ARA identification burst.
0x1D †	The tape drive attempted to backspace over a bad record but was unable to detect the record.
0x1E †	The ARA identification burst was unreadable during a GR write command.
0x1F †	No data was detected during a read-after-write while in write or write-tape-mark mode.

† *Command incomplete*, **fsb** bit 7, will also be set.

mux3 Defines the contents of the status lines from the addressed tape drive. Table 10 shows the interpretation of **mux3**.

Table 10

BIT	MNEMONIC	STATUS DESCRIPTION
8	WRTS	Write tape status (**2920 Series Only**; not used in 1960 Series). The tape drive is in write mode.
7	EOTS	End-of-tape status. The tape is positioned at or beyond the end-of-tape mark.
6	BOTS	Beginning-of-tape status. The tape is positioned at BOT, the load point.
5	WRTINH	Write inhibit (**1960 Series Only**; not used in 2920 Series). Asserted in write mode when the read/write head is positioned in the inter-record gap.
4	FPTS	File-protect status. No write enable ring is installed in the tape reel.
3	BWDS	Backward status. Asserted when the command in progress requires backward tape motion and the tape is in motion.
2	HDNS	High-density status. The tape drive is in GCR mode.
1	RDYS	Ready status. The tape is loaded and drive is able to accept a command.
0	ONLS	On-line status. The tape is loaded and the on-line switch on the operator control panel has been depressed.

`ta%d: not online.`
The tape drive %d was accessed while the tape drive was off-line. The tape drive should be made online and ready.

`ta%d: not ready.`
The tape drive %d was accessed while the tape drive was not ready. The tape drive should be made online and ready.

`ta%d: no write ring.`
The tape drive %d was opened for writing and the mounted tape does not contain a write ring. Unload the tape, add a write ring, and remount the tape.

`ta%d: velocity error.`
The tape drive %d reported an error in which the tape drive's velocity is outside acceptable limits. This error only occurs only during write commands. This condition is the result of a tape drive failure. Hardware maintenance is required.

`ta%d: chain overrun%d.`
During a read or write operation on tape drive %d, the CCU *ta* device driver was not able to service interrupts from the tape controller quickly enough. When this condition occurs, the CCU *ta* device driver will retry the operation. This condition typically occurs more often on the Multibus tape controller. Increasing the boot-time tunable parameter `number_ta_iop_wndw` can reduce the occurrence of this condition for Multibus tape controllers.

`ta%d: command %s not issued, command %s in execution level.`
A command for tape drive %d was loaded into the controller and started, but the CCU *ta* device driver never saw the command execute and the controller did not issue an interrupt. This condition is the result of a hardware controller failure or a tape drive failure. Hardware maintenance is required.

`ta%d: tape controller chassis %d ccu %d slot %d is not responding.`
A command for tape drive %d was loaded into the controller and started, but the command never completed, and the controller is hung. This condition is the result of a hardware controller failure or a tape drive failure. Hardware maintenance is required.

`ta%d: unable to free message %x.`
The *ta* device driver was unable to free an internal message resource used to execute a command on tape drive %d. This condition is a result of a software failure. A *crashdump*(8) should be taken.

`ta%d: unable to return message %x.`
The CCU *ta* device driver was unable to return an internal message resource to the CPU that was used to execute a command on tape drive %d. This condition is a result of a software failure. A *crashdump*(8) should be taken.

`ta: dq_recv error on subqueue %d.`
The CCU *ta* device driver was unable to receive an internal message resource from the CPU. This condition is a result of a software failure. A *crashdump*(8) should be taken.

`ta: spurious interrupt for MBTC`
The Multibus tape controller issued an interrupt when the CCU *ta* device driver was not expecting an interrupt from the controller. This condition is the result of a hardware controller failure or a tape drive failure. Hardware maintenance is required.

NAME

tc – cartridge tape driver

SYNOPSIS

ctlr MTC-202 csr 0x%x int %d
unit %d subunit %d type MTD-207 (3480-compatible tape drive)

DESCRIPTION

tc provides a standard cartridge tape drive interface as described in *mtio*(4). MTC-202 is the VMEbus SCSI Host Adapter.

DEVICE SPECIFIC IOCTL

There is one special operation *MTBLANK* of ioctl MTIOCTOP available for the 3480-compatible cartridge tape drive. The operation *MTBLANK* allows the remaining medium to be erased. To use the operation the following include files must be used.

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

The operation *MTBLANK* of ioctl MTIOCTOP erases the remaining medium.

SEE ALSO

mt(1), tar(1), mtio(4), tpconfig(8), opreq(1)

BUGS

If the cartridge tape drive is reset while tape writing is in progress, all subsequent tape commands will fail, and a media error message will be returned. To clear the media error, cycle the power of the formatter. When the formatter is turned back on, it will perform the self-test and unload the tape cartridge in each tape drive.

NAME

tcp – Internet Transmission Control Protocol

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_STREAM, 0);

DESCRIPTION

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. TCP uses the standard Internet address format and, in addition, provides a per-host collection of "port addresses". Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets utilizing the tcp protocol are either "active" or "passive". Active sockets initiate connections to passive sockets. By default TCP sockets are created active; to create a passive socket the *listen*(2) system call must be used after binding the socket with the *bind*(2) system call. Only passive sockets may use the *accept*(2) call to accept incoming connections. Only active sockets may use the *connect*(2) call to initiate connections.

Passive sockets may "underspecify" their location to match incoming connection requests from multiple networks. This technique, termed "wildcard addressing", allows a single server to provide service to clients on multiple networks. To create a socket which listens on all networks, the Internet address INADDR_ANY must be bound. The TCP port may still be specified at this time; if the port is not specified the system will assign one. Once a connection has been established the socket's address is fixed by the peer entity's location. The address assigned the socket is the address associated with the network interface through which packets are being transmitted and received. Normally this address corresponds to the peer entity's network.

TCP supports one socket option which is set with *setsockopt*(2) and tested with *getsockopt*(2). Under most circumstances, TCP sends data when it is presented; when outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgement is received. For a small number of clients, such as window systems that send a stream of mouse events which receive no replies, this packetization may cause significant delays. Therefore, TCP provides a boolean option, TCP_NODELAY (from *<netinet/tcp.h>*, to defeat this algorithm. The option level for the *setsockopt* call is the protocol number for TCP, available from *getprotobyname*(3N).

Options at the IP transport level may be used with TCP; see *ip*(4P). Incoming connection requests that are source-routed are noted, and the reverse source route is used in responding.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN]	when trying to establish a connection on a socket which already has one;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[ETIMEDOUT]	when a connection was dropped due to excessive retransmissions;
[ECONNRESET]	when the remote peer forces the connection to be closed;
[ECONNREFUSED]	when the remote peer actively refuses connection establishment (usually because no process is listening to the port);
[EADDRINUSE]	when an attempt is made to create a socket with a port which has already been allocated;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

SEE ALSO

getsockopt(2), socket(2), intro(4N), inet(4F), ip(4P)

NOTES

Tcp is an optional product; for more information, contact your CONVEX sales representative.

NAME

termios – general terminal interface

SYNOPSIS

#include <sys/termios.h>

DESCRIPTION

Asynchronous communications ports, pseudo-terminals, and the special interface accessed by **/dev/tty** all use the same general interface, no matter what hardware (if any) is involved. The remainder of this section discusses the common features of this interface.

Opening a Terminal Device File

When a terminal file is opened, the process normally waits until a connection is established. (In practice, users' programs seldom open these files; they are opened by *getty*(8) and become a user's standard input, output, and error files.) If the **O_NDELAY** flag is set in the second argument to *open*(2), the open will complete immediately without waiting for a connection to be established.

Process Groups

A terminal may have a distinguished process group associated with it. This distinguished process group plays a special role in handling signal-generating input characters, as discussed below in the **Special Characters** section below.

A command interpreter, such as *csh*(1), that supports "job control" can allocate the terminal to different *jobs*, or process groups, by placing related processes in a single process group and associating this process group with the terminal. A terminal's associated process group may be set or examined by a process with sufficient privileges. The terminal interface aids in this allocation by restricting access to the terminal by processes that are not in the current process group; see **Job Access Control** below.

The Controlling Terminal

A terminal may belong to a process as its *controlling terminal*. If a process that is a "session process group leader," and that does not have a controlling terminal, opens a terminal file not already associated with a process group, the terminal associated with that terminal file becomes the controlling terminal for that process, and the terminal's distinguished process group is set to the process group of that process. (Currently, this also happens if a process that does not have a controlling terminal and is not a member of a process group opens a terminal. In this case, if the terminal is not associated with a process group, a new process group is created with a process group ID equal to the process ID of the process in question, and the terminal is assigned to that process group. The process is made a member of the terminal's process group.)

The controlling terminal is inherited by a child process during a *fork*(2). A process relinquishes its control terminal when it changes its process group using *setpgrp*(2), when it becomes the leader of a new session using *setsid*(2), or when it issues a **TIOCNOTTY** ioctl call on a file descriptor associated with its controlling terminal.

When a session process group leader that has a controlling terminal terminates, the distinguished process group of the controlling terminal is set to zero (indicating no distinguished process group). This allows the terminal to be acquired as a controlling terminal by a new session process group leader.

Closing a Terminal Device File

When a terminal device file is closed, the process closing the file waits until all output is drained; all pending input is then flushed, and finally a disconnect is performed. If **HUPCL** is set, the existing connection is severed (by hanging up the phone line, if appropriate).

Job Access Control

If a process is in the (non-zero) distinguished process group of its controlling terminal, or if the terminal's distinguished process group is zero (if either of these are true, the process is said to be a *foreground process*), then read operations are allowed as described below in **Input Processing and Reading Characters**. If a process is not in the (non-zero) distinguished process group of its controlling terminal (if this is true, the process is said to be a *background process*), then any attempts to read from that terminal will send that process' process group a SIGTTIN signal unless the process is ignoring SIGTTIN, has SIGTTIN blocked, or is in the middle of process creation using *vfork*(2); in that case, the read will return −1 and set **errno** to EIO, and the SIGTTIN signal will not be sent. The SIGTTIN signal will normally stop the members of that process group.

When the TOSTOP bit is set in the **c_lflag** field, attempts by a background process to write to its controlling terminal will send that process' process group a SIGTTOU signal unless the process is ignoring SIGTTOU, has SIGTTOU blocked, or is in the middle of process creation using *vfork*(2); in that case, the process will be allowed to write to the terminal and the SIGTTOU signal will not be sent. The SIGTTOU signal will normally stop the members of that process group. Certain ioctl calls that set terminal parameters are treated in this same fashion, except that TOSTOP is not checked; the effect is identical to that of terminal writes when TOSTOP is set. See IOCTLS and the *tty*(4) man page.

Input Processing and Reading Characters

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. If the IMAXBEL mode has not been selected, all the saved characters are thrown away without notice when the input limit is reached; if the IMAXBEL mode has been selected, the driver refuses to accept any further input, and echoes a bell (ASCII BEL).

Two general kinds of input processing are available, determined by whether the terminal device file is in canonical mode or non-canonical mode (see ICANON in the **Local Modes** section).

The style of input processing can also be very different when the terminal is put in non-blocking I/O mode; see *read*(2). In this case, reads from the terminal will never block.

It is possible to simulate terminal input using the TIOCSTI ioctl call, which takes as its third argument the address of a character. The system pretends that this character was typed on the argument terminal, which must be the process' controlling terminal unless the process' effective user ID is super-user.

Canonical Mode Input Processing

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a NEWLINE (ASCII LF) character, an EOF (by default, an ASCII EOT) character, or one of two user-specified end-of-line characters, EOL and EOL2. This means that a *read*(2) will not complete until an entire line has been typed or a signal has been received. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read an entire line at once; any number of characters may be requested in a read, even one, without losing information.

Erase and kill processing occurs during input. The ERASE character (by default, the character DEL) erases the last character typed in the current input line. The WERASE character (by default, the character CTRL-W) erases the last "word" typed in the current input line (but not any preceding SPACE or TAB characters). Normally, a "word" is defined as a sequence of non-blank characters, with TAB characters counted as blanks. If ALTWERASE is set, a sequence of alphanumeric and underscore characters is treated as a "word," separated by punctuation, control and white-space characters. See the **Local Modes** section for information on the alternate word erase algorithm. Neither ERASE nor WERASE will erase beyond the beginning of the line.

The KILL character (by default, the character CTRL-U) kills (deletes) the entire current input line, and optionally outputs a NEWLINE character. All these characters operate on a key-stroke basis, independent of any backspacing or tabbing that may have been done.

The REPRINT character (the character CTRL-R) prints a NEWLINE followed by all characters that have not been read. Reprinting also occurs automatically if characters that would normally be erased from the screen are fouled by program output. The characters are reprinted as if they were being echoed; as a consequence, if ECHO is not set, they are not printed.

The ERASE and KILL characters may be entered literally by preceding them with the escape character (\). In this case the escape character is not read. The ERASE and KILL characters may be changed.

Non-Canonical Mode Input Processing

In non-canonical mode input processing, input characters are not assembled into lines, and erase and kill processing does not occur. The MIN and TIME values are used to determine how to process the characters received.

MIN represents the minimum number of characters that should be received when the read is satisfied (when the characters are returned to the user). TIME is a timer of 0.10 second granularity that is used to timeout bursty and short term data transmissions. The four possible values for MIN and TIME and their interactions are described below.

Case A: MIN > 0, TIME > 0

In this case TIME serves as an intercharacter timer and is activated after the first character is received. Since it is an intercharacter timer, it is reset after a character is received. The interaction between MIN and TIME is as follows: as soon as one character is received, the intercharacter timer is started. If MIN characters are received before the intercharacter timer expires (remember that the timer is reset upon receipt of each character), the read is satisfied. If the timer expires before MIN characters are received, the characters received to that point are returned to the user. Note: if MIN expires, at least one character will be returned because the timer would not have been enabled unless a character was received. In this case (MIN > 0, TIME > 0), the read will sleep until the MIN and TIME mechanisms are activated by the receipt of the first character.

Case B: MIN > 0, TIME = 0

In this case, since the value of TIME is zero, the timer plays no role and only MIN is significant. A pending read is not satisfied until MIN characters are received (the pending read will sleep until MIN characters are received). A program that uses this case to read record-based terminal I/O may block indefinitely in the read operation.

Case C: MIN = 0, TIME > 0

In this case, since MIN = 0, TIME no longer represents an intercharacter timer. It now serves as a read timer that is activated as soon as a read function is processed. A read is satisfied as soon as a single character is received or the read timer expires. Note: in this case if the timer expires, no character will be returned. If the timer does not expire, the only way the read can be satisfied is if a character is received. In this case the read will not block indefinitely waiting for a character; if no character is received within TIME*0.10 seconds after the read is initiated, the read will return with zero characters.

Case D: MIN = 0, TIME = 0

In this case, return is immediate. The minimum of either the number of characters requested or the number of characters currently available will be returned without waiting for more characters to be input.

Comparison of the Different Cases of MIN, TIME Interaction

Some points to note about MIN and TIME:

1. In the following explanations, one may notice that the interactions of MIN and TIME are not symmetric. For example, when MIN > 0 and TIME = 0, TIME has no effect. However, in the opposite case where MIN = 0 and TIME > 0, both MIN and TIME play a role in that MIN is satisfied with the receipt of a single character.
2. Also note that in case A (MIN > 0, TIME > 0), TIME represents an intercharacter timer while in case C (MIN = 0, TIME > 0) TIME represents a read timer.

These two points highlight the dual purpose of the MIN/TIME feature. Cases A and B, where MIN > 0, exist to handle burst mode activity (for example, file transfer programs) where a program would like to process at least MIN characters at a time. In case A, the intercharacter timer is activated by a user as a safety measure; while in case B, it is turned off.

Cases C and D exist to handle single character timed transfers. These cases are readily adaptable to screen-based applications that need to know if a character is present in the input queue before refreshing the screen. In case C the read is timed; while in case D, it is not.

Another important note is that MIN is always just a minimum. It does not denote a record length. That is, if a program does a read of 20 bytes, MIN is 10, and 25 characters are present, 20 characters will be returned to the user.

Writing Characters

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 as they are typed if echoing has been enabled. 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.

Special Characters

Certain characters have special functions on input and/or output. These functions and their default character values are summarized as follows:

INTR (CTRL-C or ASCII ETX) generates a SIGINT signal, which is sent to all processes in the distinguished process group associated with the 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 *sigvec*(2).

QUIT (CTRL-| or ASCII FS) generates a SIGQUIT signal, which is sent to all processes in the distinguished process group associated with the terminal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.

ERASE (Rubout or ASCII DEL) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, EOL, or EOL2 character.

ERASE2 (disabled) is an additional erase character and performs the same function as ERASE.

WERASE (CTRL-W or ASCII ETB) erases the preceding "word". It will not erase beyond the start of a line, as delimited by a NL, EOF, EOL, or EOL2 character.

KILL (CTRL-U or ASCII NAK) deletes the entire line, as delimited by a NL, EOF, EOL, or EOL2 character.

REPRINT (CTRL-R or ASCII DC2) reprints all characters that have not been read, preceded by a NEWLINE.

EOF (CTRL-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 NEWLINE, and the EOF is discarded. 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 cannot be changed; it can, however, be escaped by the LNEXT character.

EOL EOL2 (disabled) are additional line delimiters, like NL. They are not normally used.

SUSP (CTRL-Z or ASCII SUB) is used by the job control facility to change the current job to return to the controlling job. It generates a SIGTSTP signal, which stops all processes in the terminal's process group.

DSUSP (CTRL-Y or ASCII EM) generates a SIGTSTP signal like SUSP but the signal is sent when a program attempts to read the character, rather than when it is typed.

STOP (CTRL-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 (CTRL-Q or ASCII DC1) is used to resume output that has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read.

FLUSHO (CTRL-O or ASCII SI) causes subsequent output to be discarded until another FLUSHO character is typed, more input arrives, or the condition is cleared by a program.

QUOTE (\) causes the special meaning of ERASE, ERASE2 and KILL to be ignored. The QUOTE character has no special meaning when followed by any character other than one of these three.

LNEXT (CTRL-V or ASCII SYN) causes the special meaning of the next character to be ignored; this works for all the special characters mentioned above. This allows characters to be input that would otherwise get interpreted by the system (for example, KILL, QUIT.)

DISABLE is the special character that disables the function of other special characters.

The character values for INTR, QUIT, ERASE, ERASE2, WERASE, KILL, REPRINT, EOF, EOL, EOL2, SUSP, STOP, START, FLUSHO, LNEXT, QUOTE, and DISABLE may be changed to suit individual tastes. If the value of a special control character is the value of the control character DISABLE, the function of that special control character (except DISABLE) will be disabled. The ERASE, ERASE2, and KILL characters may be escaped by a preceding QUOTE character, in which case no special function is done. Any of the special characters may be preceded by the LNEXT character, in which case no special function is done.

Modem Disconnect

If a modem disconnect is detected, and the CLOCAL flag is not set in the **c_cflag** field, a SIGHUP signal is sent to all processes in the distinguished process group associated with this terminal. Unless other arrangements have been made, this signal terminates the processes. If SIGHUP is ignored or caught, any subsequent *read*(2) returns with an end-of-file indication until the terminal is closed. Thus, programs that read a terminal and test for end-of-file can terminate appropriately after a disconnect. Any subsequent *write*(2) will return –1 and set **errno** to EIO until the terminal is closed.

Terminal Parameters

The parameters that control the behavior of devices and modules providing the **termios** interface are specified by the **termios** structure, defined by **<sys/termios.h>**. Several ioctl calls that get or change these parameters use this structure:

```
#define NCCS            32
typedef unsigned long   tcflag_t;
typedef unsigned long   speed_t;
typedef unsigned short  cc_t;

struct termios {
        tcflag_t  c_iflag;              /* input flags */
        tcflag_t  c_oflag;              /* output flags */
        tcflag_t  c_cflag;              /* control flags */
        tcflag_t  c_lflag;              /* local flags */
        tcflag_t  c_tflag;              /* tty flags and state */
        speed_t   c_ispeed;             /* input speed */
        speed_t   c_ospeed;             /* output speed */
        struct
#ifndef _POSIX_SOURCE
          winsize
#endif
          {
            unsigned short ws_row;      /* rows, in characters */
            unsigned short ws_col;      /* columns, in characters */
            unsigned short ws_xpixel;   /* horizontal size, pixels */
            unsigned short ws_ypixel;   /* vertical size, pixels */
          }       c_winsize;            /* window size */
        char      c_unused[32];         /* spares */
        cc_t      c_cc[NCCS];           /* 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	VEOF	^D	
1	VEOL	disabled	
2	VEOL2	disabled	
3	VERASE	^?	(DEL)
4	VERASE2	disabled	
5	VWERASE	^W	
6	VKILL	^U	
7	VREPRINT	^R	
8	VINTR	^C	
9	VQUIT	^\	(FS)
10	VSUSP	^Z	
11	VDSUSP	^Y	
12	VSTART	^Q	
13	VSTOP	^S	
14	VLNEXT	^V	
15	VFLUSHO	^O	
16	VMIN	1	
17	VTIME	0	
18	VQUOTE	\	
19	VDISABLE	0377	(0xff)

The calls that use the **termios** structure only affect the flags and control characters that can be stored in the **termios** structure; all other flags and control characters are unaffected.

Input Modes

The **c_iflag** field describes the basic terminal input control:

```
IGNBRK      0x00000001  /* ignore BREAK condition */
BRKINT      0x00000002  /* map BREAK to SIGINTR */
IGNPAR      0x00000004  /* ignore (discard) parity errors */
PARMRK      0x00000008  /* mark parity and framing errors */
INPCK       0x00000010  /* disable checking of parity errors */
ISTRIP      0x00000020  /* strip 8th bit off chars */
INLCR       0x00000040  /* map NL into CR */
IGNCR       0x00000080  /* ignore CR */
ICRNL       0x00000100  /* map CR to NL (ala CRMOD) */
IXON        0x00000200  /* enable output flow control */
IXOFF       0x00000400  /* enable input flow control */
IXANY       0x00000800  /* any char will restart after stop */
IMAXBEL     0x00002000  /* ring bell on input queue full */
IUCLC       0x00004000  /* map uppercase to lowercase */
```

If IGNBRK is set, a break condition (a character framing error with data all zeros) detected on input is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise, if BRKINT is set, a break condition will generate a SIGINT and flush both the input and output queues. If neither IGNBRK nor BRKINT is set, a break condition is read as a single ASCII NUL character ('\0').

If IGNPAR is set, characters with framing or parity errors (other than break) are ignored. Otherwise, if PARMRK is set, a character with a framing or parity error that is not ignored is read as the three-character sequence: '\377', '\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 '\377' is read as '\377', '\377'. If neither IGNPAR nor PARMRK is set, a framing or parity error (other than break) is read as a single ASCII NUL character ('\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 IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. The STOP and START characters will not be read, but will merely perform flow control functions. If IXON is cleared while output is suspended the system will restart output immediately. This is done to prevent a line from remaining stopped permanently since the START character will no longer restart output once IXON is cleared.

If IXANY is set, any input character will restart output that has been suspended. Only has an effect when set if IEXTEN is also set.

If IXOFF is set, the system will transmit a STOP character when the input queue is nearly full, and a START character when enough input has been read that the input queue is nearly empty again.

If IMAXBEL is set, the ASCII BEL character is echoed if the input stream overflows. Further input will not be stored, but any input already present in the input stream will not be disturbed. If IMAXBEL is not set, no BEL character is echoed, and all input present in the input queue is silently discarded if the input stream overflows. Only has an effect when set if IEXTEN is also set.

If IUCLC is set, a received uppercase alphabetic character is translated into the corresponding lowercase character. This is currently unimplemented.

The initial input control value is BRKINT, ISTRIP, IMAXBEL, IXON, IXANY, ICRNL.

Output Modes

The **c_oflag** field specifies the system treatment of output:

```
OPOST    0x00000001  /* enable following output processing */
ONLCR    0x00000002  /* map NL to CR-NL (ala CRMOD) */
OXTABS   0x00000004  /* expand tabs to spaces */
ONOEOT   0x00000008  /* discard EOT's (^D) on output) */
OLCUC    0x00000010  /* map lowercase to uppercase */
```

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair.

If OXTABS is set, tabs are expanded to the proper number of spaces.

If ONOEOT is set, the EOT character is discarded on output.

If OLCUC is set, a lowercase alphabetic character is transmitted as the corresponding uppercase character. This function is often used in conjunction with IUCLC. It is currently unimplemented.

The initial output control value is OPOST, ONLCR, OXTABS, ONOEOT.

The **c_cflag** field describes the hardware control of the terminal:

```
CSIZE    0x00000300  /* character size mask */
  CS5      0x00000000    /* 5 bits - pseudo */
  CS6      0x00000100    /* 6 bits */
```

```
    CS7         0x00000200  /* 7 bits */
    CS8         0x00000300  /* 8 bits */
 CSTOPB     0x00000400 /* send 2 stop bits */
 CREAD      0x00000800 /* enable receiver */
 PARENB     0x00001000 /* parity enable */
 PARODD     0x00002000 /* odd parity, else even */
 HUPCL      0x00004000 /* hang up on last close */
 CLOCAL     0x00008000 /* ignore modem status lines */
 CRTSCTS    0x00010000 /* RTS/CTS flow control */
```

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 modem control lines for the port will be disconnected when the last process with the line open closes it or terminates.

If CLOCAL is set, a connection does not depend on the state of the modem status lines. Otherwise modem control is assumed.

If CRTSCTS is set, and the terminal has modem control lines associated with it, the Request To Send (RTS) modem control line will be raised, and output will occur only if the Clear To Send (CTS) modem status line is raised. If the CTS modem status line is lowered, output is suspended until CTS is raised. Some hardware may not support this function, and other hardware may not permit it to be disabled; in either of these cases, the state of the CRTSCTS flag is ignored. This is currently unimplemented.

The initial hardware control value after open is CREAD, CS8, HUPCL.

Local Modes

The **c_lflag** field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline provides the following:

```
 ECHOKE     0x00000001 /* visual erase for line kill */
 ECHOE      0x00000002 /* visually erase chars */
 ECHOK      0x00000004 /* echo NL after line kill */
 ECHO       0x00000008 /* enable echoing */
 ECHONL     0x00000010 /* echo NL even if ECHO is off */
 ECHOPRT    0x00000020 /* visual erase mode for hardcopy */
 ECHOCTL    0x00000040 /* echo control chars as ^(Char) */
 ISIG       0x00000080 /* enable signals INTR, QUIT, [D]SUSP */
 ICANON     0x00000100 /* canonicalize input lines */
 ALTWERASE  0x00000200 /* use alternate WERASE algorithm */
 IEXTEN     0x00000400 /* extended functions enabled */
 XCASE      0x00004000 /* case translation */
 MDMBUF     0x00100000 /* flow control output via Carrier */
 TOSTOP     0x00400000 /* stop background jobs from output */
 FLUSHO     0x00800000 /* output being flushed (state) */
 NOHANG     0x01000000 /* XXX this should go away */
 PENDIN     0x20000000 /* retype pending input (state) */
 NOFLSH     0x80000000 /* don't flush after interrupt */
```

If ISIG is set, each input character is checked against the special control characters INTR, QUIT, SUSP, and DSUSP. 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.

If ICANON is set, canonical processing is enabled. This enables the erase, word erase, kill, and reprint edit functions, and the assembly of input characters into lines delimited by NL, EOF, EOL, and EOL2. 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 or the timeout value TIME has expired between characters. This allows fast bursts of input to be read efficiently while still allowing single character input. The time value represents tenths of seconds. See the **Non-canonical Mode Input Processing** section for more details.

If XCASE is set, and if ICANON is set, an uppercase letter is accepted on input by preceding it with a \ character, and is output preceded by a \ character. In this mode, the following escape sequences are generated on output and accepted on input:

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

For example, A is input as \a, \n as \\n, and \N as \\\n. XCASE is currently ignored in this implementation. See LCASE in the *tty*(4) man page for this functionality.

If MDMBUF is set, output is controlled by carrier transitions. This is currently unimplemented.

If ALTWERASE is set, an alternate algorithm is used when erasing words with the WERASE character. The alternate algorithm treats words as adjacent alphanumeric and underscore characters separated by punctuation, control and white-space characters. When ALTWERASE is not set, white-space is used to delimit words.

If IEXTEN is set, the checking of input characters against the special characters DSUSP, ERASE2, EOL2, FLUSHO, LNEXT, QUOTE, REPRINT, and WERASE is enabled. When not set, these characters have no special meaning. Also enables ECHOCTL, ECHOKE, ECHOPRT, IMAXBEL, and IXANY (i.e. these flags have no effect unless IEXTEN is also set).

If ECHO is set, characters are echoed as received. If ECHO is not set, input characters are not echoed.

If ECHOCTL is not set, all control characters (characters with codes between 0 and 37 octal) are echoed as themselves. If ECHOCTL and IEXTEN are set, all control characters other than ASCII TAB, ASCII NL, the START character, and the STOP character are echoed as ^*X*, where *X* is the character given by adding 100 octal to the control character's code (so that the character with octal code 1 is echoed as '^A'), and the ASCII DEL character, with code 177 octal, is echoed as '^?'.

When ICANON is set, the following echo functions are possible:

1. If ECHO and ECHOE are set, and ECHOPRT is not set, the ERASE and WERASE characters are echoed as one or more ASCII BS SP BS, which will clear the last character(s) from a CRT screen.

2. If ECHO and ECHOPRT are set, the first ERASE and WERASE character in a sequence echoes as a backslash (\) followed by the characters being erased. Subsequent ERASE and WERASE characters echo the characters being erased, in reverse order. The next non-erase character types a slash (/) before it is echoed. ECHOPRT only has an effect when set if IEXTEN is also set.

3. If ECHOKE is set, the kill character is echoed by erasing each character on the line from the screen (using the mechanism selected by ECHOE and ECHOPRT). Only has an effect when set if IEXTEN is also set.

4. If ECHOK is set, and ECHOKE is not set, the NL character will be echoed after the kill character to emphasize that the line will be deleted. Note: an escape character (\) or an LNEXT character preceding the erase or kill character removes any special function.

5. If ECHONL is set, the NL character will be echoed even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex).

If NOFLSH is set, the normal flush of the input and output queues associated with the INTR, QUIT, and SUSP characters will not be done.

If TOSTOP is set, the signal SIGTTOU is sent to a process that tries to write to its controlling terminal if it is not in the distinguished process group for that terminal. This signal normally stops the process. Otherwise, the output generated by that process is delivered to the current output stream. Processes that are blocking or ignoring SIGTTOU signals are excepted and allowed to produce output.

If FLUSHO is set, data written to the terminal will be discarded. This bit is set when the FLUSH character is typed. A program can cancel the effect of typing the FLUSH character by clearing FLUSHO.

If PENDIN is set, any input that has not yet been read will be reprinted when the next character arrives as input.

The initial line-discipline control value is ECHO, ECHOK, ECHOE, ECHOCTL, ICANON, ISIG, IEXTEN.

Minimum and Timeout

The MIN and TIME values are described above under **Non-canonical Mode Input Processing**. The initial value of MIN is 1, and the initial value of TIME is 0.

IOCTLS

The ioctl calls supported by terminal devices providing the **termios** interface are listed below. Some calls may not be supported by all devices.

Unless otherwise noted for a specific ioctl call, these functions are restricted from use by background processes. Attempts to perform these calls will cause the process group of the process performing the call to be sent a SIGTTOU signal. If the process is ignoring SIGTTOU, has SIGTTOU blocked, or is in the middle of process creation using *vfork*(2), the process will be allowed to perform the call and the SIGTTOU signal will not be sent.

TIOCGETA The argument is a pointer to a **termios** structure. The current terminal parameters are fetched, and those parameters that can be stored in a **termios** structure are stored into that structure. This call is allowed from a background process; however, the information may subsequently be changed by a foreground process.

TIOCSETA The argument is a pointer to a **termios** structure. Those terminal parameters that can be stored in a **termios** structure are set from the values stored in that structure. The change is immediate.

If SETTFLAG is set in the **c_lflag** field of the **termios** argument, the terminal flags are set from the **c_tflag** field of the **termios** structure. This can be used instead of the TIOCSETP and TIOCLSET ioctls to set LCASE, TILDE and delay attributes of the terminal.

If SETWINSIZE is set in the **c_lflag** field of the **termios** argument, the terminal window size parameters are set from the **c_winsize** field of the **termios** structure. This can be used instead of the TIOCSWINSZ ioctl. As with the TIOCSWINSZ ioctl, if the new terminal size information is different than the old information, a SIGWINCH signal is sent to the foreground process group of

the terminal.

TIOCSETAW The argument is a pointer to a **termios** structure. Those terminal parameters that can be stored in a **termios** structure are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted. This form should be used when changing parameters that will affect output.

The SETTFLAG and SETWINSIZE flags in **c_lflag** of the **termios** structure are treated the same as in the TIOCSETA call.

TIOCSETAF The argument is a pointer to a **termios** structure. Those terminal parameters that can be stored in a **termios** structure are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted; all characters queued for input are discarded and then the change occurs.

The SETTFLAG and SETWINSIZE flags in **c_lflag** of the **termios** structure are treated the same as in the TIOCSETA call.

SEE ALSO

csh(1), stty(1), fork(2), ioctl(2), open(2), read(2), setpgrp(2), sigvec(2), vfork(2), tty(4), getty(8)

NAME

tty – general terminal interface

SYNOPSIS

#include <sgtty.h>
#include <sys/ttydev.h>

DESCRIPTION

This section describes both a particular special file **/dev/tty** and the terminal drivers used for conversational computing. The interface described here is supported through the use of the *termios* interface described in *termios*(4). See that man page for more information.

Line disciplines.

Historically, the system has provided different *line disciplines* for controlling communications lines. In this version of the system there is only one discipline but the behavior of "old" and "new" is simulated using this discipline. The effect of switching between line disciplines now is simply is a reinitialization of the special terminal control characters to a new set of defaults. The defaults for the "new" discipline enable the WERASE, REPRINT, SUSP, DSUSP, LNEXT, and FLUSHO characters while switching from "new" to "old" causes these characters to be disabled and thus have no special meaning. Consult *termios*(4) for the behavior of these control characters when enabled.

The behaviors of "new" and "old" are intended to mimic these definitions:

old — The old (standard) terminal driver. This is used when using the standard shell *sh*(1) and for compatibility with other standard version 7 UNIX systems. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

new — A newer terminal driver, with features for job control; this must be used when using *csh*(1) or *window*(1).

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

```
int ldisc = LDISC;
ioctl(fildes, TIOCSETD, &ldisc);
```

where LDISC is OTTYDISC for the standard tty driver, and NTTYDISC for the new driver. Although there is only one discipline, the values of OTTYDISC and NTTYDISC are maintained with the effect of switching between them as noted above, i.e. enabling or disabling certain control characters. The standard (currently old) tty driver is discipline 0 by convention. The current line discipline can be obtained with the TIOCGETD *ioctl*. Pending input is discarded when the line discipline is changed.

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

The control terminal.

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

A process which has no control terminal acquires one by first becoming a session leader via *setsid*(2) and then opening a terminal file which is not a control terminal for any other process in the system. That terminal becomes the control terminal for the session. The control terminal is thereafter inherited by a child process during a *fork*(2), even if the control terminal is closed.

A terminal may be the control terminal for only one session in the system at a time. A process may open a terminal file and guarantee not to acquire that terminal as its control terminal (even if the process is a session leader) by opening the terminal file with the O_NOCTTY flag (from fcntl.h).

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

A process can remove the association it has with its controlling terminal by opening the file **/dev/tty** and issuing a

ioctl(fildes, TIOCNOTTY, 0);

or by issuing this call on any *fildes* associated with the process's controlling terminal. This is often desirable in server processes.

Process groups.

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

ioctl(fildes, TIOCSPGRP, &pgrp);

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

Modes.

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

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

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

RAW This mode eliminates all input processing and makes all input characters available as they are typed; no output processing is done either.

The style of input processing can also be very different when the terminal is put in non-blocking i/o mode; see the FNDELAY flag as described in *fcntl*(2). In this case a *read*(2) from the control terminal will never block, but rather return an error indication (EWOULDBLOCK) if there is no input available.

A process may also request a SIGIO signal be sent it whenever input is present. To enable this mode the FASYNC flag should be set using *fcntl*(2).

Input editing.

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

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

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

Input characters are normally echoed by putting them in an output queue as they arrive. This may be disabled by clearing the ECHO bit in the flags word using the *stty*(3C) call or the TIOCSETN or TIOCSETP *ioctl*s (see the **Summary** below).

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

During input, line editing is normally done, with the erase character (normally DELETE or ^H) logically erasing the last character typed and the line kill character (normally ^U) logically erasing the entire current input line. These characters never erase beyond the beginning of the current input line or an ^D. These characters may be entered literally by preceding them with '\'; the '\' will normally be erased when the character is typed.

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

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

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

Input echoing and redisplay

The terminal driver has several modes (not present in standard UNIX Version 7 systems) for handling the echoing of terminal input, controlled by bits in a local mode word.

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

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

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

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

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

Output processing.

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

The terminal drivers provide necessary processing for cooked and CBREAK mode output including parity generation. Unlike some versions of UNIX, delays are *not* available after backspaces ^H, form feeds ^L, carriage returns ^M, tabs ^I and newlines ^J, due to difficulty in implementation. The driver will also optionally expand tabs into spaces, where the tab stops are assumed to be set every eight columns, and optionally convert newlines to carriage returns followed by newline. These functions are controlled by bits in the tty flags word; see **Summary** below.

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

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

Upper case terminals and Hazeltines

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

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

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

Flow control.

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

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

Line control and breaks.

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

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

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

Interrupt characters.

There are several characters that generate interrupts in cooked and CBREAK mode; all are sent to the processes in the control group of the terminal, as if a TIOCGPGRP *ioctl* were done to get the process group and then a *killpg*(2) system call were done, except that these characters also flush pending input and output when typed at a terminal (*à ``la* TIOCFLUSH). The characters shown here are the defaults; the field names in the structures (given below) are also shown. The characters may be changed.

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

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

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

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

Job access control.

When using the new terminal driver, if a process which is not in the distinguished process group of its control terminal attempts to read from that terminal its process group is sent a SIGTTIN signal. This signal normally causes the members of that process group to stop. If, however, the process is ignoring SIGTTIN, has SIGTTIN blocked, or is in the middle of process creation using *vfork*(2)), the read will return –1 and set *errno* to EIO.

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

Terminal/window sizes.

In order to accommodate terminals and workstations with variable-sized windows, the terminal driver provides a mechanism for obtaining and setting the current terminal size. The driver does

not use this information internally, but only stores it and provides a uniform access mechanism. When the size is changed, a SIGWINCH signal is sent to the terminal's process group so that knowledgeable programs may detect size changes.

Summary of modes.

Unfortunately, due to the evolution of the terminal driver, there are 4 different structures which contain various portions of the driver data. The first of these (**sgttyb**) contains that part of the information largely common between version 6 and version 7 UNIX systems. The second contains additional control characters added in version 7. The third is a word of local state added in 4BSD, and the fourth is another structure of special characters added for the new driver. In the future a single structure may be made available to programs which need to access all this information; most programs need not concern themselves with all this state.

Basic modes: sgtty.

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

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

The *sg_ispeed* and *sg_ospeed* fields describe the input and output speeds of the device according to the following table, which, for historical reasons, corresponds to the DEC DH-11 interface. Requests to change baud rates to unsupported baud rates, including split baud rates on current hardware, are ignored. Symbolic values in the table are as defined in *<sys/ttydev.h>*.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud (not available on current hardware)
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	19200 baud
EXTB	15	External B

Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

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

The *sg_flags* field of the argument structure contains several bits that determine the system's treatment of the terminal. Delay bits are defined, but not supported.

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

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

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

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

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

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

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

Input characters with the wrong parity, as determined by bits 200 and 100, are ignored in cooked and CBREAK mode.

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

CRMOD causes input carriage returns to be turned into new-lines, and output and echoed new-lines to be output as a carriage return followed by a line feed.

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line; all processing is done except the input editing: character and word erase and line kill, input reprint, and the special treatment of \ and EOT are disabled.

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

Note: The same "stop" and "start" characters are used for both directions of flow control; the *t_stopc* character is accepted on input as the character that stops output and is produced on output as the character to stop input, and the *t_startc* character is accepted on input as the character that restarts output and is produced on output as the character to restart input.

Basic ioctls

A large number of *ioctl*(2) calls apply to terminals. Some have the general form:

#include <sgtty.h>

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

The applicable codes are listed below. Some of these *ioctl*(2) calls cause the calling process to stop with a SIGTTOU signal if that process is not in the foreground process group of the terminal. These *ioctl*(2) calls are marked with a (*).

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

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

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

With the following codes *arg* is ignored.

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

TIOCNXCL Turn off "exclusive-use" mode.

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

TIOCRESET Request the driver to do a hard reset of the controller to which the specified terminal is attached. After waiting for the controller's self test (if any) to complete, attempt to restore sanity to the state of each port. This ioctl is intended for use only when the controller has gotten into a sufficiently confused state that a reboot would otherwise be required. It is valid only for superusers, and will return EPERM if attempted by others. While the driver attempts to preserve as much state as it can, hardware limitations may cause terminals attached to modems and port selectors to be disconnected by the reset operation. Terminals which are not disconnected may lose a small amount of input or output.

With the following codes *arg* is a pointer to an **int**.

TIOCGETD *arg* is a pointer to an **int** into which is placed the current line discipline number.

TIOCSETD(*) *arg* is a pointer to an **int** whose value becomes the current line discipline

number.

TIOCFLUSH(*) If the **int** pointed to by *arg* has a zero value, all characters waiting in input or output queues are flushed. Otherwise, the value of the **int** is treated as the logical OR of the FREAD and FWRITE defined in **<sys/file.h>**; if the FREAD bit is set, all characters waiting in input queues are flushed, and if the FWRITE bit is set, all characters waiting in output queues are flushed.

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

TIOCSTI(*) the argument points to a character which the system pretends had been typed on the terminal.

TIOCSBRK the break bit is set in the terminal.

TIOCCBRK the break bit is cleared.

TIOCSDTR data terminal ready is set.

TIOCCDTR data terminal ready is cleared.

TIOCSTOP output is stopped as if the "stop" character had been typed. See TIOCXONC.

TIOCSTART output is restarted as if the "start" character had been typed. See TIOCXONC.

TIOCXONC start and stop control for input and output. *arg* is a pointer to an **int** with the following 4 legal values and associated actions. If the value is 0, suspend output (identical to TIOCSTOP); if 1, restart suspended output (identical to TIOCSTART); if 2, suspend input; if 3, restart suspended input.

TIOCGPGRP *arg* is a pointer to an **int** into which is placed the process group ID of the process group for which this terminal is the control terminal.

TIOCSPGRP *arg* is a pointer to an **int** (typically a process ID); the process group whose process group ID is the value of this **int** becomes the process group for which this terminal is the control terminal. This process group ID must match a process in the same session as the calling process. *fildes* must be a file descriptor associated with the control terminal for the calling process's session.

TIOCSPGRP2 is identical to TIOCSPGRP with the exception that it may result in a SIGTTOU signal for the calling process if the terminal process group does not match the caller's; i.e., this may only be used to "give away" ownership of the terminal.

TIOCOUTQ returns in the **int** pointed to by *arg* the number of characters queued up to be output to the terminal.

TIOCDRAIN blocks the calling process until all characters in the output queue for the terminal have drained. This function differs from TIOCFLUSH in that queued characters are written to the terminal using TIOCDRAIN, but are discared using TIOCFLUSH.

FIONREAD returns in the **int** pointed to by *arg* the number of immediately readable characters from the argument unit. This works for files, pipes, and terminals.

Tchars

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

```
struct tchars {
        char    t_intrc;        /* interrupt */
        char    t_quitc;        /* quit */
```

```
        char    t_startc;       /* start output */
        char    t_stopc;        /* stop output */
        char    t_eofc;         /* end-of-file */
        char    t_brkc;         /* input delimiter (like nl) */
};
```

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

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

TIOCSETC(*)
Set the special characters to those given in the structure.

Local mode

The third structure associated with each terminal is a local mode word. The bits of the local mode word are:

LCRTBS	000001	Backspace on erase rather than echoing erase
LPRTERA	000002	Printing terminal erase mode
LCRTERA	000004	Erase character echoes as backspace-space-backspace
LTILDE	000010	Convert ~ to ` on output (for Hazeltine terminals)
LMDMBUF	000020	Stop/start output when carrier drops
LLITOUT	000040	Suppress output translations
LTOSTOP	000100	Send SIGTTOU for background output
LFLUSHO	000200	Output is being flushed
LNOHANG	000400	Don't send hangup when carrier drops
LETXACK	001000	Diablo style buffer hacking (unimplemented)
LCRTKIL	002000	BS-space-BS erase entire line on line kill
LCTLECH	010000	Echo input control chars as ^X, delete as ^?
LPENDIN	020000	Retype pending input at next read or input character
LDECCTQ	040000	Only ^Q restarts output after ^S, like DEC systems
LNOFLSH	100000	Inhibit flushing of pending I/O when an interrupt character is typed.

The applicable *ioctl* functions are:

TIOCLBIS(*) *arg* is a pointer to an **int** whose value is a mask containing the bits to be set in the local mode word.

TIOCLBIC(*) *arg* is a pointer to an **int** whose value is a mask containing the bits to be cleared in the local mode word.

TIOCLSET(*) *arg* is a pointer to an **int** whose value is stored in the local mode word.

TIOCLGET *arg* is a pointer to an **int** into which the current local mode word is placed.

Local special chars

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

```
struct ltchars {
        char    t_suspc;        /* stop process signal */
        char    t_dsuspc;       /* delayed stop process signal */
        char    t_rprntc;       /* reprint line */
        char    t_flushc;       /* flush output (toggles) */
```

```
        char    t_werasc;       /* word erase */
        char    t_lnextc;       /* literal next character */
};
```

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

The applicable *ioctl* functions are:

TIOCSLTC(*)
: *arg* is a pointer to an *ltchars* structure which defines the new local special characters.

TIOCGLTC
: *arg* is a pointer to an *ltchars* structure into which is placed the current set of local special characters.

Window/terminal sizes

Each terminal has provision for storage of the current terminal or window size in a *winsize* structure, with format:

```
struct winsize {
        unsigned short  ws_row;     /* rows, in characters */
        unsigned short  ws_col;     /* columns, in characters */
        unsigned short  ws_xpixel;  /* horizontal size, pixels */
        unsigned short  ws_ypixel;  /* vertical size, pixels */
};
```

A value of 0 in any field is interpreted as "undefined;" the entire structure is zeroed on final close.

The applicable *ioctl* functions are:

TIOCGWINSZ
: *arg* is a pointer to a **struct winsize** into which will be placed the current terminal or window size information.

TIOCSWINSZ(*)
: *arg* is a pointer to a **struct winsize** which will be used to set the current terminal or window size information. If the new information is different than the old information, a SIGWINCH signal will be sent to the terminal's process group.

Termios

User accessible terminal characteristics for the termios interface are stored in a *struct termios* defined in *<sys/termios.h>*, which is automatically included in *<termios.h>*. More information is available in *termios*(4).

```
struct termios {
        tcflag_t  c_iflag;      /* input flags */
        tcflag_t  c_oflag;      /* output flags */
        tcflag_t  c_cflag;      /* control flags */
        tcflag_t  c_lflag;      /* local flags */
        tcflag_t  c_tflag;      /* tty flags and state */
        speed_t   c_ispeed;     /* input speed */
        speed_t   c_ospeed;     /* output speed */
        struct
#ifndef _POSIX_SOURCE
          winsize
#endif
          {
```

```
        unsigned short ws_row;          /* rows, in characters */
        unsigned short ws_col;          /* columns, in characters */
        unsigned short ws_xpixel;       /* horizontal size, pixels */
        unsigned short ws_ypixel;       /* vertical size, pixels */
    }   c_winsize;                      /* window size */
    char    c_unused[32];               /* spares */
    cc_t    c_cc[NCCS];                 /* control chars */
};
```

The following *ioctl* calls expect *arg* to be a pointer to a *struct termios :*

TIOCGETA
: Get the current terminal attributes and place them in the structure specified by *arg* . This call is allowed from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

TIOCSETA(*)
: Set the current terminal attributes to the value of the *struct termios* pointed to by *arg.* The change occurs immediately.

TIOCSETAW(*)
: Set the terminal attributes after output has drained. This call is similar to **TIOCSETA** except that the change occurs after all output queued to *fildes* has been transmitted. This call should be used when changing parameters that affect output.

TIOCSETAF(*)
: Set the terminal attributes after output has drained and flush any input. This call is similar to **TIOCSETA** except that the change occurs after all output queued to *fildes* has been transmitted, and al input that has been received but not read are discarded before the change is made.

FILES

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

SEE ALSO

csh(1), stty(1), tset(1), intro(2), ioctl(2), sigvec(2), stty(3C), termios(4), getty(8), init(8)

BUGS

Half-duplex terminals are not supported.

NAME

udp – Internet User Datagram Protocol

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_DGRAM, 0);

DESCRIPTION

UDP is a simple, unreliable datagram protocol which is used to support the SOCK_DGRAM abstraction for the Internet protocol family. UDP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls, though the *connect*(2) call may also be used to fix the destination for future packets (in which case the *recv*(2) or *read*(2) and *send*(2) or *write(2)* system calls may be used).

UDP address formats are identical to those used by TCP. In particular UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space (i.e. a UDP port may not be "connected" to a TCP port). In addition broadcast packets may be sent (assuming the underlying network supports this) by using a reserved "broadcast address"; this address is network interface dependent.

Options at the IP transport level may be used with UDP; see *ip*(4P).

For each socket (UDP or otherwise), the system maintains values indicating the amount of space allocated for buffering reads and for buffering writes. When a socket is created, these values are assigned system default values (these default values are subject to change). Values for a particular socket may manipulated using the SO_SNDBUF and SO_RCVBUF options with the *setsockopt*(2) and *getsockopt*(2) system calls. If necessary, a write to a UDP socket is automatically fragmented into shorter packets and reassembled on the receiving end to give the appearance of sending and receiving a single datagram.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;

[ENOBUFS] when the system runs out of memory for an internal data structure;

[EADDRINUSE] when an attempt is made to create a socket with a port which has already been allocated;

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists.

SEE ALSO

getsockopt(2), recv(2), send(2), socket(2), intro(4N), inet(4F), ip(4P)

NOTES

Udp is an optional product; for more information, contact your CONVEX sales representative.

Section 5
File formats

NAME

L-devices – UUCP device description file

DESCRIPTION

The L-devices file is consulted by the UUCP daemon *uucico*(8C) under the direction of *L.sys*(5) for information on the devices that it may use. Each line describes exactly one device.

A line in *L-devices* has the form:

Caller Device Call_Unit Class Dialer [Expect Send]....

Each item can be separated by any number of blanks or tabs. Lines beginning with a '#' character are comments; long lines can be continued by appending a '\' character to the end of the line.

Caller denotes the type of connection, and must be one of the following:

ACU Automatic call unit, e.g., autodialing modems such as the Hayes Smartmodem 1200 or Racal-Vadic "VA-212".

DIR Direct connect; hardwired line (usually RS-232) to a remote system.

PAD X.25 PAD connection over CONVEX OSI. Note that listing OSI connections in *L-devices* is superfluous; *uucico* does not even bother to look here since it has all the information it needs in *L.sys*(5).

PCP GTE Telenet PC Pursuit.

TCP Berkeley TCP/IP or 3Com UNET connection. These are mutually exclusive. Note that listing TCP connections in *L-devices* is superfluous; *uucico* does not even bother to look here since it has all the information it needs in *L.sys*(5).

X25 X.25 PAD connection using a hardware PAD.

Device is a device file in /dev/ that is opened to use the device. The device file must be owned by UUCP, with access modes of 0600 or better. (See *chmod*(2)).

Call_Unit is an optional second device file name. True automatic call units use a separate device file for data and for dialing; the *Device* field specifies the data port, while the *Call_unit* field specifies the dialing port. If the *Call_unit* field is unused, it must not be left empty. Insert a dummy entry as a placeholder, such as "0" or "unused."

Class is an integer number that specifies the line baud (for dialers and direct lines) or the port number (for network connections).

The *Class* may be preceded by a non-numeric prefix. This is to differentiate among devices that have identical *Caller* and baud, but are distinctly different. For example, "1200" could refer to all Bell 212-compatible modems, "V1200" to Racal-Vadic modems, and "H1200" to Hayes modems, all at 1200 baud. Similarly, "W1200" could denote long distance lines, while "L1200" could refer to local phone lines. On the other hand, "0" is used for the TrailBlazer 9600 baud modem.

Dialer applies only to ACU devices. This is the "brand" or type of the ACU or modem.

hayes Hayes Smartmodem 1200 and compatible autodialing modems. It is permissible to include the letters 'T' and 'P' in the phone number (in *L.sys*) to change to tone or pulse midway through dialing. (Note that a leading 'T' or 'P' will be interpreted as a dial-code!)

hayes2400
Hayes Smartmodem 2400 and compatible modems.

va212 Racal-Vadic 212 autodialing modem.

vadic Racal-Vadic 3450 and 3451 series autodialing modems.

trailblazer

Telbit TrialBlazer 9600 baud modem. Note: the class (see above) should be a 0.

Expect/Send is an optional *Expect/Send* script for getting through a smart port selector, or for issuing special commands to the modem. The syntax is identical to that of the Expect/Send script of *L.sys*. The difference is that the *L-devices* script is used *before* the connection is made, while the *L.sys* script is used *after*.

FILES

/usr/lib/uucp/L-devices
/usr/lib/conf/uucp/L-devices L-devices example

SEE ALSO

uucp(1C), uux(1C), L.sys(5), uucico(8C)

NAME

L-dialcodes – UUCP phone number index file

DESCRIPTION

The *L-dialcodes* file defines the mapping of strings from the phone number field of *L.sys*(5) to actual phone numbers.

Each line in L-dialcodes has the form:

alpha_string phone_number

The two items can be separated by any number of blanks or tabs. Lines beginning with a '#' character are comments.

A phone number in *L.sys* can be preceded by an arbitrary alphabetic character string; the string is matched against the list of *alpha_strings* in *L-dialcodes*. If a match is found, *phone_number* is substituted for it. If no match is found, the string is discarded.

L-dialcodes is commonly used either of two ways:

(1) The alphabetic strings are used as prefixes to denote area codes, zones, and other commonly used sequences. For example, if *L-dialcodes* included the following lines:

```
out     9,
outld   8,1
```

In *L.sys* you could enter:

```
foovax  Any ACU 1200 out5551234  ogin:--ogin: nuucp
barvax  Any ACU 1200 outld5556001   ogin:--ogin: Uuucp
```

instead of

```
foovax  Any ACU 1200 out5551234  ogin:--ogin: nuucp
barvax  Any ACU 1200 outld4155556001  ogin:--ogin: Uuucp
```

(2) All phone numbers are placed in *L-dialcodes*, one for each remote site. *L.sys* then refers to these by name. For example, if *L-dialcodes* contains the following lines:

```
bazvax  13125551234
amnesia14155556001
```

then *L.sys* could have:

```
bazvax Any ACU 1200 bazvax  ogin:--ogin: nuucp
amnesia  Any ACU 1200 amnesia   ogin:--ogin: Uuucp
```

This scheme allows a site administrator to give users read access to the table of phone numbers, while still protecting the login/password sequences in *L.sys*.

SEE ALSO

uucp(1C), uux(1C), L.sys(5), uucico(8C).

NAME

L.aliases – UUCP hostname alias file

DESCRIPTION

The *L.aliases* file defines mapping (aliasing) of system names for uucp. This is intended for compensating for systems that have changed names, or do not provide their entire machine name (like most USG systems). It is also useful when a machine's name is not obvious or commonly misspelled.

Each line in *L.aliases* is of the form:

real_name alias_name

Any amount of whitespace may separate the two items. Lines beginning with a '#' character are comments.

All occurrences of *alias_name* are mapped to *real_name* by *uucico*(8C), *uucp*(1), and *uux*(1). The mapping occurs regardless of whether the name was typed in by a user or provided by a remote site. An exception is the **-s** option of *uucico*; only the site's real hostname (the name in *L.sys*(5)) will be accepted there.

Aliased system names should not be placed in *L.sys*; they will not be used.

FILES

/usr/lib/uucp/L.aliases

SEE ALSO

uucp(1C), uux(1C), L.sys(5), uucico(8C)

NAME

L.cmds – UUCP remote command permissions file

DESCRIPTION

The *L.cmds* file contains a list of commands, one per line, that are permitted for remote execution via *uux*(1C).

The default search path is /bin:/usr/bin:/usr/ucb. To change the path, include anywhere in the file a line of the form:

PATH=/bin:/usr/bin:/usr/ucb

Normally, an acknowledgment is mailed back to the requesting site after the command completes. If a command name is suffixed with **,Error**, then an acknowledgment will be mailed only if the command fails. If the command is suffixed with **,No**, then no acknowledgment will ever be sent. (These correspond with the **–z** and **–n** options of *uux*, respectively.)

For most sites, *L.cmds* should only include the lines:

rmail
ruusend
nfrcv
nfxmit
uncompress

While file names supplied as arguments to *uux* commands will be checked against the list of accessible directory trees in *USERFILE*(5), this check can be easily circumvented and should not be depended upon. In other words, it is unwise to include any commands in *L.cmds* that accept local file names. In particular, *sh*(1) and *csh*(1) are extreme risks.

It is common (but hazardous) to include *uucp*(1C) in *L.cmds*; see the NOTES section of *USERFILE*.

FILES

/usr/lib/uucp/L.cmds

SEE ALSO

uucp(1C), uux(1C), USERFILE(5), uucico(8C), uuxqt(8C)

NAME

L.sys – UUCP remote host description file

DESCRIPTION

The *L.sys* file is consulted by the UUCP daemon *uucico*(8C) for information on remote systems. *L.sys* includes the system name, appropriate times to call, phone numbers, and a login and password for the remote system. *L.sys* is thus a privileged file, owned by the UUCP Administrator; it is accessible only to the Administrator and to the superuser.

Each line in *L.sys* describes one connection to one remote host, and has the form:

System Times Caller Class Device/Phone_Number [Expect Send]....

Fields can be separated by any number of blanks or tabs. Lines beginning with a '#' character are comments; long lines can be continued by appending a '\' character to the end of the line.

The first five fields (*System* through *Device/Phone_Number*) specify the hardware mechanism that is necessary to make a connection to a remote host, such as a modem or network. *Uucico* searches from the top down through *L.sys* to find the desired *System*; it then opens the *L-devices*(5) file and searches for the first available device with the same *Caller*, *Class*, and (possibly) *Device*. ("Available" means that the device is ready and not being used for something else.) *Uucico* attempts a connection using that device; if the connection cannot be made (for example, a dialer gets a busy signal), *uucico* tries the next available device. If this also fails, it returns to *L.sys* to look for another line for the same *System*. If none is found, *uucico* gives up.

System is the hostname of the remote system. Every machine with which this system communicates via UUCP should be listed, regardless of who calls whom. Systems not listed in *L.sys* will not be permitted a connection. The local hostname should **not** appear here for security reasons.

Times is a comma-separated list of the times of the day and week that calls are permitted to this *System*. *Times* is most commonly used to restrict long distance telephone calls to those times when rates are lower. List items are constructed as:

*keyword*hhmm-hhmm/*grade*;*retry_time*

Keyword is required, and must be one of:

Any Any time, any day of the week.

Wk Any weekday. In addition, **Mo**, **Tu**, **We**, **Th**, **Fr**, **Sa**, and **Su** can be used for Monday through Sunday, respectively.

Evening
When evening telephone rates are in effect, from 1700 to 0800 Monday through Friday, and all day Saturday and Sunday. **Evening** is the same as **Wk1700-0800,Sa,Su**.

Night When nighttime telephone rates are in effect, from 2300 to 0800 Monday through Friday, all day Saturday, and from 2300 to 1700 Sunday. **Night** is the same as **Any2300-0800,Sa,Su0800-1700**.

NonPeak
This is a slight modification of **Evening**. It matches when the USA X.25 carriers have their lower rate period. This is 1800 to 0700 Monday through Friday, and all day Saturday and Sunday. **NonPeak** is the same as **Any1800-0700,Sa,Su**.

Never Never call; calling into this *System* is forbidden or impossible. This is intended for polled connections, where the remote system calls into the local machine periodically. This is necessary when one of the machines is lacking either dial-in or dial-out modems.

The optional *hhmm-hhmm* subfield provides a time range that modifies the keyword. *hhmm* refers to *hours* and *minutes* in 24-hour time (from 0000 to 2359). The time range is permitted to "wrap" around midnight, and will behave in the obvious way. It is invalid to follow the **Evening**, **NonPeak**, and **Night** keywords with a time range.

The *grade* subfield is optional; if present, it is composed of a '/' (slash) and single character denoting the *grade* of the connection, from **0** to **9**, **A** to **Z**, or **a** to **z**. This specifies that only requests of grade *grade* or better will be transferred during this time. (The grade of a request or job is specified when it is queued by *uucp* or *uux*.) By convention, mail is sent at grade **C**, news is sent at grade **d**, and uucp copies are sent at grade **n**. Unfortunately, some sites do not follow these conventions, so it is not 100% reliable.

The *retry_time* subfield is optional; it must be preceded by a ';' (semicolon) and specifies the time, in minutes, before a failed connection may be tried again. (This restriction is in addition to any constraints imposed by the rest of the *Time* field.) By default, the retry time starts at 10 minutes and gradually increases at each failure, until after 26 tries *uucico* gives up completely (MAX RETRIES). If the retry time is too small, *uucico* may run into MAX RETRIES too soon.

Caller is the type of device used:

ACU Automatic call unit or auto-dialing modem such as the Hayes Smartmodem 1200 or Racal-Vadic. See *L-devices* for a list of supported modems.

DIR Direct connect; hardwired line (usually RS-232) to a remote system.

PAD X.25 PAD connection using CONVEX OSI. OSI connections do **not** need entries in *L-devices* since all the necessary information is contained in *L.sys*.

PCP GTE Telenet PC Pursuit. See *L-devices* for configuration details.

TCP Berkeley TCP/IP or 3Com UNET connection. These are mutually exclusive. TCP ports do **not** need entries in *L-devices* since all the necessary information is contained in *L.sys*. If several alternate ports or network connections should be tried, use multiple *L.sys* entries.

X25 X.25 PAD connection using a hardware PAD.

Class is usually the speed (baud) of the device, typically 300, 1200, or 2400 for ACU devices and 9600 for direct lines. Valid values are device dependent, and are specified in the *L-devices* file.

On some devices, the baud may be preceded by a non-numeric prefix. This is used in *L-devices* to distinguish among devices that have identical *Caller* and baud, but yet are distinctly different. For example, 1200 could refer to all Bell 212-compatible modems, V1200 to Racal-Vadic modems, and H1200 to Hayes modems, all at 1200 baud.

On TCP connections, *Class* is the port number (an integer number) or a port name from */etc/services* that is used to make the connection. For standard Berkeley TCP/IP, UUCP normally uses port number 540.

For CONVEX OSI connections, *Class* is ignored.

Device/Phone_Number varies based on the *Caller* field. For ACU devices, this is the phone number to dial. The number may include: digits **0** through **9**; **#** and ***** for dialing those symbols on tone telephone lines; **-** (hyphen) to pause for a moment, typically two to four seconds; **=** (equal sign) to wait for a second dial tone (implemented as a pause on many modems). Other characters are modem dependent; generally standard telephone punctuation characters (such as the slash and parentheses) are ignored, although *uucico* does not guarantee this.

The phone number can be preceded by an alphabetic string; the string is indexed and converted through the *L-dialcodes*(5) file.

For DIR devices, the *Device/Phone_Number* field contains the name of the device in */dev* that is used to make the connection. There must be a corresponding line in *L-devices* with identical *Caller*, *Class*, and *Device* fields.

For TCP and other network devices, *Device/Phone_Number* holds the true network name of the remote system, which may be different from its UUCP name (although one would hope not).

Expect and *Send* refer to an arbitrarily long set of strings that alternately specify what to *expect* and what to *send* to login to the remote system once a physical connection has been established. A complete set of expect/send strings is referred to as an *expect/send script*. The same syntax is used in the *L-devices* file to interact with the dialer prior to making a connection; there it is referred to as a *chat script*. The complete format for one *expect/send* pair is:

expect-timeout-send-expect-timeout send

Expect and *Send* are character strings. *Expect* is compared against incoming text from the remote host; *send* is sent back when *expect* is matched. By default, the *send* is followed by a '\r' (carriage return). If the *expect* string is not matched within *timeout* seconds (default 45), then it is assumed that the match failed. The '*expect-send-expect*' notation provides a limited loop mechanism; if the first *expect* string fails to match, then the *send* string between the hyphens is transmitted, and *uucico* waits for the second *expect* string. This can be repeated indefinitely. When the last *expect* string fails, *uucico* hangs up and logs that the connection failed.

The timeout can (optionally) be specified by appending the parameter '~*nn*' to the expect string, when *nn* is the timeout time in seconds.

Backslash escapes that may be embedded in the *expect* or *send* strings include:

\b	Generate a 3/10 second BREAK.
\b*n*	Where *n* is a single-digit number; generate an *n*/10 second BREAK.
\c	Suppress the \r at the end of a *send* string.
\d	Delay; pause for 1 second. (*Send* only.)
\r	Carriage Return.
\s	Space.
\n	Newline.
\xxx	Where *xxx* is an octal constant; denotes the corresponding ASCII character.

As a special case, an empty pair of double-quotes "" in the *expect* string is interpreted as "expect nothing"; that is, transmit the *send* string regardless of what is received. Empty double-quotes in the *send* string cause a lone '\r' (carriage return) to be sent.

One of the following keywords may be substituted for the *send* string:

BREAK	Generate a 3/10 second BREAK
BREAK*n*	Generate an *n*/10 second BREAK
CR	Send a Carriage Return (same as "").
EOT	Send an End-Of-Transmission character, ASCII \004. Note that this will cause most hosts to hang up.
NL	Send a Newline.
PAUSE	Pause for 3 seconds.
PAUSE*n*	Pause for *n* seconds.
P_ODD	Use odd parity on future send strings.
P_ONE	Use parity one on future send strings.
P_EVEN	Use even parity on future send strings. (Default)
P_ZERO	Use parity zero on future send strings.

Finally, if the *expect* string consists of the keyword **ABORT**, then the string following is used to arm an abort trap. If that string is subsequently received any time prior to the completion of the entire *expect/send* script, then *uucico* will abort, just as if the script had timed out. This is useful for trapping error messages from port selectors or front-end processors such as "Host Unavailable" or "System is Down."

For example:

"" "" ogin:--ogin: nuucp ssword: ufeedme

This is executed as, "When the remote system answers, *expect* nothing. *Send* a carriage return. *Expect* the remote to transmit the string 'ogin:'. If it doesn't within 45 seconds, send another carriage return. When it finally does, *send* it the string 'nuucp'. Then *expect* the string 'ssword:'; when that is received, *send* 'ufeedme'."

FILES

/usr/lib/uucp/L.sys

SEE ALSO

uucp(1C), uux(1C), L-devices(5), services(5), uucico(8C)

BUGS

"ABORT" in the send/expect script is expressed "backwards," that is, it should be written " *expect* **ABORT**" but instead it is " **ABORT** *expect*".

Several of the backslash escapes in the send/expect strings are confusing and/or different from those used by AT&T and Honey-Danber UUCP. For example, '\b' requests a BREAK, while practically everywhere else '\b' means backspace. '\t' for tab and '\f' for formfeed are not implemented. '\s' is a kludge; it would be more sensible to be able to delimit strings with quotation marks.

NAME

a.out – CONVEX assembler and link editor output

SYNOPSIS

```
#include <convex/filehdr.h>
#include <convex/opthdr.h>
#include <convex/scnhdr.h>
#include <nlist.h>
#include <convex/reloc.h>
```

DESCRIPTION

The file *a.out* is the output of the assembler *as*(1) and the link editor *ld*(1). The loader makes *a.out* executable if there were no errors and no unresolved external references.

Standard Object File Format

A standard format object file consists of a file header, an optional (secondary) header, a table of section headers, a table of section data, relocation information, a symbol table, and a string table. The order for a link edited executable is:

File header
Optional (secondary) file header
Section 1 header
...
Section *n* header
Section 1 raw data
...
Section *n* raw data
Section 1 relocation information
...
Section *n* relocation information
Symbol table
String table

The order for a link edited relocatable object is:

File header
Optional (secondary) file header
Section 1 header
...
Section *n* header
Section 1 raw data
Section 1 relocation information
...
Section *n* raw data
Section *n* relocation information
Symbol table
String table

The last three partitions (relocation information, symbol table and string table) may be missing if the program was linked with the **-s** option of *ld*(1) or if the information was removed by *strip*(1). Also note that if there are no unresolved external references after linking, the relocation information will be removed from *a.out* by the loader.

When an *a.out* file is executed, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), and

a stack. The text segment begins at *o_entry* in the core image; the headers are never loaded.

For standard format object files, the magic number (the first field of the file header) is always set to SOFF_MAGIC (0601).

The structure of the file header is:

```
struct filehdr {
  unsigned long       h_magic;     /* magic number */
  unsigned long       h_version;   /* version number */
  long                h_timdat;    /* time & date stamp */
  unsigned long       h_nscns;     /* number of sections */
  unsigned long long  h_scnptr;    /* file ptr to 1st scnhdr */
  unsigned long       h_opthdr;    /* sizeof(optional header) */
  unsigned long long  h_strptr;    /* file ptr to string tab */
  unsigned long long  h_strsiz;    /* # bytes in string tab */
  unsigned long long  h_flags;     /* flag word */
  long                h_spares[3]; /* room to grow */
};
```

The structure of the optional file header is:

```
struct opthdr {
  unsigned long long  o_symptr;    /* file ptr to symbol tab */
  unsigned long       o_nsyms;     /* # of entries in sym tab */
  long                o_spare;     /* reserved - must be 0 */
  unsigned long       o_entry;     /* entry point */
  unsigned long long  o_flags;     /* flag word */
};
```

File pointers are byte offsets into the file; the pointers may be cast and used as the offset in a call to *fseek*(3S).

Attributes of the object file are defined in the flags field of the optional file header.

The various meanings of the bits are:

OF_OBJECT
: If OF_OBJECT is set in the flags word, the file is assembler object output and is not executable.

OF_EXEC
: If OF_EXEC is set, the file is executable. There are three types of executables. If the type is OF_DEMAND, then the text segment is not writable by the program, and if other processes are executing the same file, they will share the text segment. For OF_DEMAND format, the size of the text and data section must both be multiples of 4096 bytes, and the pages will be brought into the running image as needed. This is the default format produced by *ld*(1). The OF_PREPAGED and OF_NON_SWAP formats are identical to the OF_DEMAND format. However, when an OF_PREPAGED file is executed, it is pre-paged rather than demand-paged. When an executable's type is OF_NON_SWAP, it will be pre-paged and not swapped when executed. Only the superuser can execute OF_NON_SWAP files. Note that pre-paging includes only the text and original data segments. Pages added by *sbrk(2)* and pages in BSS are *not* pre-paged.

OF_FP_MODE_NATIVE
: In addition to the type of the executable, each *a.out* executes in a particular floating-point mode. If the mode is OF_FP_MODE_NATIVE, then the CONVEX

representation of floating point numbers is used (this representation is referred to as *native* mode). If the mode bit is set to OF_FP_MODE_IEEE, then the IEEE standard representation of floating point numbers is used (*ieee* mode).

OF_STRIPPED
: If this bit is set, the file has been stripped.

Each standard format object file has a table of section headers to specify the layout of the data within the file. Each raw data section within an object file has its own header.

The structure of a section header is:

```
/*
 * s_vaddr and s_align are synonyms, s_align is used for alignment
 * data in object files, since they can't have virtual addresses,
 * and s_vaddr is used in executable images, since they already have
 * the alignment data figured in to the load address.
 */
#define s_align       s_vaddr    /* also used for alignment data */

struct scnhdr {
  unsigned long long s_strndx; /* section name index in string table */
  unsigned long      s_vaddr;  /* virtual load address of section  */
  unsigned long long s_size;   /* size of section in bytes */
  unsigned long long s_scnptr; /* offset in file to raw data */
  unsigned long long s_relptr; /* offset in file to relocation data */
  unsigned long      s_nrel;   /* number of relocation entries */
  unsigned long      s_prot;   /* protection flags (see pte.h) */
  unsigned long long s_flags;  /* flags word */
};
```

If a section is initialized, the file contains the actual bytes. An uninitialized section is somewhat different. It has a size, symbols defined in it, and symbols that refer to it, but it can have no relocation entries or data. Consequently, an uninitialized section has no raw data in the object file, and the values for *s_scnptr*, *s_relptr*, and *s_nrel* are zero.

s_size for an initialized common section includes a table of contents which precedes the raw data in the section. The table of contents is a map of the raw data. The first four bytes of the table gives the number of table entries. Each table entry is a tuple of the form (offset, length), where offset is the offset into the common data, and length is the size of the datum (e.g., 1 byte for a character datum). The table of contents is used by *ld*(1), to load multiply defined common sections.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file **<nlist.h>** as follows:

```
/* format of a symbol table entry */

struct nlist {
  union {
    char        *n_name;   /* for use when in core */
    long        n_strx;    /* index into string table */
  } n_un;                  /* union for name and string index */

  unsigned char  n_type;   /* type flag, see below */
  char           n_other;  /* unused */
```

NAME

a.out – CONVEX assembler and link editor output

SYNOPSIS

#include <convex/filehdr.h>
#include <convex/opthdr.h>
#include <convex/scnhdr.h>
#include <nlist.h>
#include <convex/reloc.h>

DESCRIPTION

The file *a.out* is the output of the assembler *as*(1) and the link editor *ld*(1). The loader makes *a.out* executable if there were no errors and no unresolved external references.

Standard Object File Format

A standard format object file consists of a file header, an optional (secondary) header, a table of section headers, a table of section data, relocation information, a symbol table, and a string table. The order for a link edited executable is:

File header
Optional (secondary) file header
Section 1 header
...
Section *n* header
Section 1 raw data
...
Section *n* raw data
Section 1 relocation information
...
Section *n* relocation information
Symbol table
String table

The order for a link edited relocatable object is:

File header
Optional (secondary) file header
Section 1 header
...
Section *n* header
Section 1 raw data
Section 1 relocation information
...
Section *n* raw data
Section *n* relocation information
Symbol table
String table

The last three partitions (relocation information, symbol table and string table) may be missing if the program was linked with the **-s** option of *ld*(1) or if the information was removed by *strip*(1). Also note that if there are no unresolved external references after linking, the relocation information will be removed from *a.out* by the loader.

When an *a.out* file is executed, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), and

a stack. The text segment begins at *o_entry* in the core image; the headers are never loaded.

For standard format object files, the magic number (the first field of the file header) is always set to SOFF_MAGIC (0601).

The structure of the file header is:

```
struct filehdr {
  unsigned long       h_magic;     /* magic number */
  unsigned long       h_version;   /* version number */
  long                h_timdat;    /* time & date stamp */
  unsigned long       h_nscns;     /* number of sections */
  unsigned long long  h_scnptr;    /* file ptr to 1st scnhdr */
  unsigned long       h_opthdr;    /* sizeof(optional header) */
  unsigned long long  h_strptr;    /* file ptr to string tab */
  unsigned long long  h_strsiz;    /* # bytes in string tab */
  unsigned long long  h_flags;     /* flag word */
  long                h_spares[3]; /* room to grow */
};
```

The structure of the optional file header is:

```
struct opthdr {
  unsigned long long  o_symptr;    /* file ptr to symbol tab */
  unsigned long       o_nsyms;     /* # of entries in sym tab */
  long                o_spare;     /* reserved - must be 0 */
  unsigned long       o_entry;     /* entry point */
  unsigned long long  o_flags;     /* flag word */
};
```

File pointers are byte offsets into the file; the pointers may be cast and used as the offset in a call to *fseek*(3S).

Attributes of the object file are defined in the flags field of the optional file header.

The various meanings of the bits are:

OF_OBJECT
: If OF_OBJECT is set in the flags word, the file is assembler object output and is not executable.

OF_EXEC
: If OF_EXEC is set, the file is executable. There are three types of executables. If the type is OF_DEMAND, then the text segment is not writable by the program, and if other processes are executing the same file, they will share the text segment. For OF_DEMAND format, the size of the text and data section must both be multiples of 4096 bytes, and the pages will be brought into the running image as needed. This is the default format produced by *ld*(1). The OF_PREPAGED and OF_NON_SWAP formats are identical to the OF_DEMAND format. However, when an OF_PREPAGED file is executed, it is pre-paged rather than demand-paged. When an executable's type is OF_NON_SWAP, it will be pre-paged and not swapped when executed. Only the superuser can execute OF_NON_SWAP files. Note that pre-paging includes only the text and original data segments. Pages added by *sbrk(2)* and pages in BSS are *not* pre-paged.

OF_FP_MODE_NATIVE
: In addition to the type of the executable, each *a.out* executes in a particular floating-point mode. If the mode is OF_FP_MODE_NATIVE, then the CONVEX

representation of floating point numbers is used (this representation is referred to as *native* mode). If the mode bit is set to OF_FP_MODE_IEEE, then the IEEE standard representation of floating point numbers is used (*ieee* mode).

OF_STRIPPED
If this bit is set, the file has been stripped.

Each standard format object file has a table of section headers to specify the layout of the data within the file. Each raw data section within an object file has its own header.

The structure of a section header is:

```
/*
 * s_vaddr and s_align are synonyms, s_align is used for alignment
 * data in object files, since they can't have virtual addresses,
 * and s_vaddr is used in executable images, since they already have
 * the alignment data figured in to the load address.
 */
#define s_align       s_vaddr    /* also used for alignment data */

struct scnhdr {
  unsigned long long s_strndx; /* section name index in string table */
  unsigned long      s_vaddr;  /* virtual load address of section  */
  unsigned long long s_size;   /* size of section in bytes */
  unsigned long long s_scnptr; /* offset in file to raw data */
  unsigned long long s_relptr; /* offset in file to relocation data */
  unsigned long      s_nrel;   /* number of relocation entries */
  unsigned long      s_prot;   /* protection flags (see pte.h) */
  unsigned long long s_flags;  /* flags word */
};
```

If a section is initialized, the file contains the actual bytes. An uninitialized section is somewhat different. It has a size, symbols defined in it, and symbols that refer to it, but it can have no relocation entries or data. Consequently, an uninitialized section has no raw data in the object file, and the values for *s_scnptr*, *s_relptr*, and *s_nrel* are zero.

s_size for an initialized common section includes a table of contents which precedes the raw data in the section. The table of contents is a map of the raw data. The first four bytes of the table gives the number of table entries. Each table entry is a tuple of the form (offset, length), where offset is the offset into the common data, and length is the size of the datum (e.g., 1 byte for a character datum). The table of contents is used by *ld*(1), to load multiply defined common sections.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file **<nlist.h>** as follows:

```
/* format of a symbol table entry */

struct nlist {
  union {
    char        *n_name;  /* for use when in core */
    long        n_strx;   /* index into string table */
  } n_un;                 /* union for name and string index */

  unsigned char  n_type;  /* type flag, see below */
  char           n_other; /* unused */
```

```
  short          n_desc;    /* see <stab.h> */
  unsigned long  n_value;   /* value of symbol (or sdb 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_TDATA    0xa    /* thread data */
#define N_TBSS     0xc    /* thread bss */
#define N_SCNHDR   0x10   /* extended symbol, fully defined */
                          /* by a section header */
#define N_COMM     0x12   /* un-initialized common */
#define N_FN       0x1f   /* file name symbol */

#define N_TCDATA   0x14   /* thread common initialized */
#define N_TCBSS    0x16   /* thread common uninitialized */

#define N_EXT      01     /* external bit, or'ed in */
#define N_TYPE     0x1e   /* mask for all the type bits */

/* csd entries have some of the N_STAB bits set. these are given
  in <stab.h> */

#define N_STAB     0xe0   /* any of these bits set -> csd entry */


/* format for namelist values */

#define N_FORMAT   "%08x"


/* FILE_MOD_FAIL is returned by libc routine nlist() to signify that */
/* the file being read has been changed in the time it took to do a  */
/* symbol name lookup.                                               */

#define FILE_MOD_FAIL    -2
```

In the *a.out* file, a symbol's *n_un.n_strx* field gives an index into the string table. An *n_strx* value of 0 indicates that no name is associated with a the 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 as the name of an uninitialized common block whose size is indicated by the value of the symbol. An extended symbol represents an initialized common block. In this case, the symbol will be marked as "external", its value will be the size of the common region, and its description field will be an index into the section header table. An extended symbol is fully defined by its section header.

The value of a word in a text, data, or common section that is not a reference to an undefined external symbol is exactly the value that will appear in memory when the file is executed. If a word involves a reference to an undefined external symbol, the storage class of the symbol-table entry for that word will be labeled "external"; when the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the word in the file.

If relocation information is present, it amounts to eight bytes per relocatable datum defined by the following C structure:

```
/*
 * Format of a relocation datum.
 */
struct relocation_info {
  int          r_address;       /* address which is relocated */
  unsigned int r_fill:4;        /* nothing */
  unsigned int r_extern:1;      /* does not include value of sym */
  unsigned int r_length:2;      /* 0=byte, 1=half, 2=word */
  unsigned int r_pcrel:1;       /* was relocated pc relative already */
  unsigned int r_symbolnum:24; /* local symbol ordinal */
};
```

There is no relocation information for a section if the *s_relptr* and *s_nrel* fields of the respective section header are zero. If *r_extern* is zero, then *r_symbolnum* is actually an *n_type* for the relocation entry (i.e. N_TEXT meaning relative to the text section origin.)

Nonstandard Object File Format

Nonstandard object file format was produced by the assembler and loader prior to Version 6.1 of the operating system. Nonstandard form is still accepted as input to the standard utilities but is no longer produced by *as*(1), or *ld*(1). Structures and macros for nonstandard format are defined in the include file **<a.out.h>**

A nonstandard object file has five segments: a header, the program text and data, relocation information, a symbol table, and a string table, in that order. The last three segments may be omitted if the program was loaded with the **-s** option of *ld*(1) or if the symbols and relocation have been removed by *strip*(1).

The layout of the file is described by the header:

```
/*
 * Header prepended to each a.out file.
 */
struct exec
{
  long          a_magic;    /* magic number */
  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 */
  unsigned long a_talign;   /* text external alignment */
  unsigned long a_dalign;   /* data external alignment */
  unsigned long a_balign;   /* bss external alignment */
  unsigned long a_torigin   /* origin of text segment */
```

```
};

#define OMAGIC   0507  /* old impure format */
#define NMAGIC   0510  /* compromise format */
#define ZMAGIC   0513  /* demand load format */
#define PMAGIC   0515  /* pre-paged load format */
#define LMAGIC   0517  /* pre-paged, non-swapped load format */
```

In the header the sizes of each segment are given in bytes. The size of the header is not included in any of the sizes.

When a nonstandard *a.out* file is executed, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at *a_torigin* in the core image; the header is not loaded. If the magic number in the header is OMAGIC (0507), it indicates that the data segment is immediately contiguous with the text segment. This is the format of the assembler object output and is not executable. If the magic number is ZMAGIC (0513) the data segment begins at the first 0 mod 4096 byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same file, they will share the text segment. For ZMAGIC format, the text segment begins at a 0 mod 4096 byte boundary in the *a.out* file, the remaining bytes after the header in the first block are reserved and should be zero. In this case, the text and data sizes must both be multiples of 4096 bytes, and the pages of the file will be brought into the running image as needed.

The PMAGIC and LMAGIC formats are identical to the ZMAGIC format. However, when a file with the PMAGIC number is executed, it is pre-paged rather than demand-paged. When a file with the LMAGIC number is executed, it is pre-paged and not swapped. This has the effect of locking the program into memory. Only superuser can execute files with the LMAGIC number.

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 at the byte 4096 in the file for ZMAGIC format or just after the header for the other format. The N_TXTOFF macro (shown below) returns this absolute file position when given the name of an exec structure as argument. The data segment 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 include file continues:

```
/*
 * Macros which take exec structures as arguments and tell
 * whether the file has a reasonable magic number or offsets
 * to text | symbols | strings.
 */
#define N_BADMAG(x) \
  (((x).a_magic) != OMAGIC && ((x).a_magic) != ZMAGIC \
&& ((x).a_magic) != PMAGIC && ((x).a_magic) != LMAGIC)

#define N_TXTOFF(x) \
  (((x).a_magic) == ZMAGIC || ((x).a_magic) == PMAGIC || \
   ((x).a_magic) == LMAGIC \
                    ? 4096 : sizeof (struct exec))
```

```
#define N_DATOFF(x)   (N_TXTOFF(x) + (x).a_text)
#define N_TROFF(x)    (N_TXTOFF(x) + (x).a_text + (x).a_data)
#define N_DROFF(x)    (N_TROFF(x)  + (x).a_trsize)
#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)
```

that distinguish symbol types are given in the include file as follows:

```
/*
 * 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, 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 sdb 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 */

/*
 * 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, an SDB entry */

/*
 * Format for namelist values.
 */
#define N_FORMAT "%08x"
```

In the nonstandard *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_fill:4;       /* nothing */
  unsigned int r_extern:1;     /* does not include value of sym */
  unsigned int r_length:2;     /* 0=byte, 1=half, 2=word */
  unsigned int r_pcrel:1;      /* was relocated pc relative already */
  unsigned int r_symbolnum:24; /* local symbol ordinal */
};
```

There is no relocation information if *a_trsize+a_drsize==0*. If *r_extern* is 0, then *r_symbolnum* is actually an *n_type* for the relocation (i.e. N_TEXT meaning relative to text segment origin.)

SEE ALSO

as(1), cc(1), ld(1), sod(1), filehdr(5), opthdr(5), scnhdr(5), *CONVEX C Guide*, *CONVEX C1 Architecture Handbook*, *CONVEX C2/C3 Architecture Reference Manual*, *CONVEX Compiler Utilities User's Guide*

NOTES

In order to run programs in *ieee* mode, the CONVEX machine must be equipped with IEEE hardware. For more information, contact your CONVEX sales representative.

NAME

acct – execution accounting file

SYNOPSIS

#include <sys/time.h>
#include <sys/types.h>
#include <sys/acct.h>

DESCRIPTION

The *acct*(2) system call makes entries in the accounting file */usr/adm/acct* for each process that terminates.

The format of the account record has been changed for post-release-3.0 versions of ConvexOS. The new record contains larger fields in several cases, in order to provide more precision, greater range, or both.

The new and old account records as defined by the include file are:

```
/*      $CHeader: acct.h 1.2 91/01/09 16:24:41 $*/
/*      Copyright 1984-1990 Convex Computer Corp.*/

#ifndef _SYS_ACCT_H_                    /* { _SYS_ACCT_H_*/
#define _SYS_ACCT_H_                    /* { _SYS_ACCT_H_*/
/*
 * Accounting structure
 */

#ifndef _KERNEL
#include <sys/types.h>
#include <sys/time.h>
#else
#include "h/types.h"
#include "h/time.h"
#endif
struct  acct
{
        char            ac_comm[10];    /* Accounting command name */
        struct timeval  ac_utime;       /* Accounting user time */
        struct timeval  ac_stime;       /* Accounting system time */
        struct timeval  ac_etime;       /* Accounting elapsed time */
        time_t          ac_btime;       /* Beginning time */
        u_short         ac_uid;         /* (real) user ID */
        u_short         ac_gid;         /* (real) group ID */
        long            ac_aid;         /* activity ID */
        s64_t           ac_kbsec;       /* memory usage integral */
        long            ac_io;          /* number of disk IO blocks */
        dev_t           ac_tty;         /* control typewriter */
        char            ac_flag;        /* Accounting flag */
        char            ac_unused[5];   /* future expansion */
        float           ac_avconcur;    /* Average concurrency level */
};

/*
 * "Old" (Convex release 3.0 and before) accounting structure
 * this uses a comp_t type which is a 3 bits base 8
 * exponent, 13 bit fraction "floating point" number.
```

```
 */
typedef u_short comp_t_release3;

struct  acct_release3
{
        char            ac_comm[10];  /* Accounting command name */
        comp_t_release3 ac_utime;     /* Accounting user time */
        comp_t_release3 ac_stime;     /* Accounting system time */
        comp_t_release3 ac_etime;     /* Accounting elapsed time */
        time_t          ac_btime;     /* Beginning time */
        short           ac_uid;       /* Accounting user ID */
        short           ac_gid;       /* Accounting group ID */
        short           ac_mem;       /* average memory usage */
        comp_t_release3 ac_io;        /* number of disk IO blocks */
        dev_t           ac_tty;       /* control typewriter */
        char            ac_flag;      /* Accounting flag */
};

#define AFORK                       0001/* has executed fork, but no exec */
#define ASU                         0002/* used super-user privileges */
#define ACOMPAT                     0004/* used compatibility mode */
#define ACORE                       0010/* dumped core */
#define AXSIG                       0020/* killed by a signal */
#define AFIXSCH                     0040/* used fixed scheduling */
#define ARESTART    0100            /* was a (chkpnt-)restarted process */

#ifdef _KERNEL
/* Group daemon has several magic properties */
#define GROUP_DAEMON            1
/* Group mem has special privileges with respect to pattach() */
#define GROUP_MEM   2               /* should match /etc/group */
#endif                              /* } _KERNEL*/
#endif                              /* } _SYS_ACCT_H_*/
```

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

The *ac_kbsec* field contains the integral of the process' resident memory size; it is in units of kilobytes * seconds of execution. Very often (during the kernel routine *hardclock*, generally once every 1/100 second) the system looks at the currently executing process and adds the instantaneous values of its resident text size, resident data size, and resident stack size to the process' cumulative total. When the process **exits** all three cumulative totals are added together and divided by the frequency of the sampling (as stated above, generally once every 1/100 second.)

FILES

/usr/adm/acct

SEE ALSO

lastcomm(1), acct(2), execve(2), sa(8), sumscripts(8),
"Accounting Reports" chapter in the *Managing ConvexOS:Operations Guide.*

NAME

activities – activity list

DESCRIPTION

The *activities* file is an ASCII file which serves two functions. First, it is a list of activities. Second, it is a dictionary between activity names and activity ID numbers. For each activity, there is a single line containing the activity name, followed by a colon, followed by the activity ID. To reduce search time, the activity list should be sorted, with activities referenced most often placed at the beginning of the file.

Activity names may not contain colons, and it is recommended that they not contain spaces so as not to confuse *awk* scripts. NOTE: see *actwho*(5) for more information concerning activity names.

Upon login, activity ID is set to 0. For this reason, the activity with activity ID 0 should be a "catch-all" activity and accordingly should have an entry to this effect in the activities list. An activity ID must fit in 32 bits and must therefore fall in the range -2,147,483,648 to 2,147,483,647. There is a one-to-one mapping between activity names and activity ID's, so that each activity name is associated with exactly one activity ID and vice versa.

When used in conjunction with the *SEt ACTivity_id_offset* capability in *qmgr*(8), activity ID's provide a means of keeping track of CXbatch requests. When a process is executed from a batch queue, the activity ID of the process is set to the activity ID of the user who submitted the job plus the number specified (if any) by the queue's *ACTivity_id_offset* capability. A suggested system is to number all activity ID's 10 apart. Each batch queue would than have an activity id offset having a unique number from 1 to 9, 0 being reserved for jobs that were not submitted using the batch queues.

FILES

/etc/activities

SEE ALSO

bill(1), idtoname(1), setaid(2), getactent(3), acct(5), actwho(5), qmgr(8),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

BUGS

A utility similar to *edactwho* (see *actwho*(5)) should be provided to allow editing and syntax checking.

NAME

actwho – group-activity access control file

DESCRIPTION

The *actwho* file is used by the *bill* command to determine who can bill to which group-activity combinations. Each entry in the *actwho* file is of the form:

group[.[*activity*]]:*users*

Group is any group as listed in */etc/group*.

Activity is any activity as listed in */etc/activities*. In the absence of an *activity* field, the activity with activity ID 0 is assumed. This activity is usually named something like "catch-all" or "miscellaneous".

Users is a comma-separated list of user names who are allowed to bill to the group-activity combination given in the initial part of the entry.

Group, *activity*, or *users* may consist of the single character *, which stands for "all". *Group* or *activity* may contain the *?* character, which matches any character in that character position. With creative naming of activities, this capability can be used to allow certain users to bill to classes of activities. For example, suppose all activity names related to documentation were of the form "3820-xxx-7589", where x is some character that varies with the activity. Then an entry in */etc/actwho* like "*.3820-???-7589:user, ..." would list the users that are allowed to bill to documentation.

To reduce search time, */etc/actwho* should be sorted, with group-activity combinations referenced most often placed at the beginning of the file. It is a good idea to use the *edactwho* utility to edit */etc/actwho* to guarantee correct syntax and to coordinate file locking with the *bill* command.

EXAMPLES

Users **malone, erving,** and **shapira** may bill to the group-activity combination of group **nba** and any activity:

```
nba.*:malone, erving, shapira
```

All users may bill to the group-activity combination of group **sdev** and any activity starting with "017-" and ending in any 4 characters:

```
sdev.017????:*
```

FILES

/etc/actwho

SEE ALSO

bill(1), getacwent(3), activities(5), edactwho(8),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

NAME

aliases – aliases file for sendmail

SYNOPSIS

/usr/lib/aliases

DESCRIPTION

This file describes user id aliases used by */usr/lib/sendmail*. It is formatted as a series of lines of the form

name: name_1, name2, name_3, . . .

The *name* is the name to alias, and the *name_n* are the aliases for that name. Lines beginning with white space are continuation lines. Lines beginning with '**#**' are comments.

Aliasing occurs only on local names. Loops can not occur, since no message will be sent to any person more than once.

After aliasing has been done, local and valid recipients who have a ".forward" file in their home directory have messages forwarded to the list of users defined in that file.

This is only the raw data file; the actual aliasing information is placed into a binary format in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag* using the program *newaliases*(1). However, *newaliases* need not be executed each time the aliases file is changed; *sendmail*(8) periodically checks and rebuilds the database as needed.

SEE ALSO

newaliases(1), dbm(3X), sendmail(8)

BUGS

Because of restrictions in *dbm*(3X) a single alias cannot contain more than about 1000 bytes of information. You can get longer aliases by "chaining"; that is, make the last name in the alias be a dummy name which is a continuation alias.

NAME

ar – archive (library) file format

SYNOPSIS

#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:

```
/*      $CHeader: ar.h 0.0 84/04/23 23:28:25 $*/
/*      Copyright 1984 Convex Computer Corp.*/

#define ARMAG   "!<arch>\n"
#define SARMAG 8

#define ARFMAG "`\n"

struct ar_hdr {
        char    ar_name[16];
        char    ar_date[12];
        char    ar_uid[6];
        char    ar_gid[6];
        char    ar_mode[8];
        char    ar_size[10];
        char    ar_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 a 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 portable across machines. 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

badlogins – log of failed login attempts

DESCRIPTION

This file is obsoleted by the new *login(1)*. *login(1)* now uses the *syslog(3)* facility to report failed login attempts.

SEE ALSO

login(1), syslog(3)
"System Protection" chapter in the *CONVEX System Manager's Guide.*

NAME

bill-acct – log generated by bill command

SYNOPSIS

#include <bill.h>

DESCRIPTION

/usr/adm/bill-acct is a record of successfully executed *bill* commands. The include file *bill.h* defines the structure which is written twice each time any user changes billing accounts from a login shell. The first record written corresponds to the group-activity combination that is terminating; the second corresponds to the new combination. The include file *bill.h* is shown below:

```
/*      $CHeader: bill.h 0.1 86/02/05 16:23:05 $*/
/*      Copyright 1984 Convex Computer Corp.*/
struct billlog {
        short           bi_uid;         /* user id */
        short           bi_gid;         /* new/old group id */
        int             bi_aid;         /* new/old activity id */
        long            bi_time;        /* time as returned by time() */
        char            bi_tty[8];      /* tty line, as in <sys/acct.h> */
        unsigned short  bi_flags;       /* see BI_xx #defines */
        char            bi_unused[4];   /* reserved for future use */
};

#define BILLLOG     "/usr/adm/bill-acct" /* log file */
#define BI_START    0x0001               /* 1 if starting, 0 if ending */
```

FILES

/usr/adm/bill-acct

NOTE

bill commands are only recorded in *bill-acct* if they are executed from a user's login shell. If *bill* commands were recorded when executed from nested shells, then an inconsistency would result when a user exits a nested shell; connect time would be billed to the nested *bill* command's account, while new commands executed by that user would be billed to the higher-level *bill* command's account.

SEE ALSO

bill(1), utmp(5), connecttime(8), sumscripts(8)

NAME

chkpnt – process checkpoint file format

SYNOPSIS

#include <sys/param.h>
#include <convex/core.h>
#include <convex/chkpnt.h>

DESCRIPTION

A chkpnt file is a SOFF format file similar in contents to a *core*(5) file. For more information on the various SOFF header structures, see *a.out*(5). The format of a SOFF style chkpnt file is as follows:

File header
Optional (secondary) file header
Section header for (process) context
Section header for each thread context
Section header for each region or thread region (text, data, or bss)
Section header for checkpoint context
Raw data for process context section
Raw data for each thread context section
Raw data for each region or thread region (text, data, or bss)
Raw data for checkpoint context

The file header will have a magic number of CHKPNT_SOFF_MAGIC, which is defined in <convex/filehdr.h>.

See *core*(5) for a description of process and thread context sections and region sections. The layout of the chkpnt file can be found in <convex/chkpnt.h>.

The raw data for the checkpoint context is organized as follows:

File descriptor section
Memory section
Process data section
Process children section
Pipe data section
String section

The *file descriptor section* contains information for each of the active file descriptors of the checkpointed process. If the file descriptor references a pipe or named pipe this section contains on offset into the *pipe data section* where the associated data is stored.

The *memory section* contains information for each shared memory section mapped in by the process.

The *process data section* contains miscellaneous process state.

The *pipe data section* contains data associated with a pipe or named pipe which has not been read by any process.

The *string section* contains the checkpoint string table where pathnames and filenames are stored. Other sections contain offsets into this table.

SEE ALSO

chkpnt(1), restart(1), core(5)

NAME

contactcap – system configuration file for contact

SYNOPSIS

/usr/lib/contactcap

DESCRIPTION

contactcap is a simplified version of the *termcap*(5) database used to describe the *contact*(1) system configuration. The *contactcap* file is accessed every time *contact* is used, allowing dynamic modification of the data. This database may not be substituted for, as is possible for *termcap*.

CAPABILITIES

Refer to *termcap* for a description of the file layout.

Name	Type	Default	Description
cc	str	"%s"	carbon copy list
ed	str	*/usr/ucb/vi*	text editor
em	str	none	completion message from contact.
mb	str	"contact"	mail box to send report to
ml	num	10000	mail file size limit
nh	str	none	Ethernet host that has UUCP connection
pa	str	"convex!"	UUCP path name to CONVEX
ph	str	"(800) 952-0379"	TAC phone number
rt	bool	false	user "root" may use contact
ta	bool	false	site has tape drive
tn	str	none	technical assistance center name
ul	num	50000	UUCP file size limit
uu	bool	false	site has UUCP connection

The carbon copy field is a list of names to whom a carbon copy of the report will be sent. A %s in this field will be replaced by the name of the user that is submitting the problem report. Names must be separated by blanks.

EXAMPLES

An entry in the */usr/lib/contactcap* file could be:

```
c0|contact|contact site configuration:\
        :cc=%s contact-reports swmgr smith:ph=(800) 952-0379:\
        :ed=/usr/ucb/vi:pa=c1east!convex!:mb=contact:\
        :uu:ta::ml#10000:ul#50000:
```

Entries may continue onto multiple lines by giving a backslash (\) as the last character of a line.

The *cc* field lists that the user (the *%s* is replaced with the user name), *smith*, *swmgr*, and *contact-reports* will receive copies of the contact report. The default editor is *vi*, the machine has a tape drive (*ta*), and mail messages are limited to 10000 bytes. Also, this example is suitable for a machine that *has* a UUCP connection (as evidenced by the existence of the *:uu:* flag in the entry.) The UUCP mail path is given as *c1east!convex!*. The recipient of the mail (found by concatenating the *pa* and *mb* field) is *c1east!convex!contact*, the contact report receiving program. The *ul* field limits the size of transmitted files to 50000 bytes.

For a site that is connected via an Ethernet to another machine which has the UUCP connection, something like *:nh=uucpsite:*, where *uucpsite* names the machine connected to UUCP, should be specified **in addition** to the *:uu:* flag.

contact-reports is a standard alias in */usr/lib/aliases;* users appearing in this alias will receive mail about the status of contact reports. By default, the alias is the data sink */dev/null*.

The next entry may be used by *contact* to track local problem reports.

```
l0|contact|local contact report configuration:\
        :cc=%s contact-reports:ph=123-4567:ed=/usr/ucb/vi:\
        :pa=site!:mb=sysadmin:ml#10000::em=Local contact-reports are filed in "notes local".\
        :tn=John Wiz, Local site troubleshooter.:
```

Here, the user and the mail alias *contact-reports* receive copies of the contact report. The recipient of the mail is *site!sysadmin*, where *site* is the machine name where the system maintainer can be reached. The final message after sending a contact report is *Local*contact-reports*are*filed*in*notes Should a problem occur while running contact, the user is told to contact *John Wiz, Local site troubleshooter.*

The mail features of *contactcap* can be used as an effective interface to record and track contact reports. For example, if a site does not have a UUCP connection but would like to generate hard-copy reports, the *contactcap* entry can specify the line printer as the recipient of the mail as in *:pa=site!:mb=printer:. printer* is a pipe to a line printer program such as */usr/ucb/lpr. site* is a machine connected to a line printer. The mail address might also be an alias which forwards text to a notes file.

SEE ALSO

contact(1), aliases(5), termcap(5)

NAME

core – format of memory image file

SYNOPSIS

#include <sys/param.h>
#include <convex/core.h>

DESCRIPTION

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

The maximum size of a *core* file is limited by *setrlimit*(2). Files which would be larger than the limit are not created.

The layout of the core file can be found in <convex/core.h>. Beginning with Version 7.0 of the CONVEX operating system, the core file layout utilizes the more general SOFF style format. For more information on the various SOFF header structures, see *a.out*(5). Basically, the format of a SOFF style core file is as follows:

File header
Optional (secondary) file header
Section header for (process) context
Section header for each thread context
Section header for each region or thread region (text, data, or bss)
Raw data for process context section
Raw data for each thread context section
Raw data for each region or thread region (text, data, or bss)

The file header will have a magic number of CORE_SOFF_MAGIC, which is defined in <convex/filehdr.h>. The *h_version* field of the file header will be the version (see *vers*(1)) of the executable that caused the core dump, but the *h_timdat* field will be the creation time of the core file itself. Some fields of the file header will be unused in the context of a core file (such as *h_strptr*, *h_strsiz*, *h_flags*, and *h_spares*).

The optional header (defined in <convex/opthdr.h>) will only use the *o_flags* field, which will have the same settings as the executable which caused the creation of the core file.

Unused fields in the section headers (see <convex/scnhdr.h>) include *s_strndx*, *s_relptr*, and *s_nrel*, since core files do not contain string tables or relocation information. The *s_flags* field will indicate what type of data the corresponding section represents, such as (thread) context, (thread) text, (thread) data, or (thread) bss. There are eight different types of sections, and there may be multiple sections of the same type, with the exception of the context section, of which there will be only one.

The context section will hold process-global data such as the process structure (struct proc in <sys/proc.h>) and user structure (struct user in <sys/user.h>). There will be one thread context section from each thread that the system ever noticed running. Since threads can come and go without the kernel ever knowing (the kernel will only be aware of a thread and create a thread structure for those threads that have reason to enter the kernel, such as through a system call or in the process of scheduling), there may have been threads in existence that do not have thread context sections created for them. There may also be thread contexts for "dead" threads - those which did enter the kernel, but have since completed execution. The validity of the thread state (whether it is dead or alive) may be determined by the *t_state* flag in the thread structure. Within each thread context will be its associated thread structure (struct thread in

<sys/thread.h>), vector registers (struct vregs in <convex/reg.h>), and scalar, address, and control registers (struct syscall_context in <convex/reg.h>).

The S_CONTEXT and S_TCONTEXT (defined in <convex/scnhdr.h>) raw data sections will share the same 16-word header (see struct core_header in <convex/core.h>), which will identify the (thread) context type (using such variables as CPU type and OS version). Also at the beginning of the raw data for each of these types of sections will be a structure containing absolute file offsets, which give the location of the actual data structures these sections contain. For example, the first few words of the context section will contain the header (including version and CPU type), followed by the offset of the user structure, then the offset of the process structure. With these offsets, the actual data within the section can easily be found. See <convex/core.h> for the exact definitions of these core (thread) context header structures.

The rest of the sections are self-explanatory (text, data, bss for thread and non-thread) and contain only the raw data.

Since the core file is now a SOFF file, the *sod*(1) utility may be used to examine its contents.

SEE ALSO

adb(1), csd(1)(optional product), sod (1), sigvec(2), getrlimit(2), a.out(5)

NAME

cpio – POSIX compatible extended cpio archive file format

DESCRIPTION

The extended cpio format provides a mechanism supporting the copying of files from an archival/transport medium to the files hierarchy, and from the file hierarchy to an archival/transport medium. The utilities *pax(1)* and *cpio(1)* implement this functionality under the **ConvexOS.**

The byte-oriented *cpio* archive format is a series of entries, each comprised of a header that describes the file, the name of the file, and then the contents of the file.

The archive may be recorded as a series of fixed size blocks of bytes. This blocking be used only to make physical I/O more efficient. The last group of blocks is always at full size.

For the byte-oriented *cpio* archive format, the individual file header information must be in the order indicated and described below.

Field Name	**Length (in bytes)**	**Interpreted as**
c_magic	6	Octal Number
c_dev	6	Octal Number
c_ino	6	Octal Number
c_mode	6	Octal Number
c_uid	6	Octal Number
c_gid	6	Octal Number
c_nlink	6	Octal Number
c_rdev	6	Octal Number
c_mtime	11	Octal Number
c_namesize	6	Octal Number
c_filesize	11	Octal Number
c_name	c_namesize	Pathname String
c_filedata	c_filesize	Data

For each file in the archive, a header as defined above, is written. The information in the header fields is written as streams of ASCII characters interpreted as octal numbers. The octal numbers are extended to the necessary length by appending ASCII zeros at the most significant digit end of the number; the result is written to the stream of bytes most significant digit first. The fields are interpreted as follows:

c_magic shall identify the archive as being a transportable archive by containing the magic bytes as defined by MAGIC (**070707**).

c_dev and *c_ino* shall contain values which uniquely identify the file within the archive (i.e., no files shall contain the same pair of *c_dev* and *c_ino* values unless they are links to the same file). The values shall be determined in an implementation defined manner.

c_mode shall contain the file type and access permission as defined below.

c_uid shall contain the user ID of the owner.

c_gid shall contain the group ID of the owner.

c_nlink shall contain the number of links referencing the file at the time the archive was created.

c_rdev shall contain implementation-defined information for character or block special files.

c_mtime shall contain the latest time of modification of the file at the time the archive was created.

c_namesize shall contain the length of the pathname including the terminating null byte.

c_filesize shall contain the length of the file in bytes. This is the length of the data section following the header structure.

c_name shall contain the pathname of the file. The length of this field in bytes is the value of *c_namesize*. If a file name is found on the medium that would create an invalid pathname, the implementation shall define if the data from the file is stored on the file hierarchy and under what name it is stored. The implementation my choose to ignore these files as long as it produces an error indicating that it has done so.

c_filedata immediately follows *c_filename* containing *c_filesize* bytes of data. The interpretation of such data shall occur in a manner dependent on the type of the file. If *c_filesize* is zero,, no data shall be contained in *c_filedata*.

All characters are represented in the American Standard Code for Information Interchange, ASCII. For maximum portability between implementations, names should be selected from the characters represented by the **portable filename character set** as 8-bit characters with zero parity.

FIFO special files, directories, and the trailer are recorded with *c_filesize* equal to zero. For other special files, *c_filesize* is implementation defined. The header for the next file entry in the archive is written directly after the last byte of the file entry preceding it. A header denoting the filename **'TRAILER!!!'** indicates the end of the archive; the contents of bytes in the last block of the archive following such a header are undefined.

Values needed by *cpio* archive format are described below:

File access permissions:

Name	Value	Indicates
C_IRUSR	000400	Read by owner
C_IWUSR	000200	Write by owner
C_IXUSR	000100	Execute by owner
C_IRGRP	000040	Read by group
C_IWGRP	000020	Write by group
C_IXGRP	000010	Execute by group
C_IROTH	000004	Read by other
C_IWOTH	000002	Write by other
C_IXOTH	000001	Execute by other
C_ISUID	004000	Set uid
C_ISGID	002000	Set gid
C_ISVTX	001000	Reserved

File types:

Name	Value	Indicates
C_ISDIR	040000	Directory
C_ISFIFO	010000	FIFO
C_ISREG	0100000	Regular file
C_ISBLK	060000	Block special file
C_ISCHR	020000	Character special file
C_ISCTG	0110000	Reserved
C_ISLNK	0120000	Reserved
C_ISSOCK	0140000	Reserved

C_ISDIR, C_ISFIFO, and C_ISREG are supported on an IEEE Std 1003.1-1988 conforming system; additional values defined previously are reserved for compatibility with existing systems. Additional file types may be supported; however, such files should not be written on archives intended for transport to portable systems.

C_ISVTX, C_ISCTG, C_ISLNK, and C_ISSOCK have been reserved by this standard to retain compatibility with some existing implementations.

When restoring from an archive:

If the user does not have sufficient privilege to create a file of the specified type, the entry is ignored, and an error issued on standard error.

Only regular files have data to be restored. Presuming a regular file meets any selection criteria that might be imposed by the user, such data is restored.

If a user lacks sufficient privilege to set a particular mode flag, the flag is ignored. If the implementation does not support a particular flag, it too may be ignored.

SEE ALSO

cpio(1), pax(1), P1003.1, Section 10.1.2

NAME

crashdump – crash dump tape

SYNOPSIS

crash dump tape format

DESCRIPTION

Crash dumps are generated from the SPU of a machine that is down using the *crashdump*(8) utility which resides in */mnt/os*. Crash dumps are stored on 2400 foot 6250 bpi magnetic tape. The tape drive must be specified as a non-rewind device for both reading and writing a crash dump.

If a machine has 128 MB or less of physical memory, a crash dump will fit on a single tape, and the tape consists solely of the files as given below.

If a machine has more than 128 MB of physical memory, the first tape consists of the files as given below, but only the first 128 MB of physical memory is present on the first tape. Subsequent tapes consist of a header, followed by up to 128 MB of physical memory.

The files on a crash dump tape are regular files, with the exception of the last file of the first tape, which is in *tar* format. The *tar* format file consists of as many of the files given below as will fit in a 6 megabyte block. If these files take up more than 6 megabytes, then only the first 6 megabytes will be copied (it is most likely that only */mnt/errlog* will be truncated). The header is stored using a block size of 8K; other files use 64K block size. The following files are on a crash dump tape in the order listed:

(1) 8K Header
  (a) A description of what is on the dump tape.
  (b) As many of the most recent entries in */mnt/errlog* as possible are packed into the rest of the header. */mnt/errlog* is included to allow someone looking at a crash dump to quickly determine what caused the crash, without having to read the whole tape. If this does not take up 8K, the remainder of the header is null-padded.
(2) Physical Memory
(3) MBS Memory
(4) CCU Memories
  (a) One file for each IOP's memory.
  (b) One file for each HSP's memory.
  (c) One file for each VIOP's memory.
(5) A *tar*-format file consisting of:
  (a) *vmunix*
  (b) IOP executable
  (c) HSP executable
  (d) VIOP executable
  (e) */mnt/usr/scn/cop.out*
  (f) */ioconfig*
  (g) *bootcmd*
  (h) *tunables*
  (i) *registers* The *registers* file will be present **only** if a hardware dump was not taken. If a hardware dump was taken, *hwdump.out* will contain the CPU registers.
  (j) *commregs*, communication registers for a C2
  (k) *bootcmd.local*, if found
  (l) *hwdump.out*, if hardware dump taken

(m) */mnt/errlog*

SEE ALSO

crashdump(8), crashread(8)

NAME

crontab – cron command list file

SYNOPSIS

home_directory/.crontab

DESCRIPTION

The *cron*(1) daemon executes commands at specified dates and times according to the instructions found in *.crontab* files. A *.crontab* file may reside in any user's home directory.

A *.crontab* file consists of lines of six fields each. Lines which begin with the # character and blank lines are ignored. (Thus you may begin lines which contain comments with #.) The six fields are separated by spaces or tabs. The first five fields are integer patterns to specify the following:

minute (0-59),
hour (0-23),
day of the month (1-31),
month of the year (1-12),
and day of the week (1-7 with 1=Monday).

Each of these patterns may be either an asterisk (meaning all legal values), or a list of elements. A list of elements is either an element, or a set of elements where the elements are separated by commas. An element is either a number in the specified range, or two numbers separated by a minus sign (indicating an inclusive range). The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the Shell. The other lines (or whatever characters and lines following the %) are made available to the command as standard input. If a user forgets to redirect standard output or standard error, *cron* will mail the results of the command to the user.

A user may change the environment in which the commands will run. If a *.cronrc* file exists in a user's home directory, this file will be used to set the environment for each of his commands. A *.cronrc* file consists of lines that set environment variables (e.g., **PATH=:/bin:/usr/ucb:/usr/bin**). *cron* automatically sets the USER, HOME and SHELL variables for the user. Unless otherwise specified in the *.cronrc* file, *cron* will set **PATH=:/usr/ucb:/bin:/usr/bin:/usr/convex** as the default search path, and SHELL to the user's default login shell.

EXAMPLES

On the first day of every month, remove all of the user's ".o" files that have not been accessed for a week:

```
0 0 1 * * find ~ -name "*.o" -atime +7 -exec rm {} \;
```

Every Wednesday afternoon at 1:30 p.m. mail "Weekly staff meeting at 2:00" to the mail-alias called staff:

```
30 13 * * 3 mail staff % Weekly staff meeting at 2:00
```

Change directory to *~/bin* and run "make update" at 6 a.m. and 6 p.m. every day of the year. Direct the output of make to *~/bin/ERRS*:

```
0 6,18 * * * cd ~/bin; make update >& ERRS
```

If the following lines are found in *~/.cronrc*, then the user's *cron* commands will be executed using the Bourne shell and the path used to search for commands will be the given PATH. Please note that the syntax of commands in *~/.crontab* must follow the syntax rules of the executing SHELL (Bourne shell syntax if using */bin/sh* and C shell syntax if using */bin/csh*):

```
SHELL=/bin/sh
PATH=:/usr/local/bin:~/bin:/bin
```

SEE ALSO

cron(1), tellcron(1)

NOTES

If several users share the same home directory, then *cron* will execute the *.crontab* file only for the *.crontab* owner. If the home directory is /, then only the superuser, *root*, can execute commands from */.crontab*.

NAME

dir – format of directories

SYNOPSIS

#include <dirent.h>

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry; see *fs*(5). The directory structure is defined in */usr/include/sys/dir.h* which is included by */usr/include/dirent.h*. The structure is defined as follows:

```
/*
 * A directory consists of some number of blocks of DIRBLKSIZ
 * bytes, where DIRBLKSIZ is chosen such that it can be transferred
 * to disk in a single atomic operation (e.g. 512 bytes on most machines).
 *
 * Each DIRBLKSIZ byte block contains some number of directory entry
 * structures, which are of variable length. Each directory entry has
 * a struct direct at the front of it, containing its inode number,
 * the length of the entry, and the length of the name contained in
 * the entry. These are followed by the name padded to a 4 byte boundary
 * with null bytes. All names are guaranteed null terminated.
 * The maximum length of a name in a directory is MAXNAMLEN.
 *
 * The macro DIRSIZ(dp) gives the amount of space required to represent
 * a directory entry. Free space in a directory is represented by
 * entries which have dp->d_reclen > DIRSIZ(dp). All DIRBLKSIZ bytes
 * in a directory block are claimed by the directory entries. This
 * usually results in the last entry in a directory having a large
 * dp->d_reclen. When entries are deleted from a directory, the
 * space is returned to the previous entry in the same directory
 * block by increasing its dp->d_reclen. If the first entry of
 * a directory block is free, then its dp->d_ino is set to 0.
 * Entries other than the first in a directory do not normally have
 * dp->d_ino set to 0.
 */
#ifdef KERNEL
#define DIRBLKSIZ DEV_BSIZE
#else
#define DIRBLKSIZ 512
#endif

#define MAXNAMLEN 255

/*
 * The DIRSIZ macro gives the minimum record length which will hold
 * the directory entry. This requires the amount of space in struct direct
 * without the d_name field, plus enough space for the name with a terminating
 * null byte (dp->d_namlen+1), rounded up to a 4 byte boundary.
 */
#undef DIRSIZ
#define DIRSIZ(dp) \
  ((sizeof (struct direct) - (MAXNAMLEN+1)) + (((dp)->d_namlen+1 + 3) &~ 3))

struct  direct {
        u_long    d_ino;
        short     d_reclen;
        short     d_namlen;
        char      d_name[MAXNAMLEN + 1];
        /* typically shorter */
};

struct _dirdesc {
        int       dd_fd;
```

```
        long    dd_loc;
        long    dd_size;
        char    dd_buf[DIRBLKSIZ];
};
```

By convention, the first two entries in each directory are for "." and "..". The first is an entry for the directory itself. The second is for the parent directory. The meaning of ".." is modified for the root directory of the master file system ("/"), where ".." has the same meaning as ".".

SEE ALSO

fs(5)

NAME

disktab – disk description file

SYNOPSIS

#include <disktab.h>

DESCRIPTION

disktab is a simple database which describes disk geometries and disk partition characteristics. The format is patterned after the *termcap*(5) terminal data base. Entries in *disktab* consist of a number of ":" separated fields. The first entry for each disk gives the names which are known for the disk, separated by "|" characters. The last name given should be a long name fully identifying the disk.

The following list indicates the normal values stored for each disk entry.

Name	Type	Description
ns	num	Number of sectors per track
nt	num	Number of tracks per cylinder
nc	num	Total number of cylinders on the disk
ba	num	Block size for partition "a" (bytes)
bd	num	Block size for partition "d" (bytes)
be	num	Block size for partition "e" (bytes)
bf	num	Block size for partition "f" (bytes)
bg	num	Block size for partition "g" (bytes)
bh	num	Block size for partition "h" (bytes)
fa	num	Fragment size for partition "a" (bytes)
fd	num	Fragment size for partition "d" (bytes)
fe	num	Fragment size for partition "e" (bytes)
ff	num	Fragment size for partition "f" (bytes)
fg	num	Fragment size for partition "g" (bytes)
fh	num	Fragment size for partition "h" (bytes)
pa	num	Size of partition "a" in sectors
pb	num	Size of partition "b" in sectors
pc	num	Size of partition "c" in sectors
pd	num	Size of partition "d" in sectors
pe	num	Size of partition "e" in sectors
pf	num	Size of partition "f" in sectors
pg	num	Size of partition "g" in sectors
ph	num	Size of partition "h" in sectors
se	num	Sector size in bytes
sp	num	Number of spare sectors per cylinder (IDC ONLY)
ty	str	Type of disk (e.g. removable, winchester)

FILES

/etc/disktab

SEE ALSO

newfs(8)

BUGS

This file shouldn't exist, the information should be stored on each disk or disk pack. For the IDC disks, this information IS stored on the disk but there is no user-level access to the data so it is provided in this file. The file will be obsoleted in a future release of the OS when all disk drivers have been modified to provide this information via ioctl system calls.

NAME

dump, dumpdates – incremental dump format

SYNOPSIS

#include <sys/types.h>
#include <ufs/inode.h>
#include <protocols/dumprestore.h>

DESCRIPTION

Tapes used by *dump (8)* and *restore*(8) contain:

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

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

```
#define CURRENT_VERS 3

#define TP_BSIZE_1    1024
#define TP_BSIZE_2    1024
#define TP_BSIZE_3    2048

#define NTREC_1       10
#define NTREC_2       10
#define NTREC_3       5

#define NTRECG_1      100
#define NTRECG_2      100
#define NTRECG_3      50

#define DUMP_MAGIC_1 (int)60011
#define DUMP_MAGIC_2 (int)60012
#define DUMP_MAGIC_3 (int)60013

#define TP_NINDIR     512

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

union u_spcl {
        char dummy[TP_BSIZE_3];
        struct s_spcl {
                int     c_type;
                time_t c_date;
                time_t c_ddate;
                int     c_volume;
                daddr_t         c_tapea;
                ino_t   c_inumber;
                int     c_magic;
                int     c_checksum;
                struct dinode c_dinode;
                int     c_count;
                char    c_addr[TP_NINDIR];
        } s_spcl;
} u_spcl;

#define spcl u_spcl.s_spcl
```

```
#define     DUMPOUTFMT    "%-16s %c %s"          /* for printf */
                                                 /* name, incno, ctime(date) */
#define     DUMPINFMT     "%16s %c %[^00 /* inverse for scanf */
```

CURRENT_VERS determines which TP_BSIZE, NTREQ, NTRECG, and DUMP_MAGIC are used. Version one is the original dump format of a 4.1 BSD file system or earlier. Version two is a dump of a 4.2 BSD file system or later. Between version one and two the internal structures changed. Version three include changes needed to dump an IDC file system. The only real difference between two and three is the TP_BSIZE had to be increased to 2048.

NTREC is the number of TP_BSIZE byte records in a physical tape block. NTREQG is the number of TP_BSIZE byte records in a physical tape block when the 'G' option is used (see dump(8) man page.) DUMP_MAGIC is the magic number (c_magic in the u_spcl struct above) that is used by restore to determine which version of dump (8) was used to create the file. It is also used by the file (1) program to classify the file as a dump formated file.

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

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

The fields of the header structure are:

c_type	The type of the header.
c_date	The date the dump was taken.
c_ddate	The date the file system was dumped from.
c_volume	The current volume number of the dump.
c_tapea	The current number of this (TP_BSIZE-byte) record.
c_inumber	The number of the inode being dumped if this is of type TS_INODE.
c_magic	This contains the value DUMP_MAGIC above, truncated as needed.
c_checksum	This contains whatever value is needed to make the record sum to CHECKSUM.
c_dinode	This is a copy of the inode as it appears on the file system; see *fs*(5).
c_count	The count of characters in *c_addr*.
c_addr	An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is non-zero. If the block was not present on the file system, no block was dumped; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, TS_ADDR records will be scattered through the file, each one picking up where the last left off.

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

The structure *idates* describes an entry in the file */etc/dumpdates* where dump history is kept. The idates structure is defined internally as follows:

```
struct idates {
        char   id_name[MAXNAMLEN+3];
        char   id_incno;
        time_t id_ddate;
};
```

The fields of the structure are:

id_name	The dumped filesystem is "/dev/*id_nam*".
id_incno	The level number of the dump tape; see *dump*(8).
id_ddate	The date of the incremental dump in system format see *types*(5).

FILES

/etc/dumpdates

SEE ALSO

dump(8), restore(8), fs(5), types(5), file(1)

NAME

failure_log – log of file access failures

SYNOPSIS

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

DESCRIPTION

The *faillog*(2) system call makes an entry in a failure log file for each file access that fails because of insufficient permissions.

Each entry in the log file consists of one copy of the structure below and the name of the accessing command, terminated by a single null character. The entries are not aligned to any boundary other than a byte boundary.

```
/*      $CHeader: faillog.h 1.1 90/11/06 00:17:11 $*/
/*      Copyright 1986 Convex Computer Corp.*/

#ifndef   _SYS_FAILLOG_H_
#define   _SYS_FAILLOG_H_

#ifdef    _KERNEL
#include  "h/types.h"
#else
#include  <sys/types.h>
#endif

/*
 * file access logging structure
 */

struct    faillog
{
    short   lg_version; /* version of this record */
    short   lg_ruid;    /* accessor's real uid */
    short   lg_euid;    /* accessor's effective uid */
    u_short lg_errno;   /* error code returned */
    u_short lg_mode;    /* IREAD | IWRITE | IEXEC */
    dev_t   lg_dev;     /* device on which file lives */
    ino_t   lg_inum;    /* inode number of file */
    time_t  lg_time;    /* time of failed access */
};

#define LG_MAGIC_1 0x0601/* Release 6.1 */
#define LG_MAGIC   LG_MAGIC_1

#endif
```

Failed access logging is not enabled by default. You enable it with the *faillogon*(8) command. The default name for the failed access log is */usr/adm/failure_log*.

SEE ALSO

faillog(2), *faillogon*(8), *faillogpr*(8).
"System Protection" chapter in the *CONVEX System Manager's Guide*.

BUGS

The failure log does not contain enough information to uniquely identify a file. Specifically, if the accessed file is deleted after the failure occurs, it is impossible to determine where that file was in the directory structure.

NAME

filehdr - file header for standard format object files

SYNOPSIS

#include <convex/filehdr.h>

DESCRIPTION

A standard format object file begins with a file header. The following C **struct** is used:

```
struct filehdr {
  unsigned long          h_magic;     /* magic number */
  unsigned long          h_version;   /* version number */
  long                   h_timdat;    /* time & date stamp */
  unsigned long          h_nscns;     /* number of sections */
  unsigned long long     h_scnptr;    /* file ptr to 1st scnhdr */
  unsigned long          h_opthdr;    /* sizeof(optional header) */
  unsigned long long     h_strptr;    /* file ptr to string tab */
  unsigned long long     h_strsiz;    /* # bytes in string tab */
  unsigned long long     h_flags;     /* flag word */
  long                   h_spares[3]; /* room to grow */
};
```

h_scnptr is the byte offset into the file at which the first section header can be found. Its value may be cast and used as the offset to *fseek*(3S) to position an I/O stream to the first section header. Likewise, *h_strptr* may be used as an offset to position an I/O stream to the string table. The optional header is a fixed size; see *opthdr*(5).

The only valid magic number is:

```
#define SOFF_MAGIC    0601
```

There is a magic number reserved for users:

```
#define USER_SOFF_MAGIC 0603
```

The value in *h_timdat* is obtained from the *time*(2) system call. Flag bits currently defined are:

```
#define H_COMM         0x0000000000000001LL /* file has .comm sectns */
#define H_RSVDBITS     0x000000ffffffffffLL /* reserved bits */
#define H_USERBITS     0xffffff0000000000LL /* reserved for users */
```

The flag H_COMM is set if there are initialized common block sections in the file.

filehdr includes some useful definitions:

```
#define FILEHDR           struct filehdr
#define FILEHSZ           sizeof(struct filehdr)
#define IS_SOFF_MAGIC(X)  ((X) == SOFF_MAGIC)
```

NOTES

The user magic number USER_SOFF_MAGIC, and the user flags are not supported in the standard utilities.

SEE ALSO

time(2), fseek(3S), a.out(5), opthdr(5), scnhdr(5)

NAME

fs, inode – format of file system volume

SYNOPSIS

#include <sys/param.h>
#include <ufs/fs.h>
#include <sys/vnode.h>
#include <ufs/inode.h>

DESCRIPTION

Every file system storage volume (disk, nine-track tape, for instance) has a common format for certain vital information. Every such volume is divided into a certain number of blocks. The block size is a parameter of the file system. Sectors 0 to 15 on a file system are used to contain primary and secondary bootstrapping programs.

The actual file system begins at sector 16 with the *super block*. The layout of the super block as defined by the include file *<ufs/fs.h>* is:

```
#define       FS_MAGIC      0x011954
struct fs {
        struct fs *fs_link;             /* linked list of file systems */
        struct fs *fs_rlink;            /*   used for incore super blocks */
        daddr_t       fs_sblkno;            /* addr of super block in filesys */
        daddr_t       fs_cblkno;            /* offset of cyl-block in filesys */
        daddr_t       fs_iblkno;            /* offset of inode-blocks in filesys */
        daddr_t       fs_dblkno;            /* offset of first data after cg */
        long    fs_cgoffset;            /* cylinder group offset in cylinder */
        long    fs_cgmask;              /* used to calc mod fs_ntrak */
        time_t fs_time;                 /* last time written */
        long    fs_size;                /* number of blocks in fs */
        long    fs_dsize;               /* number of data blocks in fs */
        long    fs_ncg;                       /* number of cylinder groups */
        long    fs_bsize;               /* size of basic blocks in fs */
        long    fs_fsize;               /* size of frag blocks in fs */
        long    fs_frag;                /* number of frags in a block in fs */
/* these are configuration parameters */
        long    fs_minfree;             /* minimum percentage of free blocks */
        long    fs_rotdelay;            /* num of ms for optimal next block */
        long    fs_rps;                       /* disk revolutions per second */
/* these fields can be computed from the others */
        long    fs_bmask;               /* ''blkoff'' calc of blk offsets */
        long    fs_fmask;               /* ''fragoff'' calc of frag offsets */
        long    fs_bshift;              /* ''lblkno'' calc of logical blkno */
        long    fs_fshift;              /* ''numfrags'' calc number of frags */
/* these are configuration parameters */
        long    fs_maxcontig;           /* max number of contiguous blks */
        long    fs_maxbpg;              /* max number of blks per cyl group */
/* these fields can be computed from the others */
        long    fs_fragshift;           /* block to frag shift */
        long    fs_fsbtodb;             /* fsbtodb and dbtofsb shift constant */
        long    fs_sbsize;              /* actual size of super block */
        long    fs_csmask;              /* csum block offset */
        long    fs_csshift;             /* csum block number */
        long    fs_nindir;              /* value of NINDIR */
        long    fs_inopb;               /* value of INOPB */
        long    fs_nspf;                /* value of NSPF */
        long    fs_sparecon[6];               /* reserved for future constants */
/* sizes determined by number of cylinder groups and their sizes */
        daddr_t fs_csaddr;              /* blk addr of cyl grp summary area */
        long    fs_cssize;              /* size of cyl grp summary area */
        long    fs_cgsize;              /* cylinder group size */
/* these fields should be derived from the hardware */
        long    fs_ntrak;               /* tracks per cylinder */
        long    fs_nsect;               /* sectors per track */
```

```
        long    fs_spc;                 /* sectors per cylinder */
/* this comes from the disk driver partitioning */
        long    fs_ncyl;                /* cylinders in file system */
/* these fields can be computed from the others */
        long    fs_cpg;                         /* cylinders per group */
        long    fs_ipg;                         /* inodes per group */
        long    fs_fpg;                         /* blocks per group * fs_frag */
/* this data must be re-computed after crashes */
        struct csum fs_cstotal;         /* cylinder summary information */
/* these fields are cleared at mount time */
        uint_t fs_flags;                /* filesystem flags defined below */
        char    fs_fsmnt[MAXMNTLEN]; /* name mounted on */
/* these fields retain the current block allocation info */
        long    fs_cgrotor;             /* last cg searched */
        struct csum *fs_csp[MAXCSBUFS];/* list of fs_cs info buffers */
        long    fs_cpc;                         /* cyl per cycle in postbl */
        short   fs_postbl[MAXCPG][NRPOS];/* head of blocks for each rotation */
        long    fs_magic;               /* magic number */
        u_char fs_rotbl[1];             /* list of blocks for each rotation */
/* actually longer */
};
```

Each disk drive contains some number of file systems. A file system consists of a number of cylinder groups. Each cylinder group has inodes and data.

A file system is described by its super block, which in turn describes the cylinder groups. The super block is critical data and is replicated in each cylinder group to protect against catastrophic loss. This is done at file system creation time and the critical super block data does not change, so the copies need not be referenced further unless disaster strikes.

Addresses stored in inodes are capable of addressing fragments of "blocks". File system blocks of at most size MAXBSIZE can be optionally broken into 2, 4, or 8 pieces, each of which is addressable; these pieces may be DEV_BSIZE, or some multiple of a DEV_BSIZE unit.

Large files consist of exclusively large data blocks. To avoid undue wasted disk space, the last data block of a small file is allocated as only as many fragments of a large block as are necessary. The file system format retains only a single pointer to such a fragment, which is a piece of a single large block that has been divided. The size of such a fragment is determinable from information in the inode, using the "blksize(fs, ip, lbn)" macro.

The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined.

The root inode is the root of the file system. Inode 0 can't be used for normal purposes and historically bad blocks were linked to inode 1, thus the root inode is 2 (inode 1 is no longer used for this purpose, however numerous dump tapes make this assumption, so we are stuck with it). The *lost+found* directory is given the next available inode when it is initially created by *mkfs*.

fs_minfree gives the minimum acceptable percentage of file system blocks which may be free. If the freelist drops below this level only the superuser may continue to allocate blocks. This may be set to 0 if no reserve of free blocks is deemed necessary, however severe performance degradations will be observed if the file system is run at greater than 90% full; thus the default value of *fs_minfree* is 10%.

Empirically the best trade-off between block fragmentation and overall disk utilization at a loading of 90% comes with a fragmentation of 4, thus the default fragment size is a fourth of the block size.

Cylinder group related limits: Each cylinder keeps track of the availability of blocks at different rotational positions, so that sequential blocks can be laid out with minimum rotational latency. NRPOS is the number of rotational positions which are distinguished. With NRPOS 8 the resolution of the summary information is 2ms for a typical 3600 rpm drive.

fs_rotdelay gives the minimum number of milliseconds to initiate another disk transfer on the same cylinder. It is used in determining the rotationally optimal layout for disk blocks within a file; the default value for *fs_rotdelay* is 2ms.

Each file system has a statically allocated number of inodes. An inode is allocated for each NBPI bytes of disk space. The inode allocation strategy is extremely conservative.

MAXIPG bounds the number of inodes per cylinder group, and is needed only to keep the structure simpler by having the only a single variable size element (the free bit map).

N.B.: MAXIPG must be a multiple of INOPB(fs).

MINBSIZE is the smallest allowable block size. With a MINBSIZE of 4096 it is possible to create files of size 2^32 with only two levels of indirection. MINBSIZE must be big enough to hold a cylinder group block, thus changes to (struct cg) must keep its size within MINBSIZE. MAXCPG is limited only to dimension an array in (struct cg); it can be made larger as long as that structure's size remains within the bounds dictated by MINBSIZE. Note that super blocks are never more than size SBSIZE.

The *fs_flags* field keeps file system status information, such as read-only status, dirty status, and sync status.

The path name on which the file system is mounted is maintained in *fs_fsmnt*. MAXMNTLEN defines the amount of space allocated in the super block for this name. The limit on the amount of summary information per file system is defined by MAXCSBUFS. It is currently parameterized for a maximum of two million cylinders.

Per cylinder group information is summarized in blocks allocated from the first cylinder group's data blocks. These blocks are read in from *fs_csaddr* (size *fs_cssize*) in addition to the super block.

N.B.: sizeof (struct csum) must be a power of two in order for the "fs_cs" macro to work.

Super block for a file system: MAXBPC bounds the size of the rotational layout tables and is limited by the fact that the super block is of size SBSIZE. The size of these tables is inversely proportional to the block size of the file system. The size of the tables is increased when sector sizes are not powers of two, as this increases the number of cylinders included before the rotational pattern repeats (*fs_cpc*). The size of the rotational layout tables is derived from the number of bytes remaining in (struct fs).

MAXBPG bounds the number of blocks of data per cylinder group, and is limited by the fact that cylinder groups are at most one block. The size of the free block table is derived from the size of blocks and the number of remaining bytes in the cylinder group structure (struct cg).

Inode: The inode is the focus of all file activity in the UNIX file system. There is a unique inode allocated for each active file, each current directory, each mounted-on file, text file, and the root. An inode is "named" by its device/i-number pair. For further information, see the include file *<ufs/inode.h>*. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

NAME

fstab – static information about filesystems

SYNOPSIS

#include <mntent.h>

DESCRIPTION

The file */etc/fstab* describes the filesystems and swapping partitions used by the local machine. The system administrator can modify it with a text editor. It is read by commands that mount, unmount, dump, restore, and check the consistency of filesystems; also by the system when providing swap space. The file consists of a number of lines of the form:

fsname dir type opts freq passno

for example:

/dev/da0a / 4.2 rw,noquota 1 1

The entries from this file are accessed using the routines in *getmntent*(3), which returns a structure of the following form:

```
struct mntent {
        char *mnt_fsname;/* filesystem name */
        char *mnt_dir;   /* filesystem path prefix */
        char *mnt_type;  /* 4.2, nfs, swap, or ignore */
        char *mnt_opts;  /* rw, ro, noquota, quota, hard, soft */
        int  mnt_freq;   /* dump frequency, in days */
        int  mnt_passno; /* pass number on parallel fsck */
};
```

Fields are separated by white space; a "#" as the first non-white character indicates a comment.

The *mnt_dir* fields is the full path name of the directory to be mounted on.

The *mnt_type* field determines how the *mnt_fsname* and *mnt_opts* fields will be interpreted. Here is a list of the filesystem types currently supported, and the way each of them interprets these fields:

4.2 *mnt_fsname* Must be a block special device.

nfs *mnt_fsname* The path on the server of the directory to be served, in the form *server*:fullpath.

swap *mnt_fsname* Must be a block special device swap partition.

If the *mnt_type* is specified as **ignore** then the entry is ignored. This is useful to show disk partitions not currently used. Note that **ignore** should be specified in the *mnt_type* field for the default swap partition (normally */dev/da0b*). The kernel "mounts" it automatically, so specifying *mnt_type* as **swap** results in an "invalid argument" error at boot time.

The *mnt_opts* field contains a list of comma-separated option words. Some *mnt_opts* are valid for all filesystem types, while others apply only to a specific type:

mnt_opts valid on *all* (both **4.2** and **nfs**) file systems (the default is **rw,suid,noquota**):

rw read/write.

ro read-only.

suid set-uid execution allowed.

nosuid set-uid execution not allowed.

quota usage limits enforced.

noquota usage limits not enforced.

grpid Create files with BSD semantics for propagation of the group ID. With this option, files inherit the group ID of the directory in which they are created, regardless of the directory's setgid bit. Note that on ConvexOS,

local 4.2 file systems always use BSD semantics for the file group ownership when creating files, regardless of the presence or absence of the grpid option.

hide ignore this entry during a **mount -a** command to allow you to define *fstab* entries for commonly used filesystems you don't want to automatically mount.

noauto a synonym for **hide**.

mnt_opts specific to **nfs** (NFS) file systems (the defaults are:

fg,retry=1,timeo=7,retrans=3,port=NFS_PORT,hard,intr

with defaults for *rsize* and *wsize* set by the kernel):

bg if the first mount attempt fails, retry in the background.

fg retry in foreground.

retry=*n* set number times to retry mount to *n*.

rsize=*n* set read buffer size to *n* bytes.

wsize=*n* set write buffer size to *n* bytes.

timeo=*n* set NFS timeout to *n* tenths of a second.

retrans=*n* set number of NFS retransmissions to *n*.

port=*n* set server IP port number to *n*.

soft return error if server doesn't respond.

hard retry request until server responds.

intr allow keyboard interrupts on hard mounts.

nointr don't allow keyboard interrupts on hard mounts.

secure Use a more secure protocol for NFS transactions.

acregmin=*n* Hold cached attributes for at least *n* seconds after file modification.

acregmax=*n* Hold cached attributes for no more than *n* seconds after file modification.

acdirmin=*n* Hold cached attributes for at least *n* seconds after directory update.

acdirmax=*n* Hold cached attributes for no more than *n* seconds after directory update.

actimeo=*n* Set *min* and *max* times for regular files and directories to *n* seconds.

The numeric values of the following fields are returned to applications calling *pathconf(3)* or *fpathconf(3)* to request _PC_LINK_MAX, _PC_NAME_MAX, _PC_NO_TRUNC, B_PC_CHOWN_RESTRICTED, or _PC_PATH_MAX.

link_max=*n*
returned for _PC_LINK_MAX

name_max=*n*
returned for _PC_NAME_MAX

path_max=*n*
returned for _PC_PATH_MAX

no_trunc=*n*
returned for _PC_NO_TRUNC

chown_r=*n*
returned for _PC_CHOWN_RESTRICTED

sysV Inclusion of **sysV** in the options list is shorthand for

link_max=1000,name_max=14,path_max=32767,
no_trunc=0,chown_restricted=0

The **bg** option causes *mount* to run in the background if the server's *mountd*(8C) does not respond. *mount* attempts each request **retry**=*n* times before giving up. Once the filesystem is mounted, each **nfs** request made in the kernel waits **timeo**=*n* tenths of a second for a response. If no response arrives, the time-out is multiplied by **2** and the request is retransmitted. When **retrans**=*n* retransmissions have been sent with no reply a **soft** mounted filesystem returns an error on the request and a **hard** mounted filesystem prints a message and retries the request. The **intr** option (default) allows keyboard interrupts to kill a process that is hung waiting for a response on a hard mounted filesystem. The **nointr** option does not allow keyboard interrupts to kill a process that is hung waiting for a response on a hard mounted filesystem; instead, the operation is guaranteed to complete when the server reboots. The number of bytes in a read or write request can be set with the **rsize** and **wsize** options.

The field *mnt_freq* indicates how often each partition should be dumped by the *dump*(8) command (and triggers that command's **w** option, which determines what filesystems should be dumped). Most systems set the *mnt_freq* field to 1, indicating that filesystems are dumped each day.

The final field, *mnt_passno*, is used by the consistency checking program *fsck*(8) to allow overlapped checking of filesystems during a reboot. All filesystems with *mnt_passno* of **1** are checked first simultaneously, then all filesystems with *mnt_passno* of **2**, and so on. It is usual to make the *mnt_passno* of the root filesystem have the value **1**, and then check one filesystem on each available disk drive in each subsequent pass, until all filesystem partitions are checked. *preen*(8) does not use this field – it determines the checking order based on the names of disk partitions.

The */etc/fstab* file is read only by programs and never written; the system administrator must maintain it manually. The order of records in */etc/fstab* is important because *fsck*, *mount*, and *umount* process the file sequentially; filesystems must appear *after* filesystems they are mounted within.

FILES

/etc/fstab

SEE ALSO

getmntent(3), fsck(8), mount(8), mountd(8C), preen(8), quotacheck(8), quotaon(8)

NOTES

NFS is an optional product; for more information, contact your CONVEX sales representative.

NAME

gateways – gateway data base for routed

DESCRIPTION

The */etc/gateways* file contains static routing information. Each line has the following format:

< **net** | **host** > *name1* **gateway** *name2* **metric** *value* < **passive** | **active** | **external** >

The **net** or **host** keyword indicates if the route is to a network or specific host.

Name1 is the name of the destination network or host. This may be a symbolic name located in */etc/networks* or */etc/hosts* (or, if started after *named*(8), known to the name server), or an Internet address specified in "dot" notation; see *inet*(3). Or this can be **default** This not a good practice on a large network, because it can result in unnecessary packets being routed to the network.

Name2 is the name or address of the gateway to which messages should be forwarded.

Value is a metric indicating the hop count to the destination host or network.

One of the keywords **passive, active** or **external** indicates if the gateway should be treated as *passive* or *active* (as described above), or whether the gateway is *external* to the scope of the *routed* protocol.

FILES

/etc/gateways

SEE ALSO

routed(8c)

NOTES

gateways is an optional product; for more information, contact your CONVEX sales representative.

NAME

gettytab – terminal configuration data base

SYNOPSIS

/etc/gettytab

DESCRIPTION

Gettytab is a simplified version of the *termcap*(5) data base used to describe terminal lines. The initial terminal login process *getty*(8) accesses the *gettytab* file each time it starts, allowing simpler reconfiguration of terminal characteristics. Each entry in the data base is used to describe one class of terminals.

There is a default terminal class, *default*, that is used to set global defaults for all other classes. (That is, the *default* entry is read, then the entry for the class required is used to override particular settings.)

CAPABILITIES

Refer to *termcap*(5) for a description of the file layout. The *default* column below lists defaults obtained if there is no entry in the table obtained, nor one in the special *default* table.

Name	Type	Default	Description
ap	bool	false	terminal uses any parity
ab	bool	false	determine baud-rate after a carriage return.
bd	num	0	backspace delay
bk	str	0377	alternate end of line character (input break)
cb	bool	false	use crt backspace mode
cd	num	0	carriage-return delay
ce	bool	false	use crt erase algorithm
ck	bool	false	use crt kill algorithm
cl	str	NULL	screen clear sequence
co	bool	false	console - add \n after login prompt
ds	str	^Y	delayed suspend character
ec	bool	false	leave echo OFF
ep	bool	false	terminal uses even parity
er	str	^?	erase character
et	str	^D	end of text (EOF) character
ev	str	NULL	initial environment
f0	num	unused	tty mode flags to write messages
f1	num	unused	tty mode flags to read login name
f2	num	unused	tty mode flags to leave terminal as
fd	num	0	form-feed (vertical motion) delay
fl	str	^O	output flush character
hc	bool	false	do NOT hangup line on last close
he	str	NULL	hostname editing string
hn	str	hostname	hostname
ht	bool	false	terminal has real tabs
ig	bool	false	ignore garbage characters in login name
im	str	NULL	initial (banner) message
in	str	^C	interrupt character
is	num	unused	input speed
kl	str	^U	kill character
lc	bool	false	terminal has lower case
lm	str	login:	login prompt
ln	str	^V	"literal next" character
lo	str	/bin/login	program to exec when name obtained
nd	num	0	newline (line-feed) delay
nl	bool	false	terminal has (or might have) a newline character
nx	str	default	next table (for auto speed selection)
op	bool	false	terminal uses odd parity
os	num	unused	output speed
pc	str	\0	pad character
pe	bool	false	use printer (hard copy) erase algorithm

pf	num	0	delay between first prompt and following flush (seconds)
ps	bool	false	read line speed from a Develcon dataswitch
qu	str	^\	quit character
rp	str	^R	line retype character
rw	bool	false	do NOT use raw for input, use cbreak
sp	num	unused	line speed (input and output)
su	str	^Z	suspend character
tc	str	none	table continuation
to	num	0	timeout (seconds)
tt	str	NULL	terminal type (for environment)
ub	bool	false	do unbuffered output (of prompts etc)
uc	bool	false	terminal is known upper case only
we	str	^W	word erase character
xc	bool	false	do NOT echo control chars as ^X
xf	str	^S	XOFF (stop output) character
xn	str	^Q	XON (start output) character

If no line speed is specified, speed will not be altered from that which prevails when getty is entered. Specifying an input or output speed will override line speed for stated direction only.

Terminal modes to be used for the output of the message, for input of the login name, and to leave the terminal set as upon completion, are derived from the boolean flags specified. If the derivation should prove inadequate, any (or all) of these three may be overridden with one of the **f0**, **f1**, or **f2** numeric specifications, which can be used to specify (usually in octal, with a leading '0') the exact values of the flags. Local (new tty) flags are set in the top 16 bits of this (32 bit) value.

Should *getty* receive a null character (presumed to indicate a line break) it will restart using the table indicated by the **nx** entry. If there is none, it will re-use its original table.

Delays are specified in milliseconds, the nearest possible delay available in the tty driver will be used. Should greater certainty be desired, delays with values 0, 1, 2, and 3 are interpreted as choosing that particular delay algorithm from the driver.

The **cl** screen clear string may be preceded by a (decimal) number of milliseconds of delay required (a la termcap). This delay is simulated by repeated use of the pad character **pc**.

The initial message, and login message, **im** and **lm** may include the character sequence **%h** to obtain the hostname. (**%%** obtains a single '%' character.) The hostname is normally obtained from the system, but may be set by the **hn** table entry. In either case it may be edited with **he**. The **he** string is a sequence of characters, each character that is neither '@' nor '#' is copied into the final hostname. A '@' in the **he** string, causes one character from the real hostname to be copied to the final hostname. A '#' in the **he** string, causes the next character of the real hostname to be skipped. Surplus '@' and '#' characters are ignored.

When getty execs the login process, given in the **lo** string (usually "/bin/login"), it will have set the environment to include the terminal type, as indicated by the **tt** string (if it exists). The **ev** string, can be used to enter additional data into the environment. It is a list of comma separated strings, each of which will presumably be of the form *name=value*.

If a non-zero timeout is specified, with **to**, then getty will exit within the indicated number of seconds, either having received a login name and passed control to *login*, or having received an alarm signal, and exited. This may be useful to hangup dial in lines.

Output from *getty* is even parity unless **op** is specified. **Op** may be specified with **ap** to allow any parity on input, but generate odd parity output. Note: this only applies while getty is being run, terminal driver limitations prevent a more complete implementation. *Getty* does not check parity of input characters in *RAW* mode.

SEE ALSO

termcap(5), getty(8).

BUGS

The default special characters change frequently, so it is wise to always specify (at least) the erase, kill, and interrupt characters in the **default** table. In **all** cases, '#' or '^H' typed in a login name will be treated as an erase character, and '@' will be treated as a kill character.

Apart from its general lack of flexibility, some of the delay algorithms are not implemented. The terminal driver should support reasonable delay settings.

Currently *login*(1) stomps on the environment, so there is no point setting it in *gettytab*.

Termcap format is bad; something more rational should have been chosen.

NAME

group – group file

SYNOPSIS

/etc/group

DESCRIPTION

group contains the following information for each group:

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

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line. The group ID field should contain a number in the range of zero to 32767. CONVEX has "reserved" gids 0 through 99 for system use. No ordinary users should be assigned gids in this range. The gids are site dependent; however, CONVEX distributes its releases with the following assumptions.

```
daemon:*:1:daemon
kmem:*:2:
sys:*:3:
tty:*:4:
nogroup:*:8:
bin:*:10:root
notes:*:13:notes
operator:*:28:root
guest:*:31:root
uucp:*:40:
staff:*:49:
```

This file resides in the */etc* directory. It has general read permission and can be used, for example, to map numerical group ID's to names.

A group file can have a line beginning with a plus (+), which means to incorporate entries from the yellow pages. There are two styles of + entries: All by itself, + means to insert the entire contents of the yellow pages group file at that point; *+name* means to insert the entry (if any) for *name* from the yellow pages at that point. If a + entry has a non-null group member field, the contents of that field will override what is contained in the yellow pages. The numerical group ID field cannot be overridden. An entry of the form: *- groupname* indicates that the group is disallowed. All subsequent entries for the indicated *groupname* , whether originating from the Yellow Pages, or the local *group* file, are ignored.

EXAMPLE

```
+mygroup:::bob,steve
+:
```

If these entries appear at the end of a group file, then the group *mygroup* will have members *bob* and *steve*, and the group ID of the yellow pages entry for the group *mygroup*. All the groups listed in the yellow pages will be pulled in and placed after the entry for *mygroup*.

FILES

/etc/group

SEE ALSO

idtoname(1), passwd(1), setgroups(2), crypt(3), initgroups(3X), activities(5), passwd(5), "Accounting Reports" chapter in the *Managing ConvexOS:Operations Guide*.

NOTES

The "encrypted password" field became obsolete with the release of Berkeley 4.2BSD UNIX. This field now consists of a * character. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

The plus and minus entries are only recognized for sites running the Yellow Pages. Yellow Pages is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

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 which contains the following information should be present:

Internet address
official host name
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 besides a field delimiter, newline, or comment character. The "official hostname" field should be a machine's hostname as given by *hostname*(1).

FILES

/etc/hosts

SEE ALSO

gethostbyname(3N)

BUGS

A binary indexed file format should be available for fast access.

NOTES

There are programs which rely on the fields of */etc/hosts* being in the following order: "Internet address" followed by "official host name" optionally followed by "aliases". These programs will not work if the fields are out of order.

hosts is an optional product; for more information, contact your CONVEX sales representative.

NAME

hosts.equiv, rhosts – list of trusted hosts, remote host access file

DESCRIPTION

hosts.equiv resides in directory */etc* and contains a list of trusted hosts. When an *rlogin*(1) or *rsh*(1) request from such a host is made, and the initiator of the request is in */etc/passwd*, then no further validity checking is done. That is, *rlogin* does not prompt for a password, and *rsh* completes successfully. So a remote user is "equivalenced" to a local user with the same user ID when the remote user is in *hosts.equiv*.

The format of *hosts.equiv* is a list of names, as in this example:

```
host1
host2
+@group1
-@group2
```

A line consisting of a simple host name means that anyone logging in from that host is trusted. A line consisting of *+@group* means that all hosts in that network group are trusted. A line consisting of *-@group* means that hosts in that group are not trusted. Programs scan *hosts.equiv* linearly, and stop at the first hit (either positive for hostname and +@ entries, or negative for -@ entries). A line consisting of a single + means that everyone is trusted.

The *.rhosts* file has the same format as *hosts.equiv*. When user *XXX* executes *rlogin* or *rsh*, the *.rhosts* file from *XXX*'s home directory is conceptually concatenated onto the end of *hosts.equiv* for permission checking. However, -@ entries are not sticky. If a user is excluded by a minus entry from *hosts.equiv* but included in *.rhosts*, then that user is considered trusted. In the special case when the user is root, then only the */.rhosts* file is checked. The *.rhosts* file may not have write permission for group, or other. If the permission is set incorrectly the file is ignored.

It is also possible to have two entries (separated by a single space) on a line of these files. In this case, if the remote host is equivalenced by the first entry, then the user named by the second entry is allowed to log in as anyone, that is, specify any name to the –l flag (provided that name is in the */etc/passwd* file, of course). Thus

```
convex1 john
```

allows *john* to log in from convex1 as anyone. The usual usage would be to put this entry in the *.rhosts* file in the home directory for *bill* . Then *john* may log in as *bill* when coming from convex1. The second entry may be a netgroup, thus

```
+@group1 +@group2
```

allows any user in *group2* coming from a host in *group1* to log in as anyone.

FILES

/etc/hosts.equiv
~/.rhosts
/.rhosts
/etc/yp/*domain*/netgroup
/etc/yp/*domain*/netgroup.byuser
/etc/yp/*domain*/netgroup.byhost

SEE ALSO

rlogin(1c), rsh(1c), netgroup(5)

NOTES

The plus and minus entries are only recognized for sites running the Yellow Pages. Yellow Pages is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

Hostname entries must be the real hostname, not an alias.

NAME

lpd-acct – printer accounting file

DESCRIPTION

Printer accounting is done by the print filters. There may be more than one printer accounting file, as determined by */etc/printcap*; however, most CONVEX filters use */usr/adm/lpd-acct*. For consistency, these filters use the same log routine (shown below). Accounting data in this format may be summarized by *awk* scripts as described in *sumscripts*(8).

```
#include <stdio.h>

log(acctfile, pages, userinfo, logname, host)
char    *acctfile;
int     pages;
char    *userinfo;
char    *logname;
char    *host;
{
        FILE    *fptr;

        if (acctfile == NULL)  /* not specified in argument list for filter */
                return;
        if ((fptr = fopen (acctfile, "a")) == NULL)
                return;
        fprintf (fptr, "%10d %6d %s %8s %8s0, time (0), pages,
                userinfo != NULL ? userinfo : "0 0 0 unknown", logname, host);
        fclose (fptr);
}
```

The arguments *acctfile*, *logname*, and *host* are generated as described by the *Berkley 4.2BSD Line Printer Spooling Manual* in the *CONVEX Tutorial Papers*. The *pages* argument is computed by the filter itself.

As a CONVEX modification to *lpd*, if the *pu* capability (propagate user information) is present in a printer's */etc/printcap* entry, then *lpd* will pass the command-line arguments **-u userinfo** to the printer's filter(s). The print filter can then pass the *userinfo* portion to the log routine.

NOTE: the *pu* capability should be included in any */etc/printcap* entry which routes jobs through a print filter that expects *userinfo*. This also applies to remote */etc/printcap* files; the *pu* capability should be included in any */etc/printcap* entry on a remote machine which routes jobs to a local print filter that expects *userinfo*. (Print filters write to */usr/adm/lpd-acct* via the log routine given above, which expects *userinfo*. If the log routine does not receive *userinfo*, the log routine's current behavior, which is subject to change, is to print "0 0 0 unknown" in the place of *userinfo*.) A commonly-used filter which needs the *pu* capability in its */etc/printcap* entries is *lpf*(8).

FILES

/usr/adm/lpd-acct

SEE ALSO

lpr(1), printcap(5), lpd(8), lpf(8), pac(8), sumscripts(8),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

NAME

magic – data base of magic number checks used by the file(1) utility

SYNOPSIS

/usr/etc/magic

DESCRIPTION

The *magic* file is a primitive data base used by the file(1) command to help find information on a particular object or executable file.

Basically, the fields of this file are as follows:

byte offset,
value type/length, (byte, short, string, long, llong)
optional operator (= by default),
value to match (numeric or string),
and string to be printed.

The valid operators are '=', '>', '<', and '&' (to perform a bit mask). All fields are separated by one or more tabs, except that the operator (if there is one) must be flush up against the value to compare to or match. Numeric values may be decimal, octal, or hex.

If the primary operator is '&', there may be two items in the line not found for the other types of operators, namely a secondary operator and another value. This extra data must be flush against the first (mask) value, to prevent confusion with the string to be printed. In this case, the lines will have a format similar to the following:

<offset> <type> &<mask>=0 <some quality this file has>

If these extra fields are not there, the mask will have the effect of testing for non-zero (are any of the bits in the mask set?).

A '>' may be found occasionally in the first column of a line, and it has special meaning. It forces *file* to continue scanning and matching additional lines. The first line afterwards not so marked terminates the search.

The last string may have a single *printf* format spec which will be replaced by the *value* field on the same line in the magic file.

EXAMPLE ENTRIES

The following pairs of lines will each compare the 32-bit number at offset 0 in a file to the value designated in the third field. If they are equal (because a missing operator implies '='), the string (in the last field) will be printed. Then the next line will be used in a comparison because it begins with a '>.'

In all cases, any secondary lines will only be checked if the starting line has "found a match." For example, once the first 32 bits are determined to be equal to 0513, the long value found at an offset of 8 bytes are checked to be greater than 0. If the second check holds true, then "not stripped" is printed.

Offset	**Type**	**(Op)Value**	**String**
0	long	0507	Convex old-style object
>8	long	>0	not stripped
0	long	0513	Convex old-style demand paged executable
>8	long	>0	not stripped

Below is an example of how the masking function may be used. Located at an offset of 88 bytes is basically a 64-bit flags word. Using subsequent lines like the following allows one to check the values of various bits in the flag word.

0	long	0601	Convex SOFF
>88	llong	&0x1	demand paged
>88	llong	&0x2	pre-paged
>88	llong	&0x2000000000000000=0	not stripped
>88	llong	&0x1800000000000000=0x1000000000000000	ieee fpmode

FILES

/usr/etc/magic file containing magic number checks

SEE ALSO

file(1)

NAME

mqueue – sendmail mail queue directory and queue files

SYNOPSIS

/usr/spool/mqueue/?f.*id*

DESCRIPTION

If sendmail (delivering mail) returns a temporary failure exit status, the message is queued. A control file is used to describe the recipients to be sent to and various other parameters. This control file is formatted as a series of lines, each describing a sender, a recipient, the time of submission, or some other important parameter of the message. The header of the message is stored in the control file, so that the associated data file in the queue is the temporary file originally collected.

The Mail Queue

The mail queue should be processed transparently. However, you may find that manual intervention is necessary. For example, if a major host is down for a period of time the queue can become clogged. Although sendmail should recover gracefully when the host comes up, you may find performance unacceptably bad in the meantime.

Printing (displaying) the queue

The contents of the queue can be displayed using the *mailq* command or by specifying the **–bp** flag to sendmail. The command is

```
mailq
```

This produces a list of the queue ids, the size of the message, the date the message entered the queue, and the sender and recipients.

Format of Queue Files

All queue files have the form *xfAA99999* where *AA99999* is the *id* for this file and the *x* is a type. The types are:

- d The data file. The message body (excluding the header) is kept in this file.
- l The lock file. If this file exists, the job is currently being processed, and a queue run will not process the file. For that reason, an extraneous **lf** file can cause a job to apparently disappear even thought it will never time out.
- n This file is created when an id is created. It is a separate file to ensure that no mail can be destroyed due to a race condition. It should exist for no more than a few milliseconds at any given time.
- q The queue control file. It contains the information necessary to process the job.
- t A temporary file. An image of the **qf** file while the **qf** is rebuilt. It should be renamed to a **qf** file quickly.
- x A transcript file that exists during a session and shows everything that occurs during that session.

The **qf** file is structured as a series of lines each beginning with a code letter. The lines are as follows:

- D The name of the data file. There can only be one of these lines.
- H A header definition, can be multiple lines. The order of these lines is important since they represent the order in the final message. These use the same syntax as header definitions in the configuration file.
- R A recipient address that is normally aliased, but is realiased when the job is

processed. There is one line for each recipient.

S The sender address. There can only be one sender address line.

E An error address. If such lines exist, they represent the addresses that should receive error messages.

T The job creation time used to compute when to time out a job.

P The current message priority used to select the order in which the queue is processed. Higher numbers reflect lower priorities. The priority changes as messages sit in the queue. The initial priority depends on the message class and the size of the message.

M A message. This line is printed by the *mailq* command, and is used to store status information. It can contain any text.

For example, the following is a queue file sent to "ruser@bigc" and "wka":

```
DdfA13557
Sjohn
T404261372
P132
Rruser@bigc
Rwka
H?D?date: 23-Oct-82 15:49:32-PDT (Sat)
H?F?from: john (John Doe)
H?x?full-name: John Doe
Hsubject: this is an example message
Hmessage-id: <8209232249.13557@UCBARPA.BERKELEY.EDU>
Hreceived: by UCBARPA.BERKELEY.EDU (3.227 [10/22/82])
        id A13557; 23-Oct-82 15:49:32-PDT (Sat)
HTo: ruser@bigc, wka
```

This shows the name of the data file, the person who sent the message, submission time (in seconds since January 1, 1970), message priority, message class, the recipients, and the headers for the message.

SEE ALSO

sendmail(8), sendmail.cf(5), mailq(8), aliases(5), newaliases(1).

FILES

/usr/spool/mqueue	The directory */usr/spool/mqueue* should be created to hold the mail queue. This directory should be mode 755 and owned by root.

NAME

/etc/mtab – mounted file system table

SYNOPSIS

#include <mntent.h>

DESCRIPTION

mtab resides in the */etc* directory, and contains a table of filesystems currently mounted by the *mount* command. *umount* removes entries from this file.

The file contains a line of information for each mounted filesystem, structurally identical to the contents of */etc/fstab*, described in *fstab*(5). There are a number of lines of the form:

fsname dir type opts freq passno

for example:

/dev/da0a / 4.2 rw,noquota 1 1

The file is accessed by programs using *getmntent*(3), and by the system administrator using a text editor.

FILES

/etc/mtab

SEE ALSO

getmntent(3), fstab(5), mount(8)

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 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 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.

FILES

/etc/networks

SEE ALSO

getnetent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NOTES

networks is an optional product; for more information, contact your CONVEX sales representative.

NAME

/etc/nurc – nu defaults database

DESCRIPTION

The */etc/nurc* file contains default constants for the *nu*(8) program which adds new users to the system. *Nu* uses the defined constants if no other values are provided. Lines in the *nurc* file have the form *tag: value*. For a full listing of *tags* and *values* see the table in the *nu*(8) man page.

EXAMPLE

Here's an example nurc file:

```
directory: /mnt
protection: 0700
group: 49
```

In this example a new user's directory will be (by default) created in the */mnt* directory (*directory: /mnt*) and that directory will be readable, writable and executable by the user only (*protection: 0700*). The new user's default group id will be *49*.

The *value* provided must be a valid constant for the given *tag*. The *value* given to the *protection* tag must be a valid argument to *chmod*(1).

FILES

/etc/nurc

SEE ALSO

nu(8)

NAME

op.access – list of operator mnemonics and restrictions enforced by the op program

SYNOPSIS

/etc/op.access

DESCRIPTION

The file *op.access* describes what operator-type functions can be performed by the *op*(8) program, how they are to be performed, and who is allowed to perform them. For security reasons, this file is shipped as read-only by root (mode 0400), and these permissions should not be changed.

The fields of the entries in *op.access* are separated by white space (blanks, tabs, or newlines). Each entry may span several lines and continues until the next alphanumeric string is found at the beginning of a line (which is taken to be the next *mnemonic*, and thus the beginning of a new entry). The # character is used to begin comments and everything from a # through the next newline is ignored. Comments are allowed within an entry's description. The entries describe:

- the name, or mnemonic, of the operator function
- the command line (full path names only) to execute to perform this function
- the uid to set (root by default)
- the gid to set (not changed, by default)
- the group vector to set (contains only gid, by default)
- the directory to *chdir*(2) to (not changed, by default)
- the root directory to set with *chroot*(2) (not changed, by default)
- the umask to set (022 by default)
- a list of groups allowed to execute this function (none by default)
- a list of users allowed to execute this function (none by default, except the superuser)
- the range of valid arguments for the command (any value per variable argument by default)
- any number of environment variable settings (none by default).

Because the syntax of the *op.access* file is fairly free-format, there are necessarily certain characters that have special meanings as field separators or clarifiers; these characters may not be used in any of the strings that make up the fields unless the strings are quoted. These special characters are the comma (,), semi-colon (;), equal sign (=), dollar sign ($), and pound sign (#), as well as the following regular expression characters: the period (.), left square bracket ([), backslash (\) and asterisk (*). Backslash-escaping these characters only works within a quoted string.

Each entry in *op.access* has the following form:

mnemonic *command* [*arg* ...] ; [*option* ...]

Where:

mnemonic
: is a unique identifier for each operator function. It should be alphanumeric and may not contain any white space. This is a required field and must appear as the first item on a line. It can be no more than 100 characters long. If there exists more than one entry in the *op.access* file for any *mnemonic*, the first one will silently override.

command
: is the full pathname of the executable that should be run by *op* when the associated *mnemonic* is chosen. This is a required field, immediately following the *mnemonic*, separated from it only by white space.

arg(s)
: are any number of either literal or variable arguments needed by *command*. Literal arguments are simply specified directly, like specific command options (**0Gun**) or files (**/dev/rmt20**). Variable arguments are specified here as **$1**, **$2** ... **$***n*; these are described more fully in the options section, but have the connotation one would expect: for example, **$1** refers to the first variable argument that is given on the *op* command line, immediately following the *mnemonic* being selected. There can be no more than 63

variable arguments per *mnemonic*. An *arg* of $* as the last *arg* (just before the required semi-colon) means there can exist any number of trailing arguments (including zero). This special token should not be quoted, as other strings containing an asterisk must be.

option(s)

are a set of optional parameters to specify settings or restrictions for the particular *mnemonic*, define variable arguments specified for the *command*, and/or define environment variable settings. All options are separated by white space and are almost exclusively of the form *keyword=value*. The absence of a specific option means the default is sufficient. The *value* can be a single value or a list of values separated by commas, where appropriate. There should be no white space in each element of the *value* string, unless quoted. The *keyword* is one of the following types:

spec where *spec* is one of:

uid Set the uid to the value specified. The value can be a numeric user ID or a login name. The default is **root**, unless an effective uid is specified. If **euid** is listed as an option, but **uid** is not, then only the effective uid will be changed, and the real uid will remain set to that of the invoking user.

euid Set the effective uid to the value specified. The value can be a numeric user ID or a login name. The default is to set the effective uid to the same value as the real uid. This option is useful for setting an effective uid to root to run a privileged command, but leaving the real uid alone for programs such as *opreq*(1) that may want, for example, to look in a user's home directory for a configuration or setup file.

gid Set the gid (group vector) to the value(s) specified. Each value can be a numeric group ID or a group name. The default is to leave the gid unchanged. If a list of values is given here, the first will be taken to be the gid to set, and they will all be used to set up the group vector for the invoked command.

egid Set the effective gid to the value specified. The value can be a numeric group ID or a group name. The default is to set the effective gid to the same value as the real gid. If the **gid** option is also given, the value(s) given there will simply be used in forming the group vector to initialize.

dir Change the current working directory to the path specified. The default is to leave the current working directory unchanged.

chroot Change the root directory to the path specified; see *chroot*(2) for more information. Please note that changing the root directory causes all full pathnames to begin with the new root as a prefix, including the *command* being executed. The default is to leave the root directory unchanged.

umask Set the file creation umask to the octal value specified. The default is to set it to **022**.

unsetenv

Unset each of the specified environment variables. This option is only allowed on normal mnemonic entries (not the special DEFAULT line), and it is used to unset any environment variable that is set in the DEFAULT line (see description of DEFAULT line below) because a certain mnemonic does not want to set the given variables.

groups

Allow any user who belongs to a group listed here to execute this *op* function. The default is not to allow any specific group.

users Allow any user listed here to execute this *op* function. The default is not to allow any specific users. However, access permissions are not checked at all for the superuser (root is allowed to run anything, even if not explicitly listed here); only the arguments are verified.

$*n* where *n* is a positive integer, defines the *n*th variable argument specified in the command *arg* list. The value for this type may be a comma-separated list of literal possibilities, and/or regular expressions. A regular expression is defined to be a non-quoted string containing one or more of the following special regular expression characters (they must not be backslash-escaped): a period (.), left square bracket ([), backslash (\) or asterisk (*). Expressions bracketed by \(and \) can later be referred to using the standard *n* backreference notation, and this capability has been expanded to "carry over" between regular expressions. For a description of regular expressions that are recognized, see the *ed*(1) manual page. There are also regular expression examples in the sample *op.access* file below.

This *option* defines the values allowed for each of the variable arguments. If a variable argument is specified as a command *arg*, but not described in the *options* section, then any value (except an empty string) is allowed (a default of $*n*==.*). If a variable argument is specified as a command *arg*, then a non-null value must be given for it in the *op* command line unless "" is specified in the *options* section as one of its valid values, which indicates an optional argument. If an optional argument is something other than the last one, then "" must be explicitly given on the *op* command line to indicate which argument is being skipped.

If an argument does not match, then a diagnostic is printed (and logged with *syslog(3)*), and the command is not executed.

$* is used in the *options* section to place restrictions on any and all of the trailing arguments that are specified as $* in the *args* section. If any variable arguments are given on the *op* command line that fit into this category, and there are restrictions placed on them in the *options* section of the *op.access* file entry, then *each* of these trailing arguments are checked for validity against the restrictions listed.

If any of these (possibly many) arguments do not match, then a diagnostic is printed (and logged with *syslog(3)*), and the command is not executed.

$*VAR* where *VAR* is the name of an environment variable. Basically, an item is assumed to be an environment variable if it is an alphanumeric (the underscore character (_) is also allowed) string that begins with a dollar sign ($) followed by a letter. The specified environment variable is set to the value given before the command is executed.

If an environment variable is specified alone (that is, without the "= *value*" portion), the current value set for that variable will be passed through to the invoked command. This is useful for keeping commonly-used variables such as $HOME and $TERM set to valid and meaningful values.

There can also be a special entry in the file beginning at the first non-comment line which can define default values to override the builtin defaults listed here, yet still be overridden by any entry that wants to redefine any of the *spec* fields described above. It should have the following format:

DEFAULT *spec_option ...*

where *spec_option* is one of the *spec=value* strings mentioned above under *options*. The *spec* element is one of **uid**, **euid**, **gid**, **egid**, **dir**, **chroot**, **umask**, **groups**, or **users**. Environment variables (either explicitly set via $*VAR*=*value* or with the existing value passed through via $*VAR*) are also allowed on this special DEFAULT line.

It should be noted that if any regular *mnemonic* entry defines its own *option*, the value given for that entry must explicitly include the item from the DEFAULT line if the default value is to be included. That is, the *options* definitions completely override any defaults; they do not add to them. In this way, if a value specified on the DEFAULT line for **users** or **groups** (for example) needs to be "erased" without redefining new values (that is, we want no users or groups to be allowed to run this mnemonic), then the default value must be overridden with nothing (as in **users=**). For the **users** and **groups** fields, such a null setting has the effect of setting the list of allowable users or groups to be empty. For the other *specs* (**uid**, **euid**, **gid**, **egid**, **dir**, **chroot**, and **umask**), a null setting leaves that attribute as it is upon invocation of the *op* program (overriding any defaults).

This file format seems complex at first glance, but is actually intuitive and flexible. See the example below for further explanation of how to take advantage of the *op* tool's capabilities.

EXAMPLES

An example *op.access* file might look like:

```
#
# first, define the site defaults we want to use here
# we would like the people in 'operators' group to be able to execute
#   almost everything, so it is easier to put it here than on every line...
DEFAULT     groups=operators
# filesystem backups
weekly      /etc/dump 0Gun $1; users=snow,white $1=/,/usr,/mnt,/project
daily       /etc/dump 5Gun $1; users=snow,white $1=/,/usr,/mnt,/project
#
# tape handling commands
#   must include 'operators' if we want them to be allowed
tpconfig    /usr/convex/tpconfig ; groups=tapeopers,operators users=sysmgr
tapequeue   /usr/convex/tpconfig set que $1 ; groups=tapeopers $1=e.*, d.*
#
# to run the operator request manager, we want to have $TERM appropriately
#    set;  also only want to change the effective uid, not both
opreq /usr/convex/opreq ; groups=tapeopers $TERM $HOME euid=root
#
# taking the system down
#   $1 shows a good use of regular expressions; $2 can be anything
shutdown    /etc/shutdown -h $1 $2; $1=now,+[0-9]*,[0-9]*:[0-9]*
reboot      /etc/shutdown -r $1 $2; $1=now,+[0-9]*,[0-9]*:[0-9]*
#
# kill all batch procs so system can be restarted
#   (want to override default groups setting given at top, allowing none)
killbatch   /etc/opbin/kill_batch_procs; groups= users=bashful,happy
startbatch  /etc/opbin/start_batch;
#
# start up disco daemon
disco       /etc/opbin/start_disco; uid=disco gid=proj dir=/scratch
                  umask=027 groups=geo,disco users=snoopy,woodstck
                  $USER=disco $SHELL=/bin/shell
#
# let certain people mount and unmount the removable drive
```

```
rdsmount    /etc/mount $1 $2; groups=operators,swdev,disco dir=/
      users=bob,steve $1=/dev/dd0. $2=/.*
rdsumount   /etc/umount $1; groups=operators,swdev,disco dir=/
      users=bob,steve $1=/dev/dd0.
#
# let all engineers nfs mount directories in a "known" standard place
#   this shows good use of the ".) - 1 backreference capabilities
nfsmount    /etc/mount -o "timeo=100,hard,intr" $1 $2; groups=engr
            $1=\([a-zA-Z0-9_]*\):\(.*\)
            $2=/rmt/\1\2
```

Some example operator command lines, given the above *op.access* file, might be:

```
op weekly /mnt
op tapequeue disable
op reboot 17:30 "We have to fix our network."
op disco
op rdsmount /dev/dd0c /mnt/me/mystuff
op mounted 3 8688
op nfsmount convex1:/os2/julie /rmt/convex1/os2/julie
```

FILES

/etc/op.access file containing operator mnemonics and restrictions for *op* tool

SEE ALSO

op(8), regex(3), "Operator Interface" chapter in the *CONVEX System Manager's Guide*

NAME

opthdr - optional (secondary) header for standard format object files

SYNOPSIS

#include <convex/opthdr.h>

DESCRIPTION

Every standard format object file has an optional (secondary) file header. The following C **struct** is used:

```
struct opthdr {
    unsigned long long          o_symptr;   /* file ptr to symbol tab */
    unsigned long               o_nsyms;    /* # of entries in sym tab */
    long                        o_spare;    /* reserved - must be 0 */
    unsigned long               o_entry;    /* entry point */
    unsigned long long          o_flags;    /* flag word */
};
```

o_symptr is the byte offset into the file at which the symbol table can be found. Its value may be cast and used as the offset in *fseek*(3S) to position an I/O stream to the symbol table. *o_nsyms* is the number of symbols in the table.

The *o_flags* field of the optional header defines attributes of the file. Currently, the defined flags are:

```
#define OF_DEMAND           0x0000000000000001LL /* demand paged */
#define OF_PREPAGED         0x0000000000000002LL /* pre-paged */
#define OF_NON_SWAP         0x0000000000000004LL /* pre-pg, no swap */
#define OF_RSVD_EXEC_BITS   0x0000000000000ff8LL /* future expansion */
#define OF_EXEC_MASK        0x0000000000000fffLL /* all exec bits */
#define OF_RESERVED0        0x0400000000000000LL /* must be 0 */
#define OF_NOT_VECTOR       0x0800000000000000LL /* no vector inst */
#define OF_FP_MODE_NATIVE   0x0000000000000000LL /* native mode file */
#define OF_FP_MODE_IEEE     0x1000000000000000LL /* ieee mode file */
#define OF_FP_MODE_MIXED    0x1800000000000000LL /* dual mode file */
#define OF_IEEE_MASK        0x1800000000000000LL /* all mode bits */
#define OF_STRIPPED         0x2000000000000000LL /* file is stripped */
#define OF_OBJECT           0x4000000000000000LL /* file is object */
#define OF_EXEC             0x8000000000000000LL /* executable file */
```

Opthdr.h includes some useful definitions:

```
#define OPTHDR          struct opthdr
#define OPTHSZ          sizeof(struct opthdr)
```

NOTE

Although the header is called *optional*, it always appears in standard format object files.

SEE ALSO

ld(1), fseek(3S), a.out(5), filehdr(5), scnhdr(5)

NAME

passwd – password file

SYNOPSIS

/etc/passwd

DESCRIPTION

The *passwd* file contains for each user the following information:

name — User's login name — by convention login names must be all lower case alphabetic characters or numerics and must begin with an alphabetic character. Although some utilities work fine with login names which don't follow these conventions, CONVEX advises that these conventions be followed. Login names must be eight characters or less in length.

password — This is the user's encrypted password.

numerical user ID

This is the user's ID in the system and, by convention, is unique across the system. The sharing of uids is strongly discouraged. uid 0 has special meaning to ConvexOS: It is the "superuser" account and care should be taken to ensure ordinary users do not have this uid. uids must be in the numeric range of 0 to 32767. CONVEX has "reserved" uids 0 through 99 for system use. No ordinary users should be assigned uids in this range. The following passwd entries are assumed in CONVEX's releases.

```
root::0:10:ConvexOS:/:/bin/csh
daemon:*:1:1:The devil himself:/:/bin/csh
convex:*:2:10:Convex Computer Corp.,,,2149520200:/:/bin/csh
anonymous:*:4:40:Anon notes:/usr/spool/notes:/bin/csh
anon:*:4:40:Anon notes:/usr/spool/notes:/bin/csh
nouser:*:8:8:Generic Non-User:/tmp:/bin/false
notes:*:10:13:Notesfile Owner:/usr/spool/notes/.utilities:/bin/csh
notesys:*:10:13:Notesfile Administrator:/usr/spool/notes/.utilities:/bin/csh
uucp::14:40:UNIX-to-UNIX Copy:/usr/spool/uucppublic:/usr/lib/uucp/uucico
test:*:16:49:System Exerciser:/tmp:/bin/csh
```

(UNIX is a registered trademark of UNIX System Laboratories, Inc.)

numerical group ID

This the user's "primary" group id. A user may be added to additional "secondary" groups by modifying the */etc/group* file. See *group*(5) for more information. gids must be in the numeric range of 0 to 32767. CONVEX has "reserved" gids 0 through 99 for system use. No ordinary users should be assigned gids in this range.

user's real name

In some versions of UNIX, this field also contains the user's office, extension, home phone, and so on. For historical reasons this field is called the GCOS field.

initial working directory

The directory that the user is positioned in when they log in — this is known as the "home" directory.

shell — program to use as Shell when the user logs in.

The user's real name field may contain "&", meaning insert the login name.

The password file is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, */bin/sh* is used.

The *passwd* file may have lines beginning with a minus (-), which means to ignore the specified entries from the yellow pages. There are two styles of - entries: *-name* means to ignore the entry (if any) for *name* in the yellow pages at that point; *-@name* means to ignore the entries for all members of the network group *name* at that point.

- qualifiers should be used before a solitary + and not in conjunction with more specific inclusions.

The *passwd* file can also have lines beginning with a plus (+), which means to incorporate entries from the yellow pages. There are three styles of + entries: all by itself, + means to insert the entire contents of the yellow pages password file at that point; *+name* means to insert the entry (if any) for *name* from the yellow pages at that point; *+@name* means to insert the entries for all members of the network group *name* at that point. If a + entry has a non-null password, directory, GCOS, or shell field, they will override what is contained in the yellow pages. The numerical user ID and group ID fields cannot be overridden.

EXAMPLE

Here is a sample */etc/passwd* file:

```
root:q.mJzTnu8icF.:0:10:God:/:/bin/csh
jack:6k/7KCFRPNVXg:508:10:jack Watson:/mnt/jack:/bin/csh
+john:
+@documentation:no-login:
+:::Guest
```

In this example, there are specific entries for users *root* and *jack*, in case the yellow pages are out of order. The user *john* will have his password entry in the yellow pages incorporated without change; anyone in the netgroup *documentation* will have their password field disabled, and anyone else will be able to log in with their usual password, shell, and home directory, but with a GCOS field of *Guest*.

The password file resides in the */etc* directory. Because of the encrypted passwords, it has general read permission and can be used, for example, to map numerical user ID's to names.

A site may opt for additional password security. The security level is tunable by enforcing none, one or both of the measures described below.

If *password aging* is enabled for a particular user, the user cannot change his password for a specified length of time and is forced to change the password after some other length of time has elapsed. If *password typing* is enabled for a user, the composition of characters in the password must meet certain minimum requirements (see *passwd*(1) and *pwrestrict*(5) for more details.)

The file */etc/pwrestrict* contains the password restriction information and is supplemental to */etc/passwd*. The *pwrestrict* file is optional; if restrictions are enabled, special care must be taken when updating a user's password information. The */etc/passwd* and */etc/pwrestrict* files should only be edited using the *vipw*(8) utility.

FILES

/etc/passwd /etc/passwd.dir /etc/passwd.pag
/etc/pwrestrict /etc/pwrestrict.dir /etc/pwrestrict.pag

SEE ALSO

bill(1), chfn(1), finger(1), idtoname(1), login(1), passwd(1), crypt(3), getpwent(3), getpwrestent(3), group(5), pwrestrict(5), mkpasswd(8), nu(8), vipw(8)

BUGS

User information (name, office, etc.) should be stored in some other file, not in the GCOS field.

NOTES

The plus and minus entries are only recognized for sites running the Yellow Pages. Yellow Pages is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

phones – remote host phone number data base

DESCRIPTION

The file */etc/phones* contains the system-wide private phone numbers for the *tip*(1C) program. This file is normally unreadable, and so may contain privileged information. The format of the file is a series of lines of the form: <system-name>[\t]*<phone-number>. The system name is one of those defined in the *remote*(5) file and the phone number is constructed from [0123456789-=*K]. The "=" and "*" characters are indicators to the auto call units to pause and wait for a second dial tone (when going through an exchange). The "=" is required by the DF02-AC and the "*" is required by the BIZCOMP 1030. The "K" character must be used for Hayes modems as the pause indicator, because the comma (",") is used by *tip*(1C) to separate phone numbers in a list (as may be found in a *remote*(5) file).

Only one phone number per line is permitted. However, if more than one line in the file contains the same system name *tip*(1C) will attempt to dial each one in turn, until it establishes a connection.

FILES

/etc/phones

SEE ALSO

tip(1C), remote(5)

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 *plot*(1G). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by 4 bytes representing the x and y values; each value is a signed (short) integer. The last designated point in an **l, m, n,** or **p** instruction becomes the 'current point' for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in *plot*(3X).

m move: The next 4 bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next 4 bytes. See *plot*(1G).

p point: Plot the point given by the next 4 bytes.

l line: Draw a line from the point given by the next 4 bytes to the point given by the following 4 bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a newline.

a arc: The first 4 bytes give the center, the next 4 give the starting point, and the last 4 give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise.

c circle: The first 4 bytes give the center of the circle, the next 2 the radius.

e erase: Start another frame of output.

f linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are 'dotted,' 'solid,' 'longdashed,' 'shortdashed,' and 'dotdashed.' Effective only in *plot 4014* and *plot ver.*

s space: The next 4 bytes give the lower left corner of the plotting area; the following 4 give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *plot*(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 isn't square.

4014	space(0, 0, 3120, 3120);
ver	space(0, 0, 2048, 2048);
300, 300s	space(0, 0, 4096, 4096);
450	space(0, 0, 4096, 4096);

SEE ALSO

graph(1G), plot(1G), plot(3X)

NAME

printcap – printer capability database

SYNOPSIS

/etc/printcap

DESCRIPTION

printcap is a simplified version of the *termcap*(5) database used to describe line printers. The spooling system accesses the *printcap* file every time it is used, allowing dynamic addition and deletion of printers. Each entry in the database is used to describe one printer. This database may not be substituted for, as is possible for *termcap*, because it may allow accounting to be bypassed.

The default printer is normally *lp*, although the environment variable PRINTER may be used to override this. Each spooling utility supports an option, *–Pprinter*, to allow explicit naming of a destination printer.

Refer to the *Managing ConvexOS: Configuration guide* for a complete discussion on how to set up the database for a given printer.

CAPABILITIES

Refer to *termcap* for a description of the file layout.

Name	Type	Default	Description
af	str	NULL	name of accounting file
br	num	none	if *lp* is a *tty*, set the baud rate (ioctl call)
cf	str	NULL	cifplot data filter
df	str	NULL	tex data filter (DVI format)
fc	num	0	if *lp* is a *tty*, clear flag bits (*sgtty.h*)
ff	str	\f	string to send for a form feed
fo	bool	false	print a form feed when device is opened
fs	num	0	like *fc* but set bits
gf	str	NULL	graph data filter (*plot*(3X) format)
ic	bool	false	driver supports (nonstandard) ioctl to indent printout
if	str	NULL	name of text filter which does accounting
lf	str	*/dev/console*	error logging filename
lo	str	*lock*	name of lock file
lp	str	*/dev/lp*	device name to open for output
mx	num	1000	maximum file size (in BUFSIZ blocks - see /usr/include/stdio.h), zero = unlimited
nd	str	NULL	next directory for list of queues (unimplemented)
nf	str	NULL	*ditroff* data filter (device independent *troff*)
of	str	NULL	name of output filtering program
pl	num	66	page length (in lines)
pu	bool	false	propagate user information to filters (*lpd-acct*(5))
pw	num	132	page width (in characters)
px	num	0	page width in pixels (horizontal)
py	num	0	page length in pixels (vertical)
rf	str	NULL	filter for printing FORTRAN style text files
rm	str	NULL	machine name for remote printer
rp	str	*lp*	remote printer name argument
rs	bool	false	restrict remote users to those with local accounts
rw	bool	false	open the printer device for reading and writing
sb	bool	false	short banner (one line only)
sc	bool	false	suppress multiple copies
sd	str	*/usr/spool/lpd*	spool directory
sf	bool	false	suppress form feeds
sh	bool	false	suppress printing of burst page header
st	str	*status*	status filename
tc	str	NULL	entry of similar printer - must be last
tf	str	NULL	*troff* data filter (cat phototypesetter)
tr	str	NULL	trailer string to print when queue empties

vf	str	NULL	*raster* image filter
xc	num	0	if *lp* is a *tty*, clear local mode bits (*tty*(4))
xs	num	0	like *xc* but set bits

Error messages sent to the console have a carriage return and a line feed appended to them, rather than just a line feed.

If the local line printer driver supports indentation, the daemon must understand how to invoke it.

SEE ALSO

lpq(1), lpr(1), lprm(1), lpd-acct(5), termcap(5), lpc(8), lpd(8),

Setting Up the Line Printer System in
Managing ConvexOS: Configuration Guide

Managing the Line Printer System in
Managing ConvexOS: Operations Guide

"Accounting Reports" chapter in the *Managing ConvexOS:Operations Guide.*

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 the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Protocol names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/protocols

SEE ALSO

getprotoent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NOTES

Protocols is an optional product; for more information, contact your CONVEX sales representative.

NAME

pwrestrict – password restrictions file

SYNOPSIS

/etc/pwrestrict

DESCRIPTION

The *pwrestrict* file contains the following information for each user:

name	The user's login name — contains no upper case characters and must not be greater than eight characters long.
password	Encrypted password.
age	This is the password aging information used to enforce regular changing of the login password.
typed	This is a yes/no field which turns password type restrictions on and off, respectively.
uid	The user's login id.

The password restrictions file is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a new-line.

Password aging is effected for a particular user if the age field is a non-null string. The age is composed as follows: the first subfield of the age, m, say, denotes the minimum period in weeks which must expire before the password may be changed. The next subfield, M, say, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after his password has become out of date will be forced to supply a new one. The third field is a calendar date recording the date of the last password change. Subfields are separated by commas. M and m may be any arbitrarily chosen integer; the date may be in any form recognized by *calendar*(1).

If $\mathbf{m} = \mathbf{M} = 0$ 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 restrictions file). That is, to get a once-only password change the age should be set to "0,0". If $m > M$ only the super-user will be able to change the password. If the age field is null, no aging is applied to the login password. The age string must be introduced in the first instance by the super-user.

Warning: Having $m > M$ and $M = 0$ effectively disables the user from ever logging in. A maximum age field of 0 forces the user to change his password, but he is subsequently disallowed because only super-user may change this password.

The Typed field may be a "Y" or "N" which will turn on or turn off, respectively, the type restrictions for passwords (described in *passwd*(1)). If the Typed field is null, any composition of characters in the password will be allowed (this is equivalent to turning the type restrictions off).

The *pwrestrict* file can have lines beginning with a plus (+), which means to incorporate entries from the yellow pages. There are three styles of + entries: all by itself, + means to insert the entire contents of the yellow pages password restrictions file at that point; *+name* means to insert the entry (if any) for *name* from the yellow pages at that point; *+@ name* means to insert the entries for all members of the network group *name* at that point. If a + entry has a non-null age or type-restriction field, they will override what is contained in the yellow pages.

Note that the password and uid field are not used per se here; the password and uid are always extracted from the passwd file. The information is repeated here so that all information about a user's password may be found by looking at the *pwrestrict* file.

EXAMPLE

Here is a sample */etc/pwrestrict* file:

```
root:q.mJzTnu8icF.:1,5,Jun 1 1986:Y:0
smith:6k/7KCFRPNVXg:::508
+walt:
+@documentation:
```

In this example, there are specific entries for users *root* and *smith*, in case the yellow pages are out of order. The user *root* has password aging and password typing enabled. The user *smith* has no restrictions applied to his login password. The user *walt* will have his password restrictions, listed in the yellow pages, incorporated without change; an entry for anyone in the netgroup

documentation will be included in this file.

The *pwrestrict* file resides in the */etc* directory. Because of the encrypted passwords, it has general read permission. If the *pwrestrict* file is empty or non-existent, no restrictions are applied.

/etc/pwrestrict should only be edited with the *vipw*(8) utility.

FILES

/etc/pwrestrict

SEE ALSO

login(1), passwd(1), getpwrestent(3), passwd(5), nu(8), vipw(8)

NAME

rcsfile – format of RCS file

DESCRIPTION

An RCS file is an ASCII file. Its contents is described by the grammar below. The text is free format, i.e., spaces, tabs and new lines have no significance except in strings. Strings are enclosed by "@". If a string contains a "@", it must be doubled.

The meta syntax uses the following conventions: "|" (bar) separates alternatives; "{" and "}" enclose optional phrases; "{" and "}*" enclose phrases that may be repeated zero or more times; "{" and "}+" enclose phrases that must appear at least once and may be repeated; "<" and ">" enclose nonterminals.

```
<rcstext>     ::=  <admin> {<delta>}* <desc> {<deltatext>}*

<admin>       ::=  head          {<num>};
                   access        {<id>}*;
                   symbols       {<id> : <num>}*;
                   locks         {<id> : <num>}*;
                   comment       {<string>};

<delta>       ::=  <num>
                   date          <num>;
                   author        <id>;
                   state         {<id>};
                   branches      {<num>}*;
                   next          {<num>};

<desc>        ::=  desc          <string>

<deltatext>   ::=  <num>
                   log           <string>
                   text          <string>

<num>         ::=  {<digit>{.}}+

<digit>       ::=  0 | 1 | ... | 9

<id>          ::=  <letter>{<idchar>}*

<letter>      ::=  A | B | ... | Z | a | b | ... | z

<idchar>      ::=  Any printing ASCII character except space,
                   tab, carriage return, new line, and <special>.

<special>     ::=  ; | : | , | @

<string>      ::=  @{any ASCII character, with "@" doubled}*@
```

Identifiers are case sensitive. Keywords are in lower case only. The sets of keywords and identifiers may overlap.

The <delta> nodes form a tree. All nodes whose numbers consist of a single pair (e.g., 2.3, 2.1, 1.3, etc.) are on the "trunk", and are linked through the "next" field in order of decreasing numbers. The "head" field in the <admin> node points to the head of that sequence (i.e., contains the highest pair).

All <delta> nodes whose numbers consist of 2n fields (n≥2) (e.g., 3.1.1.1, 2.1.2.2, etc.) are linked as follows. All nodes whose first (2n)-1 number fields are identical are linked through the "next" field in order of increasing numbers. For each such sequence, the <delta> node whose number is identical to the first 2(n-1) number fields of the deltas on that sequence is called the branchpoint. The "branches" field of a node contains a list of the numbers of the first nodes of all sequences for which it is a branchpoint. This list is ordered in increasing numbers.

Example:

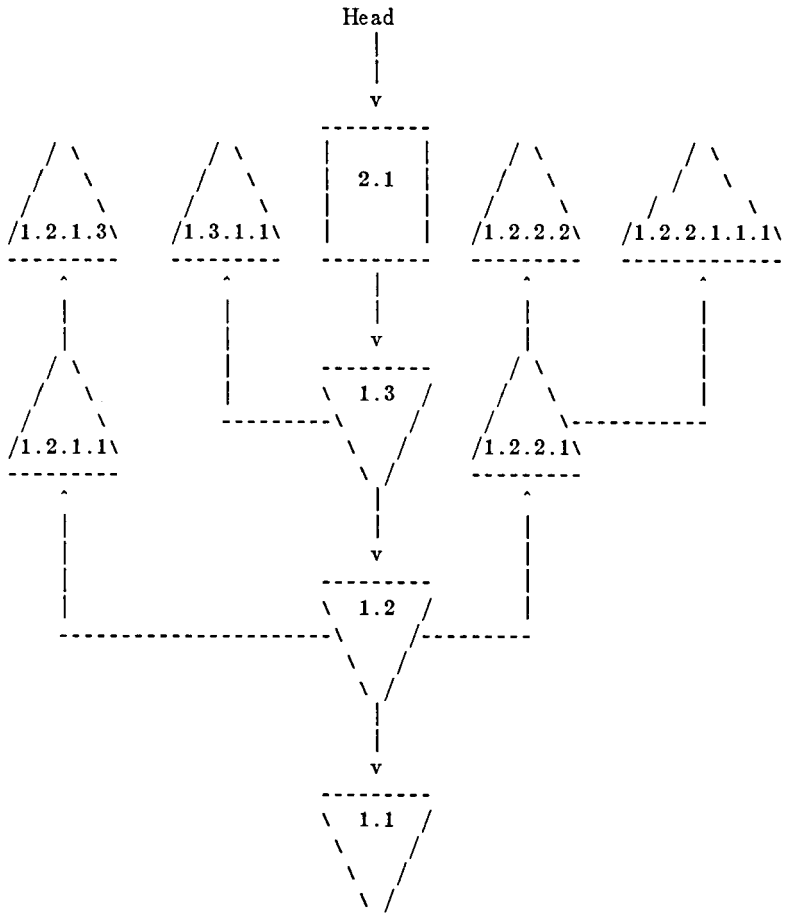

Fig. 1: A revision tree

SEE ALSO

ci(1), co(1), ident(1), rcs(1), rcsdiff(1), rcsmerge(1), rlog(1)

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

remote – remote host description file

DESCRIPTION

The systems known by *tip*(1C) and their attributes are stored in an ASCII file which is structured somewhat like the *termcap*(5) file. Each line in the file provides a description for a single *system*. Fields are separated by a colon (":"). Lines ending in a \ character with an immediately following newline are continued on the next line.

The first entry is the name(s) of the host system. If there is more than one name for a system, the names are separated by vertical bars. After the name of the system come the fields of the description. A field name followed by an "=" sign indicates a string value follows. A field name followed by a "#" sign indicates a following numeric value.

Entries named "tip*" and "cu*" are used as default entries by *tip*, and the *cu* interface to *tip*, as follows. When *tip* is invoked with only a phone number, it looks for an entry of the form "tip300", where 300 is the baud rate with which the connection is to be made. When the *cu* interface is used, entries of the form "cu300" are used.

CAPABILITIES

Capabilities are either strings (str), numbers (num), or Boolean flags (bool). A string capability is specified by *capability=value*; e.g. "dv=/dev/harris". A numeric capability is specified by *capability#value*; e.g. "xa#99". A Boolean capability is specified by simply listing the capability.

at (str) Auto call unit type.

br (num) The baud rate used in establishing a connection to the remote host. This is a decimal number. The default baud rate is 300 baud.

cm (str) An initial connection message to be sent to the remote host. For example, if a host is reached through port selector, this might be set to the appropriate sequence required to switch to the host.

cu (str) Call unit if making a phone call. Default is the same as the "dv" field.

di (str) Disconnect message sent to the host when a disconnect is requested by the user.

du (bool) This host is on a dial-up line.

dv (str) Device(s) to open to establish a connection. If this file refers to a terminal line, *tip*(1C) attempts to perform an exclusive open on the device to ensure only one user at a time has access to the port. More than one device may be listed if each device name is separated by commas. If more than one is listed, *tip*(1C) will traverse the list in the order given to find an open device. In this example *tip* will try to open ttyd3 first. If that device is already being used by another user, ttyd2 will be opened. If ttyd2 is busy, ttyc3 will be tried. If ttyc3 is busy *tip* will give up and not complete the connection.

```
modems24:\
        :dv=/dev/ttyd3,/dev/ttyd2,/dev/ttyc3:at=hayes:br#2400:
remotesys:\
        :pn=95551212:tc=modems24:
```

el (str) Characters marking an end-of-line. The default is NULL. "~" escapes are only recognized by *tip* after one of the characters in "el", or after a carriage-return.

fs (str) Frame size for transfers. The default frame size is equal to BUFSIZ.

hd (bool) The host uses half-duplex communication, local echo should be performed.

ie (str) Input end-of-file marks. The default is NULL.

oe (str) Output end-of-file string. The default is NULL. When *tip* is transferring a file, this string is sent at end-of-file.

pa (str) The type of parity to use when sending data to the host. This may be one of

"even", "odd", "none", "zero" (always set bit 8 to zero), "one" (always set bit 8 to 1). The default is even parity.

pn (str) Telephone number(s) for this host. If the telephone number field contains an @ sign, *tip* searches the file */etc/phones* file for a list of telephone numbers; c.f. *phones*(5).

tc (str) Indicates that the list of capabilities is continued in the named description. This is used primarily to share common capability information.

Here is a short example showing the use of the capability continuation feature:

```
dial-1200:\
        :dv=/dev/cua0:el=^D^U^C^S^Q^O@:du:at=ventel:ie=#$%:oe=^D:br#1200:
arpavax|ax:\
        :pn=7654321%:tc=dial-1200
```

FILES

/etc/remote

SEE ALSO

tip(1C), phones(5)

NAME

resolver – resolver configuration file

SYNOPSIS

/etc/resolv.conf

DESCRIPTION

The resolver configuration file contains information that is read by the resolver routines the first time they are invoked by a process. The file is designed to be human readable and contains a list of name-value pairs that provide various types of resolver information.

On a normally-configured system this file should not be necessary. The only name server to be queried will be on the local machine and the domain name is retrieved from the system.

The different configuration options are:

nameserver

followed by the Internet address (in dot notation) of a name server that the resolver should query. At least one name server should be listed. Up to MAXNS, as defined in *<resolv.h>* (currently 3), name servers may be listed, in that case the resolver library queries tries them in the order listed. If no **nameserver** entries are present, the default is to use the name server on the local machine. (The algorithm used is to try a name server, and if the query times out, try the next, until out of name servers, then repeat trying all the name servers until a maximum number of retries are made).

domain

followed by a default domain name. This default domain will be appended to names which do not already have a dot in them. If no **domain** entries are present, the resolver routines will approximate a default domain based on the hostname returned by *gethostname*(2). If the hostname returned contains a dot, everything right of the left-most dot is used as the default domain name. If the hostname returned does not contain a dot, the root domain is the default domain.

The name value pair must appear on a single line, and the keyword (e.g. **nameserver**) must start the line. The value follows the keyword, separated by white space.

NOTES

If the empty file **/etc/use_nameserver** does not exist, *named* will not be used. Be sure to use the *touch*(1) command to create this file when working with *named*.

FILES

/etc/resolv.conf

/etc/use_nameserver This empty file must exist if named will be used.

SEE ALSO

gethostbyname(3N), resolver(3), named(8)

Name Server Operations Guide for BIND

NAME

scnhdr - section header for standard format object files

SYNOPSIS

#include <convex/scnhdr.h>

DESCRIPTION

Every standard format object file has a table of section headers to specify the layout of the data within the file. Each section within an object file has its own header. The following C **struct** is used:

```
/*
 * s_vaddr and s_align are synonyms, s_align is used for alignment data in
 * object files, since they can't have virtual addresses, and s_vaddr is
 * used in executable images, since they already have the alignment data
 * figured in to the load address.
 */
#define s_align      s_vaddr    /* also used for alignment data */

struct scnhdr {
  unsigned long long s_strndx;  /* section name index in string table */
  unsigned long    s_vaddr;     /* virtual load address of section  */
  unsigned long long s_size;    /* size of section in bytes */
  unsigned long long s_scnptr;  /* offset in file to raw data */
  unsigned long long s_relptr;  /* offset in file to relocation data */
  unsigned long    s_nrel;      /* number of relocation entries */
  unsigned long    s_prot;      /* protection flags (see pte.h) */
  unsigned long long s_flags;   /* flags word */
};
```

File pointers are byte offsets into the file; they can be cast and used as the offset in a call to the *fseek()* function. The *s_relptr* field is the byte offset into the file at which the section relocation information can be found. *s_scnptr* is an offset into the file where the section raw data may be found.

If a section is initialized, the file contains the actual bytes. An uninitialized section is somewhat different. It has a size, symbols defined in it, and symbols that refer to it, but it can have no relocation entries, or data. Consequently, an uninitialized section has no raw data in the object file, and the values for *s_scnptr*, *s_relptr*, and *s_nrel* are zero.

The defined flags for the *s_flags* field are:

```
#define S_TEXT      0x00000001LL /* .text section */
#define S_DATA      0x00000002LL /* .data section */
#define S_UDATA     0x00000003LL /* unshared data section */
#define S_BSS       0x00000004LL /* bss section */
#define S_UBSS      0x00000005LL /* unshared bss section */
#define S_COMON     0x00000006LL /* common block section */
#define S_TYPMASK   0x000000ffLL /* mask to get section type */
#define S_FMTMASK   0x0000ff00LL /* mask to get format flags */
#define S_RSVDBITS  0x00000fffffffffffLL /* reserved bits */
#define S_USERBITS  0xfffff0000000000LL /* reserved for users */
```

s_size for an initialized common section includes a table of contents which precedes the raw data. The table of contents is a map of the raw data. The first four bytes of the table gives the number of table entries. Each table entry is a tuple of the form (offset, length), where offset is the offset into the common data, and length is the size of the datum (e.g. 1 byte for a character datum). The table of contents is used by the *ld* utility to load multiply defined common sections. The table of contents structures are:

```
long count_toc_entries;          /* count of table entries */

struct toc_entry {
  long long  offset_cvar;      /* offset of datum in section */
  int      size_cvar;        /* size of datum in section */
};
```

scnhdr includes some useful definitions:

```
#define SCNHDR  struct scnhdr
#define SCNHSZ  sizeof(struct scnhdr)
```

SEE ALSO

ld(1), fseek(3S), a.out(5), filehdr(5), scnhdr(5), *CONVEX Compiler Utilities User's Guides*

NAME

sendmail.cf – sendmail configuration file

SYNOPSIS

/usr/lib/sendmail.cf

DESCRIPTION

Sendmail reads the configuration file or a frozen version of it at the start of execution. The configuration file defines the known mailers, the settings of various options, and how sendmail will parse addresses and rewrite message headers. Although the configuration file is complex, a new configuration can be built by adjusting an existing off-the-shelf configuration (see m4 Configurations in Table of Contents). The assumes you can use one of the existing configurations and that the standard installation parameters are acceptable. CONVEX supports the /usr/lib/sendmail.cf configuration file as released and provides a configuration directory where you can generate sendmail.cf files using the m4 macro language.

Table of Contents

Before You Start Configuring

Before you configure mail, decide how you want to handle the routing of mail and the maintainence of the aliases file.

You can use several approaches. Depending on site-specific administration, department organization, and various other influences, the design of the mail configuration will vary. Some guidelines follow:

1. Each host can directly deliver mail to any other host.

2. A host acts as the relay. All mail is first sent to this mail host, which delivers the mail.

3. Some variation between the first 2 situations. The first two configurations can be combined in various ways to generate a design that suites your site requirements.

If you want host transparency, where each host can mail directly to hosts within the same domain with only the user name given for an address, the aliases file must be kept up-to-date on each system.

A brief discussion of the m4 configurations is provided followed by the syntax and semantics of the sendmail.cf file.

m4 Configurations

The m4 configurations are a set of prototype definitions for different sendmail configurations. For example, there is a *smtponly*, and *uucponly* prototype. These prototypes are used to generate a working sendmail.cf file.

To build a sendmail.cf file, define m4 options in a <*filename*>.mc file. Then run make <*filename*>.cf to generate a new sendmail.cf file. The required steps are described in the m4 configuration directories.

All directories and filenames that refer to m4 configurations in this man page are located in the root of the *sendmail* configuration subtree, */usr/lib/conf/sendmail.*

CONVEX has added a new directory under the sendmail directory named "convex"; the full path is /usr/lib/conf/sendmail/convex. This directory contains the basic m4 files from the Berkeley directories with additional rules and comments for CONVEX systems. The COVUEnet mailer and related rules and the NIS option have been included.

To use the m4 configuration files, change directories to the /usr/lib/conf/sendmail directory, and read the README file.

NOTE: This directory is installed only if the SOURCE option is selected during a ConvexOS AND UTILITIES installation.

SYNTAX AND SEMANTICS

The configuration file is organized as a series of lines, beginning with a single character label defining the semantics for the rest of the line. The possible lines are:

Label (Token)	Description
<sp>	
<tab>	Lines beginning with a space or a tab are continuation lines.
#	Blank lines and lines beginning with a pound sign (#) are ignored.
C*x word*	Create or add word into class x.
D*x val*	Define macro x to have value *val*.
F*x file [format]*	Read file for lines to add into class x. Use the scanf string *format* or use %s if not specified. The *format* must only produce one string from each line of the input file.
H*name: value*	Define header with field-name name and value as specified; this will be expanded immediately before use.
M*name* **arg**=*val*...	Define mailer. name is the internal name of the mailer. Args specify mailer parameters.
O*x value*	Set option x to *value*.
P*name*=*value*	Set precedence name to *value*.
R*lhs*<*tab*>*rhs*<*tab*>*comments*	Rewrite addresses that match left-hand side (lhs) to be the form specified by the right-hand side (rhs).
S*n*	Use ruleset (rewriting) set *n*.
T*user*	Create list of trusted users.

CLASSES - C and F lines

Classes are named with a single character. These can be selected from the entire ASCII set, but user-defined macros should be selected from the set of uppercase letters only. Lower case letters are used internally.

Classes of words can be defined to match the left-hand side of rewriting rules, where a word is a sequence of characters that do not contain characters in the $o macro, which contains the address operators (.:%@!^=/[]).

Classes are interpolated only on the LHS of a rule using the construct $=*x* *and* $~*x*, where *x* is the name of the class to be interpolated.

For example, a class of all local names for this site might be created so that attempts to send mail to oneself can be identified. These can either be defined directly in the configuration file or read in from another file. classes can be given names from the set of uppercase letters. Lowercase letters and special characters are reserved for internal sendmail use.

Special Classes

The class $=w is set to be the set of all names this host is known by. Class w can be used to match local host names. The first name should be the canonical form (the official network hostname in string format as opposed to internet address format). It is usually set by sendmail via the gethostent(2) call.

MACROS - D lines

Macros are named with a single character. These can be selected from the entire ASCII set, but user-defined macros should be selected from the set of uppercase letters only. Lowercase letters are reserved to have special semantics used to pass information in or out of sendmail, and some special characters are reserved to provide conditionals.

Macros are interpolated using the construct $\$x$, where x is the name of the macro to be interpolated.

Conditionally Referencing Macros
Conditionals can be specified by using the syntax:

$?x *text1* $| *text2* $.

This interpolates *text1* if the macro **$x** is set, and *text2* if **$x** is not set. The else ($|) clause can be omitted. The endif (**$.**) clause provides a method of ending the conditional.

Required Macros
These macros are defined internally in sendmail but are not set, therefore the macros must be defined in the sendmail.cf file. You should only have to define or change the j macro. It should be set to the official (canonical) name of the host, as returned by gethostbyname(2). Usually macro j is set to macro w.

The following macros *must* be defined, in the sendmail.cf file, to transmit information into *sendmail:*

Macro	Description
e	The SMTP entry message that is printed out by the SMTP server when a SMTP client connects to it. The first word must be the $j macro.
j	The official domain name for this site. The $j macro should be in RFC 821 format.
l	The format of the UNIX "From " line. This is not the same as the RFC standard format From: line. Can be considered constant except under unusual circumstances.
n	The name of the daemon (for error messages). Can be considered constant except under unusual circumstances.
o	The set of operators (.:%@!^=/[]) in addresses. These operators will separate addresses into atoms when being parsed. For example, if @ were in the $o macro, then the input a@b would be scanned as three tokens (atoms): a, @, and b.
q	Default format of sender address. Specifies how an address should appear in a message when it is not modified.

The default for these definitions in the CONVEX sendmail.cf file are as follows:

```
De$j Sendmail $v ready at $b
DnMAILER-DAEMON
DlFrom $g  $d
Do.:%@!^=/
Dq$g$?x ($x)$.
Dj$H.$D
```

An alternative for the **$q** macro is $?x$x $.<$g>. These correspond to the following two formats:

```
dduck@gatekeeper (Doolittle Duck)
Doolittle Duck <dduck@gatekeeper>
```

Predefined Macros

Some macros are defined by *sendmail* for interpolation into argv's for mailers or for other contexts. These macros are:

Macro	Description
a	The origination date in RFC 822 format (Tue, 16 Jul 91 17:10:21 -0500). Retrieved from the Date: header. If no Date: header then it is set to $b.
b	The current date in RFC 822 format(Tue, 16 Jul 91 17:10:21 -0500). Used on the Received: and Received-Date: headers and on the SMTP login message ($e).
c	The hop count set by the -h command line option or by counting the number of Received: headers. Informs how many times this message was processed by a mail transfer agent.
d	The date in UNIX (ctime) format.
f	The sender (from) address. For example, user_me.
g	The sender address *relative* to the recipient. For example, user_me@myhost.
h	The recipient host set from $@ of the triple.
i	The queue id. Usually used on the Received: header line (timestamp line). Helpful for tracking messages.
p	Sendmail's pid (decimal).
r	Protocol used to talk to sender.
s	Sender's host name.
t	A numeric representation of the current time.
u	The recipient user set from $: of the triple.
v	The version number of sendmail binary. Usually included in the Received: header. Helpful for tracking (known) mailing problems caused by sendmail.
w	The canonical hostname for this site.
x	The full name of the sender.
y	The tty ID of terminal.
z	The home directory of the recipient if on local system.

HEADERS - H lines

Special Headers

Several header lines have special interpretations defined by the configuration file. Others have interpretations built into *sendmail* that cannot be changed without changing the code. These built-ins are described here.

Return-Receipt-To:

If this header is sent, a message containing the message headers is sent to any specified addresses when the final delivery is complete, that is, when successfully delivered to a mailer with the l flag (local delivery) set in the mailer description. This is often used for testing or certifying that the mail was received.

Errors-To:

If errors occur anywhere during processing, this header causes error messages to go to the listed addresses rather than to the sender. The Errors-To: header is intended for use in mailing lists.

Apparently-To:

If a message arrives with no recipients listed in the message (in a To:, Cc:, or Bcc: header line) then *sendmail* will add an Apparently-To: header line for all recipients it is aware of from the message envelope. This is not put in as a standard recipient line to warn any recipients that the

list may not be complete.

At least one recipient line is required under RFC 822.

MAILERS - M lines

To add an outgoing mailer to your mail system, you have to define the characteristics of the mailer.

Each mailer must have an internal name. This can be arbitrary, except that the names *local* and *prog* must be defined.

Field	Description
P	The pathname of the mailer. If this mailer provides access through an IPC connection, use the string [IPC] instead.
F	Defines the mailer flags. See the Mailer Flags section.
S	Defines the per-mailer rewriting sets to be applied to sender addresses. Applied after ruleset 1 has been performed but before ruleset 4. Please see the RULESET section for information on the S field.
R	Defines per-mailer rewriting sets to be applied to recipient addresses. Applied after ruleset 2 has been performed but before ruleset 4. Please see the RULESET section for information on the R field.
A	Defines an argv template which is the command line arguments for the specified mailers actual program that sendmail executes. It may have embedded spaces. If there is no argv with a **$u** macro in it, *sendmail* will speak SMTP to the mailer. If the pathname for this mailer is "[IPC]", the argv should be IPC $h [*port*] where *port* is the optional port number to connect to.
E	Defines the string to use as an end-of-line indication. A string containing only newline is the default. The usual backslash escapes (\r, \n, \f, \b) may be used.
M	Defines the maximum size of a message allowed by this mailer.

Mailer Flags - F field

A number of flags can be associated with each mailer, that are identified by a letter of the alphabet. Many are assigned semantics internally. Mailer flags tell sendmail what the delivery agent expects for command line arguments and options, and for envelope and header lines. Other flags can be used to conditionally assign headers to messages destined for particular mailers.

The following flags can be set in the mailer description.

Flag	Description
f	Mailer expects a **–f** *from* flag, but only if this is a network forward operation (that is, the mailer will give an error if the executing user is not defined as a Trusted user). The flag is passed to the mailer if it was passed to *sendmail* by the mail agent. This allows you to define the proper flag required for certain mailers. If the mailer accepts either -f or -r, you can specify -f $g in the argv template.

r Same as **f**, but sends a **–r** flag.

S Do not reset the userid before calling the mailer. This would be used in a secure environment where *sendmail* runs as root. This could be used to avoid forged addresses. The S flag is suppressed if given from an unsafe environment (e.g, a user's mail.cf file). If the mailer is called as **root** the **S** flag should be given; this will not reset the userid before calling the mailer. *Sendmail* must be running setuid to root for this to work.

n Do not insert a UNIX-style From line on the front of the message.

l This mailer is local (that is, final delivery is performed rather than taking another network hop).

s Strip quote characters (backslashes and double quote marks) off of the address before calling the mailer.

m This mailer can send to multiple users on the same host in one transaction. When a *$u* macro occurs in the *argv* part of the mailer definition, that field will be repeated necessary for all qualifying users.

F Mailer expects a From: header line.

D Mailer expects a Date: header line.

M Mailer expects a Message-Id: header line.

x Mailer expects a Full-Name: header line.

P Mailer expects a Return-Path: line.

u Uppercase should be preserved in user names.

h Uppercase should be preserved in host names.

A Mailer is Arpanet-compatible; all appropriate modes should be set.

U Mailer expects UNIX-style From lines with the UUCP-style "remote from <host>" on the end.

e Mailer is expensive to connect to, so defer connection until a queue daemon runs. The **c** configuration option must be set for this to be effective.

X Mailer uses the hidden dot algorithm as specified in RFC 821; any line beginning with a dot will have an extra dot prepended (to be stripped at the other end). This ensures that lines in the message containing a dot will not terminate the message prematurely.

L Limit the line lengths as specified in RFC 821.

P Use the Return-path in the SMTP MAIL FROM: command rather than just the return address; although this is required in RFC 821, many hosts do not process return paths properly.

I Mailer connects to another *sendmail*; therefore, it can use special protocol features. This option is not required. If this option is omitted the transmission will still operate successfully, although perhaps not as efficiently as possible.

C If mail is *received* from a mailer with this flag set, any addresses in the header that do not have a domain specification, *@host.domain*, will have the domain from the sender address attached by ruleset 3. This allows mail with headers of the form

From: me@ myhost
To: user_a@ remote_a, user_b

to be rewritten as

From: me@ myhost
To: user_a@ remote_a, user_b@ myhost

automatically.

This situation occurs *if and only if* the **C** flag is defined in the mailer corresponding to me@ myhost.

E Escape lines beginning with From in the message with a greater than sign (>).

Special Mailers

The following sections discuss special reserved mailers used by sendmail.

error **mailer**

The mailer with the special name error can be used to generate a user error. The (optional) host field is a numeric exit status to be returned, and the user field is a message to be printed. For example, the entry

$#error$:Host unknown in this domain

on the RHS of a rule causes the specified error to be generated if the LHS matches. This mailer is only functional in ruleset zero.

local **mailer**

The mailer with the reserved name *local* defines the mailer delivery agent for the local system. If the mailer portion of the triple, {mailer, host, user}, resolves to the local system then the *local* mailer is used.

prog **mailer**

The mailer with the reserved name *prog* defines the mailer delivery agent (program) for the local system when a pipe (|) is specified at the beginning of the resolved address.

OPTIONS - O lines

A number of options can be set from a configuration file. Options are represented by single characters. The syntax of an option line is

O*o value*

This sets option *o* to be *value*. Depending on the option, *value* may be a string, an integer, a boolean (with legal values "t ," "T ," "f ," or "F ;" the default is TRUE), or a time interval.

The following options may be set using the **–o** flag on the command line or the **O** line in the configuration file. Many of them cannot be specified unless the invoking user is trusted.

Option	Description
A*file*	Use *file* as the alias file. If no file is specified, use *aliases* in the current directory.
a*N*	If set, wait up to *N* minutes for an @:@ entry to exist in the alias database before starting up. If it does not appear in *N* minutes, rebuild the database (if the **D** option is also set), or issue a warning.
B*c*	Set the blank substitution character to *c*. Unquoted spaces in addresses are replaced by this character.
c	If an outgoing mailer is marked as being expensive, do not connect immediately. This requires a sendmail daemon performing a queue run to actually send the mail.
d*x*	Deliver in mode *x*. Legal modes are: i Deliver interactively (synchronously) b Deliver in background (asynchronously) q Just queue the message (deliver during queue run)
D	If set, rebuild the alias database if necessary and possible. If this option is not set, *sendmail* will never rebuild the alias database unless explicitly requested using **–bi**.
e*x*	Dispose of errors using mode *x*. The values for *x* are: p Print error messages (default) q No messages, just give exit status m Mail back errors w Write back errors (mail if user not logged in) e Mail back errors and always give zero exit status
f	Save UNIX-style "From " lines at the front of headers. Normally they are assumed redundant and discarded.
g*n*	Set the default group id for mailers to run in to *n*.
H*file*	Specify the help file for SMTP.

I Insist that the BIND name server be running to resolve host names. If this is not set and the name server is not running, the */etc/hosts* file is considered complete. In general, you do want to set this option if your */etc/hosts* file does not include all hosts known to you or if you are using the MX (mail forwarding) feature of the BIND name server. The name server is still consulted even if this option is not set, but *sendmail* will resort to reading */etc/hosts* if the name server is not available. Thus, you should *never* set this option if you do not run the name server.

i Ignore dots in incoming messages.

L*n* Set the default log level to *n*. See the "Log Levels" section that follows.

M*x value* Set the macro *x* to *value*. This is intended only for use from the command line.

m This forces the message to be sent to the sender even if the sender's address is part of an alias expansion. Send to me too, even if I am in an alias expansion.

n Causes sendmail to attempt to validate the right-hand side of each alias when the aliases file is rebuilt.

o Assume that the headers may be in old format, that is spaces delimit names. This actually turns on an adaptive algorithm: if any recipient address contains a comma, parenthesis, or angle bracket, it will be assumed that commas already exist. If this flag is not on, only commas delimit names. Headers are always output with commas between the names.

Q*dir* Use the named *dir* as the queue directory.

q*factor* Use *factor* as the multiplier in the map function to decide when to queue jobs rather than run them. This value is divided by the difference between the current load average and the load average limit (**x** flag) to determine the maximum message priority that will be sent. Defaults to 10000.

r*time* Timeout reads after *time* interval.

S*file* Log statistics in the named *file*.

s Be super-safe when running things, that is, always instantiate the queue file, even if you are going to attempt immediate delivery. *Sendmail* always instantiates the queue file before returning control to the client under any circumstances.

T*time* Set the queue timeout to *time*. After this interval, messages that have not been successfully sent are returned to the sender.

t*S,D* Set the local time zone name to *S* for standard time and *D* for daylight time. This is only used under Version 6.

u*n* Set the default userid for mailers to *n*. Mailers without the *S* flag in the mailer definition will run as this user.

v Run in verbose mode.

x*LA*	When the system load average exceeds *LA*, queue messages, that is, do not try to send them.
X*LA*	When the system load average exceeds *LA*, refuse incoming SMTP connections.
y*factor*	The indicated *factor* is added to the priority (thus *lowering* the priority of the job) for each recipient, that is, this value penalizes jobs with large numbers of recipients.
Y	If set, deliver each job that is run from the queue in a separate process. Use this option if you are short of memory, since the default tends to consume considerable amounts of memory while the queue is being processed.
z*factor*	The indicated *factor* is multiplied by the message class (determined by the Precedence: field in the user header and the **P** lines in the configuration file) and subtracted from the priority. Thus, messages with a higher Priority: are favored.
Z*factor*	The *factor* is added to the priority every time a job is processed. Thus, each time a job is processed, its priority is decreased by the indicated value. In most environments this is positive, since hosts that are down are usually down for a long time.

Log Levels

One of the following log levels can be set in the sendmail.cf file by using the L option. The errors are displayed if the log level is greater than or equal to the specified number. Refer to the syslog.conf man page for setting up system logging.

Level	Description
0	No logging.

1	Major problems only.
	- LOG_CRIT, "<NOQUEUE|eid>: SYSERR: <error.msg.text>",
	- LOG_NOTICE, "collect: unexpected close on connection from <address>: <hostame>"
	- LOG_ALERT, "cannot get connection" (possibly another sendmail (-bd) daemon running).
	if (realname == from && RealHostName != NULL)
	- LOG_NOTICE, "from=<from.address> unparseable, received from <host>"
	else
	- LOG_NOTICE, "Unparseable username <name> wants from=<from.address>"
	- LOG_ALERT, "savemail: HELP!!!!"
	Unable to save message to dead.letter.

2	Message collections and failed deliveries. - LOG_INFO, "<id>: to=<to>, delay=<intvl>, stat=error_msg" - LOG_INFO, "<eid>: from=<addr>, size=<msg.size>, class=<eclass>, received from <name>0,
3	Successful deliveries. - LOG_INFO, "<id>: to=<to>, delay=<intvl>, stat=error_msg"
4	Messages being deferred (unable to connect to host, etc.).
5	Normal message queue ups. - LOG_INFO, "<id>: to=<to>, delay=<intvl>, stat=queued"
6	Unusual but benign incidents, such as trying to process a locked queue file. - LOG_NOTICE, "alias database [<auto>]rebuilt by <username>"
7	Information related to alias database. - LOG_INFO, "rebuilding alias database" - LOG_INFO, "alias database out of date"
8	Additional information related to alias database. - LOG_INFO, "<naliases> aliases, longest <longest> bytes, <bytes> bytes total"
9	Log internal queue id to external message id mappings. This is useful for tracing a message as it travels between several hosts. - LOG_INFO, "<eid>: message-id=<mid>" (conditional on headers and other flags and options)

11	Messages about envelope handing - LOG_DEBUG, "dropenvelope, id=<eid>, flags=<eflgs>, pid=<getpid()>"
12	Several messages that are of interest when debugging. - LOG_DEBUG, "connected, pid=<getpid()>" - LOG_DEBUG, "finis, pid=<getpid()>" - LOG_DEBUG, "in background, pid=<getpid()>" - LOG_DEBUG, "runqueue <Q-DIR>, pid=<PID>" - LOG_DEBUG, "<eid>: dowork, pid=<PID>"
16	Verbose information regarding the queue. - LOG_DEBUG, "<eid>: queueup, qf=<QF>, df=<DF>"
17	Additional information on queue files. LOG_DEBUG, "<eid>: assigned id"
20	Verbose information about file opening or closing and other miscellaneous information. - LOG_DEBUG, "<eid>: openx[<(no)>]" - LOG_DEBUG, "<eid>: unlock"

Load Factor Examples

The following values are site configuration and job-mix dependent, therefore they are only recommendations. The priority of mail delivery is also site dependent.

Hardware	Mail is low priority		Mail is high priority	
	x	X	x	X
C1	4	8	8	12
C210/C3210	6	10	10	14
C220/C3220	8	12	12	16
C230/C3230	10	14	14	18
C240/C3240	12	16	16	20
C3410	6	10	10	14
C3420	8	12	12	16
C3430	10	14	14	18
C3440	11	16	16	20
C3810	10	18	16	22
C3820	12	20	18	24
C3830	14	22	20	26
C3840	16	24	22	28
C3850	18	26	24	30
C3860	20	28	26	32
C3870	22	30	28	34
C3880	24	32	30	38

RULESETS - S lines

There are several sets of rewriting rules. Some of the rewriting sets are used internally and must have specific semantics. Other rewriting sets do not have specifically assigned semantics, and can be referenced by the mailer definitions or by other rewriting sets.

The S token sets the current ruleset being collected to *n*. If you begin a ruleset more than once it deletes the old definition.

Semantics

A ruleset operates like a function. When sendmail interprets $>n it jumps to ruleset Sn. Sendmail calls rules 0 through 4 internally. Mailer definition lines define rulesets to call to manipulate the address for the sender(S=n) and/or recipient(R=n).

Rules are applied to addresses in the following way:

Rule	Description
1	Apply the rule.
2	If it matches (LHS), rewrite it (RHS) and apply the rule again. If it does not match, try the next rule.
3	If RHS begins with $:, then after rewriting, apply the NEXT rule. This prevents infinite loops when calling ($>) rulesets.
4	If RHS begins with $@, then after rewriting, exit ruleset.

There are five rewriting sets that have specific semantics: 0, 1, 2, 3, and 4. These rulesets and their functions are shown in the following diagrams.

```
-------------------------------------------------------------------------
Recipient Address Rewriting to pass to delivery agent:

  To: Address--> 3 -> 0 -> 4 -->Resolved Address {mailer,host,user}

-------------------------------------------------------------------------
Header Address Rewriting for Sender Address:

  From: Address--> 3 -> D -> 1 -> S= -> 4 --> From: Address

-------------------------------------------------------------------------
Header Address Rewriting for Recipient Address:

  To:/Cc: Address--> 3 -> D -> 2 -> R= -> 4 --> To:/Cc: Address

-------------------------------------------------------------------------
D = Sender domain addition
S = Mailer-specific sender rewriting
R = Mailer-specific recipient rewriting
-------------------------------------------------------------------------
```

Ruleset 3 turns the address into canonical form. The syntax is:

local-part@host-domain-spec

If no @ sign is specified, then the host-domain-spec *may* be appended from the sender address (if the **C** flag is set in the mailer definition corresponding to the *sending* mailer).

NOTE: Ruleset 3 is applied by sendmail before doing anything with any address.

Ruleset 0 is applied after ruleset 3 to addresses that are going to actually specify recipients. It must resolve to a {*mailer, host, user*} triple. The *mailer* must be defined in the mailer definitions from the configuration file. The *host* is defined into the **$h** macro for use in the argv expansion of the specified mailer.

Rulesets 1 and 2 are applied to all sender and recipient addresses respectively. They are applied before any specification, the R= or S= fields, in the mailer definition. They must never resolve to a triple.

Ruleset 4 is applied to all addresses in the message. It is typically used to translate the address from internal form to external form.

S Mailer Flag Ruleset

The ruleset defined by the Mailer S field provides the ability to rewrite the sender address for the specified mailer.

Before the address is sent to this ruleset it has already had the domain appended (if requested) and ruleset 1 applied. After this ruleset is applied to the address it is sent to ruleset 4.

For example, a header of the form

From: me

might be changed to

From: me@myhost

or

From: myhost!me

depending on the domain to which it is being sent.

R Mailer Flag Ruleset

The ruleset defined by the Mailer R field provides the ability to rewrite the recipient address for the specified mailer. Before the address is sent to this ruleset it has already had the domain appended (if requested) and ruleset 2 applied. After this ruleset is applied to the address it will be sent to ruleset 4.

RULES - R lines

The core of address parsing is the rewriting rules. These are an ordered production system. *Sendmail* scans through a set of rewriting rules looking for a match on the left-hand side (LHS) of the rule. When a rule matches, the address is replaced by the right-hand side (RHS) of the rule. The syntax for R lines is

R*lhs*<*tab*>*rhs*<*tab*>*comments*

The fields must be separated by at least one tab character; there can be embedded spaces in the fields. The *lhs* is a pattern that is applied to the input. If it matches, the input is rewritten to the *rhs*. The *comments* are ignored.

Substitution occurs in the order described, that is, parameters from the LHS are substituted, host names are canonicalized, subroutines are called, and finally $#, $@, and $: are processed.

The Left Hand Side

The left-hand side of a rewriting rule contains a pattern. Normal words are simply matched directly. Metasyntax is introduced by a dollar sign. The metasymbols are:

LHS	Debug Symbol(-bt)	Description
$*	^P	Match 0 or more tokens
$+	^Q	Match 1 or more tokens
$-	^R	Match exactly 1 token
$=*x*	^S*x*	Match any token in class *x*

$~x	^Tx	Match any token not in class x

If any of these match, they are assigned to the symbol $n for replacement on the right-hand side, where n is the index in the LHS. For example, if the LHS

$-:$+

is applied to the input

domain:user

the rule will match, and the values passed to the RHS will be:

$1 domain
$2 user

Note that literals specified in the LHS are not passed to the RHS.

The Right Hand Side

When the left-hand side of a rewriting rule matches, the input is deleted and replaced by the right-hand side. Tokens are copied directly from the RHS unless they begin with a dollar sign. Metasymbols are listed in the following table.

RHS	Debug Symbol(-bt)	Description
$*n*	^U*n*	Substitute indefinite token *n* from LHS.
$[*name*$]	^]*hostname*	Canonicalize *name* (call gethostent(3)). $[vangogh$] might become vangogh.berkeley.edu. $[[128.32.130.2]$] would become vangogh.berkeley.edu.
$>*n*	^Y*n*	Call ruleset *n*. The remainder of line is substituted as usual, then passed to rule set *n*. Final value of rule *n* is then substituted for this rule.
$#*mailer*	^V*mailer*	Resolve to triple, {*mailer, host, user*}. Causes ruleset 0 and the resolution process of the address to terminate immediately. Use only in rule 0 as $#*mailer*#@*host*$:*user*. Sendmail processes the message through the specified mailer and ends. The *mailer* and *host* must be single words that contain no spaces or tabs, but the *user* can be mutliple words.
$@*host*	^W*host*	Specify resolved *host*, used in ruleset 0.
$@...$>	^W	RHS evaluation control which causes a ruleset to exit and returns the remainder of RHS as the value. The "..." are other RHS metasymbols.
$:*user*	^X*user*	Specify resolved *user*, used in ruleset 0.
$:...$>	^X	RHS evaluation control which causes the rule to terminate immediately, but the ruleset to continue. This allows you to call a ruleset only once and prevents infinite loops if the LHS matches anything. The "..." are other RHS metasymbols.

The $n syntax substitutes the corresponding value from a $+, $–, $*, $= or $~ match on the LHS. It may be used anywhere in the RHS.

DEBUGGING

A large number of debug flags are built into *sendmail*. Each debug flag has a number and a level. The higher the level, the more the information that is printed. The convention is that levels greater than nine are "absurd,"; they print out so much information that you would not normally want to see them except when debugging that particular piece of code. Use the **–d** option to set debug flags. The syntax is

```
debug-flag:     –d debug-list
debug-list:     debug-option [ , debug-option ]
debug-option:   debug-range [ . debug-level ]
debug-range:    integer | integer - integer
debug-level:    integer
```

where spaces are for reading ease only. For example,

Flag.Level	Description
–d12	Set flag 12 to level 1
–d12.3	Set flag 12 to level 3
–d3-17	Set flags 3 through 17 to level 1
–d3-17.4	Set flags 3 through 17 to level 4

Debugging Levels

The following table provides an explanation of what is displayed when certain flags and levels of debugging are specified on the command line. The table contains one or more level lines for each flag line. The flag line contains the names of the source code modules (which are useful if you have a full source code license or have the sendmail source code from Berkeley). The level lines contain the flag and level with a statement about what is displayed if that flag and level are specified.

Flag	Flag.Level	Source code module Description
0		main.c, recipient.c, util.c
	0.1	Don't fork in daemon mode, permit direct mailings to files, programs, and :includes:'s.
	0.4	Print names for this host
	0.15	Print configuration table.
	0.44	Prints addresses of elements - Printav( ).
1		main.c, envelope.c
	1.1	Prints From person - main( ).
2		main.c
	2.1	Print exit status and envelope flags - finis( ).
5		clock.c
	5.4	Print calls to tick.
	5.5	Print set/clrevent args.
	5.6	Prints event queue on each tick.
6		savemail.c

	6.1	Print savemail( ) error mode and return-to-sender information.
	6.5	Trace states in savemail( ) state machine.
7		queue.c
	7.1	Print info on envelope assigned to queue file.
	7.2	Print selected queue file name.
8		domain.c
	8.1	Print various information regarding resolver operations
	8.8	Set RES_DEBUG.
10		deliver.c
	10.1	Print various address information.
11		deliver.c
	11.1	Print openmailer( ) args.
13		deliver.c
	13.1	Print all addresses being sent to - sendall( ).
	13.3	Prints each addr in loop looking for failures - sendall( ).
	13.4	Follows above, printing who gets the error - sendall( ).
14		headers.c
	14.2	Print headers being converted to comma separated list (commaized)
15		daemon.c
	15.1	Print port and socket numbers in getrequests( )
	15.2	Getrequests -- note forking/returning
	15.15	Activate network debugging on daemon socket
16		daemon.c
	16.1	Print host, addr, socket - makeconnection( ).
	16.15	Print network debugging on daemon socket.
18		usersmtp.c
	18.1	Note openmailer failure, note entry to reply, print smtpmessage( ) args.
20		parseaddr.c
	20.1	Print parseaddr( ) arg and result.
21		parseaddr.c
	21.2	Print rewrite( ) arg and result.
	21.3	Note ruleset subroutine call.
	21.4	Rewritten as ...
	21.10	Note rule failure.
	21.12	Note rule attempt and success.
	21.15	Print replacement string in hex chars.
	21.35	Print elements in pattern and subject.
	21.81	All of the above.
25		recipient.c
	25.1	Print sendto( ) arguments.

26		recipient.c
	26.1	Print recipient in recipient( ) and duplicate suppression
27		alias.c
	27.1	Print arg to alias( ), print info about alias, note failure to open alias file, print arg to forward( ).
30		collect.c
	30.1	Note EOH.
	30.2	Print eatfrom arg.
	30.3	Note addition of Apparently-To.
31		headers.c
	31.6	Print chompheader argument.
32		headers.c
	32.1	Print collected header.
33		headers.c
	33.1	Print crackaddr arg and return value.
35		macro.c
	35.9	Print define() args.
	35.24	Print expand() arg and return value.
36		stab.c
	36.5	Print stab args, sym found/not found, entered.
	36.9	Print hfunc value.
37		readcf.c
	37.1	Print info re option setting/values.
40		queue.c
	40.1	Note queue insertion and print queue contents.
	40.4	Show queue file contents.
41		queue.c
	41.2	Note open failure on cf file.
51		queue.c
	51.4	Don't unlink x file.
45		envelope.c
	45.1	Print setsender argument.
50		envelope.c
	50.1	Print dropenvelope argument.
52		main.c
	52.1	Print i/o fd's for tty disconnection.
	52.5	Don't disconnect.

The debug table was created by Bill Mitchell at The University of Arizona.

Using a Different Configuration File

An alternative configuration file can be specified using the **–C** flag, for example,

```
/usr/lib/sendmail –Ctest.cf
```

uses the configuration file *test.cf* instead of the default */usr/lib/sendmail.cf*. If the **–C** flag has no value it defaults to *sendmail.cf* in the current directory.

Testing the rewriting rules with the -bt flag

When you build a configuration table, you can test using the test mode of *sendmail*. For example, invoke *sendmail* with the command

```
sendmail –bt –Ctest.cf
```

to read the configuration file test.cf and enter test mode. In test mode, enter

```
rwset address
```

where *rwset* is the rewriting set to use and *address* is an address to apply the set to. Test mode displays the steps sendmail takes as it resolves the address. You can use a list of rwsets separated by commas for sequential application of rules to an input. Ruleset 3 is always applied first, so you do not need to specify it.

Usually you want to resolve the address to a triple first. Then the recipient address from the triple is run through the mailer's specified ruleset. The sender's address (your user name or whatever your mailer agent attaches to the From: line) is run through the mailer's specified ruleset.

The following table provides a cross reference between the -bt output field designators and their related triple field.

Designator	Represented Triple
^V	mailer
^W	host
^X	user

For example:

```
# /usr/lib/sendmail -bt -Ctest.cf
ADDRESS TEST MODE
Enter <ruleset> <address>
> 0 user@remote
rewrite: ruleset  3   input: "user" "@" "remote"
rewrite: ruleset  8   input: "user" "@" "remote"
rewrite: ruleset  8 returns: "user" "@" "remote"
rewrite: ruleset  6   input: "user" "<" "@" "remote" ">"
rewrite: ruleset  6 returns: "user" "<" "@" "remote" ">"
rewrite: ruleset  3 returns: "user" "<" "@" "remote" ">"
rewrite: ruleset  0   input: "user" "<" "@" "remote" ">"
rewrite: ruleset  3   input: "user" "@" "remote"
rewrite: ruleset  8   input: "user" "@" "remote"
rewrite: ruleset  8 returns: "user" "@" "remote"
rewrite: ruleset  6   input: "user" "<" "@" "remote" ">"
rewrite: ruleset  6 returns: "user" "<" "@" "remote" ">"
rewrite: ruleset  3 returns: "user" "<" "@" "remote" ">"
rewrite: ruleset  6   input: "user" "<" "@" "remote" ">"
rewrite: ruleset  6 returns: "user" "<" "@" "remote" ">"
rewrite: ruleset  0 returns: "^V" "smtp" "^W" "remote" "." "foo" "."
```

```
"com" "^X" "user" "<" "@" "remote" "." "foo" "." "com" ">"
> 2,27,4 user@remote.foo.com # from user (^X) above
rewrite: ruleset 27 returns: "user" "<" "@" "remote" "." "foo" "." "com" ">"
rewrite: ruleset  4  input: "user" "<" "@" "remote" "." "foo" "." "com" ">"
rewrite: ruleset  4 returns: "user" "@" "remote" "." "foo" "." "com"
> 1,17,4 user
rewrite: ruleset  3  input: "user"
rewrite: ruleset  8  input: "user"
rewrite: ruleset  8 returns: "user"
rewrite: ruleset  3 returns: "user"
rewrite: ruleset  2  input: "user"
rewrite: ruleset  2 returns: "user"
rewrite: ruleset 17  input: "user"
rewrite: ruleset  3  input: "user"
rewrite: ruleset  8  input: "user"
rewrite: ruleset  8 returns: "user"
rewrite: ruleset  3 returns: "user"
rewrite: ruleset 17 returns: "user" "<" "@" "local" "." "foo" "." "com" ">"
rewrite: ruleset  4  input: "user" "<" "@" "local" "." "foo" "." "com" ">"
rewrite: ruleset  4 returns: "user" "@" "local" "." "foo" "." "com"
> ^D
```

Changing the Values of Options

Options set in the configuration file can be overridden using the **–o** flag on the command line. For example

```
/usr/lib/sendmail –oT2m
```

sets the **T** (timeout) option to two minutes for this run only.

Checking if address is deliverable

To see if an address you wish to send mail to is deliverable, enter

```
/usr/lib/sendmail -bv -v -L10 -Csendmail.cf <address>.
```

For example

```
# /usr/lib/sendmail -bv -v -L10 -Ctest.cf auser
auser... aliased to auser@remote
auser@remote... deliverable
```

One of the problems with this testing is that if the name server is running, unknown and remote addresses will return as deliverable, although the unknown address may not exist and the remote site is down or unreachable.

Using Verbose Mail

The -v option on /usr/ucb/mail is useful to verify that the mail is delivered properly.

```
# /usr/ucb/mail -v auser
Subject: Test mail

This is just a test message. Please delete.

Cc:
auser... aliased to auser@remote
auser@remote... Connecting to remote.foo.com (smtp)...
220 remote.foo.com Sendmail 5.64/1.28 ready at Mon, 16 Sep 91 09:43:29
```

```
-0500
>>> HELO local.foo.com
250 remote.foo.com Hello local.foo.com, pleased to meet you
>>> MAIL From:<root@local.foo.com>
250 <root@local.foo.com>... Sender ok
>>> RCPT To:<auser@remote.foo.com>
250 <auser@remote.foo.com>... Recipient ok
>>> DATA
354 Enter mail, end with "." on a line by itself
>>> .
250 Ok
>>> QUIT
221 remote.foo.com closing connection
auser@remote... Sent
```

LIMITS

The following limits are set in the binary and are not configurable in the sendmail.cf file.

Name	Limit	Description
MAXLINE	1024	Maximum line length of any input line. If message lines exceed this length they will still be processed correctly; however, header lines, configuration file lines, alias lines, etc., must fit within this limit.
MAXNAME	256	Maximum length of any name, such as a host or a user name.
MAXFIELD	2500	Maximum total length of any header field, including continuation lines.
MAXPV	40	Maximum number of parameters to any mailer. This limits the number of recipients that can be passed in one transaction.
MAXHOP	17	When a message has been processed by sendmail more than MAXHOP times, sendmail assumes there has been an aliasing loop and rejects the message. This can be determined from the **-h** flag or by counting the number of trace fields (e.g, Received: lines) in the message header.
MAXATOM	100	Maximum number of atoms (tokens) in a single address. For example, the address user@host is three atoms.
MAXMAILERS	25	Maximum number of mailers that can be defined in the configuration file.
MAXRWSETS	30	Maximum number of rewriting sets that can be defined.
MAXPRIORITIES	25	Maximum number of values for the Precedence: field that can be defined (using the **P** line in sendmail.cf).
MAXTRUST	30	Maximum number of trusted users that can be defined (using the **T** option in sendmail.cf).
MAXUSERENVIRON	40	Maximum number of items in the user environment that will be passed to subordinate mailers.
MAXMXHOSTS	10	Maximum number of MX records for a particular host.
QUEUESIZE	600	Maximum number of entries that will be processed in a single queue run.

SEE ALSO

Managing ConvexOS: Configuration Guide, Setting Up Sendmail
Managing ConvexOS: Operations Guide, Managing Sendmail
sendmail(8), mqueue(5), mailq(8), aliases(5), newaliases(1).

FILES

/usr/lib/sendmail	The binary of *sendmail*.
/usr/bin/newaliases	A link to /usr/lib/sendmail; causes the alias database to be rebuilt. Running this program is equivalent to using the **–bi** flag with *sendmail*.
/usr/bin/mailq	Prints a listing of the mail queue. This program is equivalent to using the **–bp** flag to *sendmail*.
/usr/lib/sendmail.cf	The configuration file in textual form.
/usr/lib/sendmail.fc	The configuration file represented as a memory image.
/usr/lib/sendmail.hf	The SMTP help file.
/usr/lib/sendmail.st	A statistics file; is not required.
/usr/lib/aliases	The textual version of the alias file.
/usr/lib/aliases.{pag,dir}	The alias file in *dbm*(3) format.
/usr/spool/mqueue	The directory in which the mail queue and temporary files reside.
/usr/spool/mqueue/qf*	Control (queue) files for messages.
/usr/spool/mqueue/df*	Data files
/usr/spool/mqueue/lf*	Lock files
/usr/spool/mqueue/tf*	Temporary versions of the qf files, used during queue file rebuild.
/usr/spool/mqueue/nf*	A file used when creating a unique id.
/usr/spool/mqueue/xf*	A transcript of the current session. This is a summary of the support files that *sendmail* creates or generates.
/usr/lib/conf/sendmail	Directory containing m4 control files to generate sendmail.cf files.
/usr/lib/conf/sendmail/bsd	Directory for building BSD sendmail.cf files.
/usr/lib/conf/sendmail/convex	Directory for building CONVEX sendmail.cf files.
convex/bin	Directory containing unsupported sendmail related maintenance commands.
convex/doc	Directory containing sendmail related documents.
convex/doc/07.sendmailop	*ConvexOS Tutorial Papers, Sendmail Installation and Operation Guide* in troff format.
convex/doc/16.sendmail	*ConvexOS Tutorial Papers, SENDMAIL - An Internet Mail Router* in troff format.
convex/doc/rfc819.lpr	RFC 819: *The Domain Naming Convention for Internet User Applications* in lpr format.
convex/doc/rfc821.lpr	RFC 821: *SIMPLE MAIL TRANSFER PROTOCOL* in lpr format.
convex/doc/rfc822.lpr	RFC 822: *STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES* in lpr format.

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 the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Service names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/services

SEE ALSO

getservent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NOTES

services is an optional product; for more information, contact your CONVEX sales representative.

NAME

/etc/shells – define the standard user shells

DESCRIPTION

/etc/shells defines the set of supported shell programs available to all users. The format of the file is one shell per line; shells must be specified using full path names. Any line that starts with a '#' is a comment line.

The information in */etc/shells* can be uniformly accessed using the *getusershell*(3) interface.

FILES

/etc/shells

SEE ALSO

csh(1), chsh(1), sh(1), getusershell(3)

NAME

stab – symbol table types

SYNOPSIS

#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 old debugger *sdb* and the Berkeley Pascal compiler *pc*(1). 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*(5), 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 old symbolic debugger, *sdb*, uses the following n_type values:

```
#define N_GSYM  0x20 /* global symbol: name,,0,type,0 */
#define N_FNAME 0x22 /* procedure name: name,,0 */
#define N_FUN   0x24 /* procedure: name,,0,linenumber,address */
```

```
#define N_STSYM 0x26 /* static symbol: name,,0,type,address */
#define N_LCSYM 0x28 /* .lcomm symbol: name,,0,type,address */
#define N_RSYM  0x40 /* register sym: name,,0,type,register */
#define N_SLINE 0x44 /* src line: 0,,0,linenumber,address */
#define N_SSYM  0x60 /* structure elt: name,,0,type,struct_offset */
#define N_SO    0x64 /* source file name: name,,0,0,address */
#define N_LSYM  0x80 /* local sym: name,,0,type,offset */
#define N_SOL   0x84 /* #included file name: name,,0,0,address */
#define N_PSYM  0xa0 /* parameter: name,,0,type,offset */
#define N_ENTRY 0xa4 /* alternate entry: name,linenumber,address */
#define N_LBRAC 0xc0 /* left bracket: 0,,0,nesting level,address */
#define N_RBRAC 0xe0 /* right bracket: 0,,0,nesting level,address */
#define N_BCOMM 0xe2 /* begin common: name,, */
#define N_ECOMM 0xe4 /* end common: name,, */
#define N_ECOML 0xe8 /* end common (local name): ,,address */
#define N_LENG  0xfe /* second stab entry with length information */
```

where the comments give *sdb* conventional use for *.stab*s and the n_name, n_other, n_desc, and n_value fields of the given n_type. *sdb* 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

The Berkeley Pascal compiler, *pc*(1), uses the following n_type value:

```
#define N_PC 0x30 /* global pascal symbol: name,,0,subtype,line */
```

and uses the following subtypes to do type checking across separately compiled files:

1	source file name
2	included file name
3	global label
4	global constant
5	global type
6	global variable
7	global function
8	global procedure
9	external function
10	external procedure
11	library variable
12	library routine

SEE ALSO

as(1), ld(1), csd(1), a.out(5)

BUGS

sdb assumes that a symbol of type N_GSYM with name *name* is located at address _*name*.

More basic types are needed.

NAME

/usr/adm/stat/stats* - system statistics

DESCRIPTION

Each file in the */usr/adm/stat* directory corresponds to a date. The statistics gathered by */etc/stat* for that date reside in the corresponding file. */etc/stat* awakens at regular intervals and samples the system usage, disk activity, load average, etc. It then appends this information to the current file in the */usr/adm/stat* directory.

Each record is 50 bytes long and contains the following fields:

Offset	Name	Size	Coefficient
0	Time record was written	4	1
4	Buffered i/o latency(ms)	1	1
5	Buffered i/o Mb/s	2	1000
7	Disk average latency(ms)	1	1
8	Disk Mb/s	2	1000
10	Stripe Mb/s	1	1000
11	Network packets in/sec	1	1
12	Network packets out/sec	1	1
13	Network input errors/sec	1	1
14	Network output errors/sec	1	1
15	Network collisions/sec	2	1
17	Cpu user time %	1	1
18	Cpu niced time %	1	1
19	Cpu system time %	1	1
20	Cpu idle time %	1	1
21	Virtual memory available	2	1
23	Free virtual memory	2	1
25	Page ins/sec	1	1
26	Page outs/sec	1	1
27	Device interrupts/sec	2	1
29	System calls/sec	2	1
31	Load average	2	10
33	Disk request size	1	1
35	Real memory available	2	1
37	System users	1	1
38	Page reclaims/sec	1	1
39	Page faults/sec	2	1
41	Page hits %	1	1
42	Total # processes	2	1
44	Pages scanned by clock/sec	1	1
45	Context switches/sec	2	1
47	Buffer i/o hits %	1	1
48	Spare bytes	3	0

The coefficient field is the number the data stored in the record has been multiplied by. For example, a load average of '2.5' will be stored in the file as '25'. The bytes in each record are ordered from least to most significant.

FILES

/dev/kmem, /vmunix, /etc/utmp	data sources
/usr/adm/stat/stats*	data stored by */etc/stat*
/lib/kernsyms/symdata_*	kernel symbol addresses

SEE ALSO

seestat(8)

NAME

streconfig – database of outstanding *mvst* processes

SYNOPSIS

/etc/streconfig

DESCRIPTION

The */etc/streconfig* file stores a list of all outstanding *mvst(8)* processes. Each */etc/streconfig* entry takes the form of the invoking *mvst(8)* command line, with additional capabilities which allow the VVM utilities to interrupt and control *mvst(8)* processes. *mvst(8)* writes an entry to */etc/streconfig* before transferring data, and deletes the entry after the transfer is completed. The file is used exclusively by the VVM stripe utilities and is not intended to be modified by any other means.

Under error conditions (such as a system crash) the *vvmdaemon* can read the file and restart any failed reconstructions with *mvst(8)*. Locking of the file is supported both by the *mvst(8)* utility and by the *vvmdaemon* using the *flock(2)* facility.

CAPABILITIES

The command line which invoked *mvst(8)* supplemented by two additional switches. The pid of the mvst process follows a -P switch, this information is used by *rmst* to interrupt non-redunadant data moves. And a -r switch followed by the restart offset in logical blocks, a comma, then the stripe sector size in bytes. If the original command included the -H option, then the streconfig entry will replace the -H with the actual disk device that was selected by the mvst process.

A Sample Entry

Invoking the **mvst** utility using:

```
/etc/mvst -H /dev/rst6 da4a
```

yields a streconfig entry of:

```
/etc/mvst -P 12345 -r 0,2048 st6 da4a da5a
```

FILES

/etc/streconfig Database of outstanding *mvst* processes.

SEE ALSO

st(4), stripecap(5), newst(8), getst(8), putst(8), mvst(8), rmst(8), qst(8), convst(8)

NAME

stripecap – stripe description data base

SYNOPSIS

/etc/stripecap

DESCRIPTION

Stripecap is a database describing file system stripes. It is used exclusively by the stripe utilities and is not intended to be modified by any other means.

Stripes are logical disk partitions which are interleaved over several physical disk partitions. A striped filesystem is a regular ConvexOS filesystem which uses a striped disk partition. Striped disk partitions are used to take advantage of performance improvements made possible by parallel operation of the multiple disk devices which make up stripes. ConvexOS filesystems can be mounted on stripes just as with "normal" disk partitions.

The "striped filesystem" is supported by the *Virtual Volume Manager (VVM).* The VVM driver provides redundant and non-redundant stripe capability. Redundant stripes protect against data loss from disk failure by either data mirroring or use of parity.

Stripes are described in *stripecap* by a textual descriptor, which contains information on the physical disk partitions which make up the stripe, and information about the layout of the logical sections of the stripe. Entries in *stripecap* consist of the entry name followed by a number of ':' separated fields. There are three entry types available; stripe entries, Hot Spare entries, and the entry **vvm:** which must appear once in the *stripecap* file to identify it as *VVM* compatible.

The first entity of a stripe entry gives the name of the stripe partition. It is of the form **st***X*, where *X* is an integer corresponding to the minor device number of the stripe pseudo-device */dev/rst?*. The *stripecap* file also contains information on *VVM* Hot Spare partitions. These are disk partitions which are used by the *VVM* system to rebuild failed partitions in a *redundant* stripe. The Hot Spare entries are of the form **hs***X*, where *X* is an integer defining the sequence in which the Hot Spare appears in the *stripecap* file.

CAPABILITIES

st entries:

Name	Description
ST#	indicates stripe redundancy, 1 is true (*redundancy*), otherwise 0.
NS#	the # of sections in the stripe.
SS#	the stripe "sector size." Must be a multiple of 512. If the stripe is redundant or contains an IDC disk, then '#' must also be a multiple of 2048.
D<s>#	the # of disk partitions which comprise section *s*. '#' must be from 0 to 32.
B<s>#	the # of logical blocks from each disk in section *s* that contribute to the stripe section.
S<s>#	stripe blocking factor for section *s*. For redundant stripes, '#' must be 128 or smaller.
F<s>#	index of failed partition, set to D<s> if no failures.

<s><d>=major<mj>minor<mn>offset<o>(name)
defines the component disk partition *d* of section *s*. *mj* is the major number, *mn* is the minor number, and *o* is the offset into the disk partition at which section *s* begins. The *(name)* component is optional and if present identifies the file system special device name associated with the partition (e.g. du1c.)

hs entries:

Name	Description
MJ#	the major number of the Hot Spare.
MN#	the minor number of the Hot Spare.
SS#	the disk partition sector size.
LN#	the length of the Hot Spare partition.

OF# the current operational offset into the partition.

NM=name the special device name of the Hot Spare.

A<i># defines an "affinity" for the Hot Spare towards the given stripe. The "#" is the stripe number of the stripe desired. The *i* is a number indicating the affinity priority and must be unique (ascending from 0 in decreasing order of priority.)

U<i># Denotes a section of the Hot Spare which is used by stripe number "#". The *i* is a number indicating the order in which the section has been allocated (ascending from 0.)

Note that the section descriptor *s* must be a lower case letter, the disk descriptors *d* must an integer, and all block arguments must be presented in logical units.

Fields Within the Descriptor

Entries may continue onto multiple lines by giving a \ as the last character of a line. Empty fields may be included for readability. Each entry must start on the beginning of a line (i.e. the previous line must not have ended with a continuation). A "#" in the first column causes the entire line to be intrepreted as a comment.

EXAMPLES

/etc/newst -Re st6 du1a dkd-502 du2h dkd-502 du3h dkd-502

yields a stripecap entry such as:

```
st6:\
        :ST#1:NS#2:SS#2048:\
        :Da#3:Ba#98400:Sa#128\
                :a0=major64minor520offset0(du2h):\
                :a1=major64minor776offset0(du3h):\
                :a2=major64minor257offset0(du1a):\
        :Db#2:Bb#488400:Sb#128\
                :b0=major64minor520offset50380800(du2h):\
                :b1=major64minor776offset50380800(du3h):
```

/etc/newst -H /dev/rst6 du4a dkd-502

yields a stripecap entry such as:

```
hs0:\
        :MJ#64:MN#1025:SS#512:LN#250000:OF#0:\
        :NM=du4a:\
        :A0#6:
```

In this case the Hot Spare made up of disk partition */dev/du4a* is established as a Hot Spare with an *affinity* for stripe #6. Note that if *affinities* are present, then that Hot Spare can be used *only* for the stripes listed as having an affinity.

FILES

/etc/stripecap Database of the stripe descriptors.

SEE ALSO

st(4), newst(8), getst(8), putst(8), mvst(8), rmst(8), qst(8), convst(8)

NAME

tar – POSIX compatible extended tape archive file format

DESCRIPTION

The extended tar format provides a mechanism supporting the copying of files from an archival/transport medium to the files hierarchy, and from the file hierarchy to an archival/transport medium. The utilities *pax(1)* and *tar(1)* provide this functionality under the ConvexOS operating system.

An extended *tar* archive tape or file contains a series of blocks. Each block is a fixed size block of 512 bytes. Each file archived is represented by a header block that describes the file, followed by zero or more blocks that give the contents of the file. At the end of the archive file are two blocks filled with binary zeroes, interpreted as an end-of-archive indicator.

The blocks may be grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the application utility creating the archive file, either *pax(1)* or *tar(1)* on ConvexOS) may be written with a single *write()* operation. On magnetic tape, the result of this write is a single tape record. The last group of blocks is always at the full size, so blocks after the two zero blocks contain undefined data.

The header block for the extended tar format is shown below. All lengths and offsets are in decimal.

Field Name	Byte Offset	Length (in bytes)
name	0	100
mode	100	8
uid	108	8
gid	116	8
size	124	12
mtime	136	12
chksum	148	8
typeflag	156	1
linkname	157	100
magic	257	6
version	263	2
uname	265	32
gname	297	32
devmajor	329	8
devminor	337	8
prefix	345	155

Symbolic constants used in the header block are defined in the header <**tar.h**> as follows:

```
        /* Values used in magic and version fields */
#define TMAGIC      "ustar" /* ustar and a null */
#define TMAGLEN     6       /* length of TMAGIC */
#define TVERSION    "00"    /* 00 and no null */
#define TVERSLEN    2       /* length of TVERSION */

        /* Values used in typeflag field */
#define REGTYPE     '0'     /* Regular file */
#define AREGTYPE    '\0'    /* Regular file */
#define LNKTYPE     '1'     /* Link*/
#define SYMTYPE     '2'     /* Symbolic link */
#define CHRTYPE     '3'     /* Character special */
#define BLKTYPE     '4'     /* Block special */
#define DIRTYPE     '5'     /* Directory*/
```

```
#define FIFOTYPE   '6'      /* FIFO Special */
#define CONTTYPE   '7'      /* Reserved */

        /* Bits used in the mode field - values in octal */
#define TSUID      04000    /* Set UID on execution */
#define TSGID      02000    /* Set GID on execution */
#define TSVTX      01000    /* Reserved */

        /* File Permissions */
#define TUREAD     00400    /* read by owner */
#define TUWRITE    00200    /* write by owner */
#define TUEXEC     00100    /* execute/search by owner */
#define TGREAD     00040    /* read by group */
#define TGWRITE    00020    /* write by group */
#define TGEXEC     00010    /* execute/search by group */
#define TOREAD     00004    /* read by other */
#define TOWRITE    00002    /* write by other */
#define TOEXEC     00001    /* execute/search by other */
```

All characters are represented in the American Standard Code for Information Interchange, ASCII. For maximum portability between implementations, names should be selected from the characters represented by the **portable filename character set** as 8-bit characters with zero parity.

Each field within the header block is contiguous; that is, there is no padding used. Each character on the archive medium is stored contiguously.

The fields *magic*, *uname*, and *gname* are null-terminated character strings. The fields *name*, *linkname* and *prefix* are null-terminated character strings except when all characters in the array contain non-null characters including the last character. The *version* field is two bytes containing the characters "00" (zero-zero). The *typeflag* contains a single character. All other fields are leading zero-filled octal numbers in ASCII. Each numeric field is terminated by one or more space or null characters.

The *name* and *prefix* fields produce the pathname of the file. The hierarchical relationship of the file is retained by specifying the pathname as a path prefix, a slash character and the filename as the suffix. If the *prefix* contains non-null characters, *prefix*, a slash character and *name* are concatenated without modification or addition of new characters to produce a new pathname. In this manner, pathnames of at most 256 characters can be supported. If a pathname does not fit in the space provided, an error is reported, and no attempt will be made to store the file in the archive.

The *linkname* field does not use the *prefix* to produce a pathname, and as such, a *linkname* is limited to 99 characters. If the name does not fit in the space provided, an error is reported, and no attempt will be made to store the link in the archive.

The *mode* field provides 9 bits specifying file permissions and 3 bits to specify the set UID, set GID, and TSVTX modes.

The *uid* and *gid* fields are the user and group ID of the file's owner and group, respectively.

The *size* field is the size of the file in bytes. If the *typeflag* field is set to specify a file to be of LINKTYPE or SYMTYPE the *size* field shall be specified as zero. If the *typeflag* field is set to specify a file of type DIRTYPE the *size* field in interpreted as described under the definition of that record type. If the *typeflag* field is set to CHARTYPE, BLKTYPE, or FIFOTYPE, the meaning of the *size* Field is implementation-defined and no data blocks are stored on the medium. If the *typeflag* field is set to any other value, the number of blocks written following the header is (*size* +511)/512 ignoring any fraction in the result of the division.

The *mtime* field is the modification type of the file at the time it was archived. It is the ASCII representation of the octal value of the modification time obtained from the *stat()* function.

The *chksum* field is the ASCII representation of the octal value of the simple sum of all bytes in the header block. Each 8-bit byte in the header is treated as an unsigned value. These values are added to an unsigned integer, initialized to zero, the precision of which shall be no less than 17 bits. When calculating the checksum, the *chksum* field is treated as if it were all blanks.

The *typeflag* field specifies the type of file archived. If the file type is unrecognized, or if the user lacks sufficient privilege to create a file type, the file is extracted as if it were a regular file, providing that the file type is defined to have a meaningful size field. If conversion to an ordinary file occurs, an error message reports that a conversion took place.

The *magic* field is the specification that this archive was output in this archive format. If this field contains TMAGIC, the *uname* and *gname* fields shall contain the ASCII representation of the owner and group of the file respectively. When the file is restored by a privileged, protection-preserving version of the utility, the password and group files are scanned for these names, and the user and group IDs contained within will be used instead of the *uid* and *gid* field contents.

The encoding of the header is designed to be portable across machines.

SEE ALSO

tar(1), pax(1), P1003.1, Section 10.1.1

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

termcap is a data base describing terminals, used, *e.g.*, by *vi*(1) and *curses*(3X). 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
bl	str	(P)	audible signal (bell)
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	bool		Display may be retained below
dB	num		Number of millisec of bs delay needed
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
le	str	(P)	move cursor left one position
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
mb	str		turn on blinking attribute
md	str		turn on bold (extra bright) attribute
me	str		turn off all attributes
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**)
rc	str	(P)	restore cursor to position of last **sc**
rs	str		reset terminal completely to sane modes
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than **^I** or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use **cm**
ti	str		String to begin programs that use **cm**
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue
ul	bool		Terminal underlines even though it doesn't 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
vt	num		virtual terminal number (not supported on all systems)

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

termcap is a data base describing terminals, used, *e.g.*, by *vi*(1) and *curses*(3X). 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
bl	str	(P)	audible signal (bell)
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	bool		Display may be retained below
dB	num		Number of millisec of bs delay needed
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
le	str	(P)	move cursor left one position
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
mb	str		turn on blinking attribute
md	str		turn on bold (extra bright) attribute
me	str		turn off all attributes
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**)
rc	str	(P)	restore cursor to position of last **sc**
rs	str		reset terminal completely to sane modes
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last

te	str	String to end programs that use **cm**
ti	str	String to begin programs that use **cm**
uc	str	Underscore one char and move past it
ue	str	End underscore mode
ug	num	Number of blank chars left by us or ue
ul	bool	Terminal underlines even though it doesn't 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
vt	num	virtual terminal number (not supported on all systems)
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|lsi adm3:am:bs:cl=^Z:li#24:co#80
```

Cursor addressing

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

%d	as in *printf*, 0 origin
%2	like %2d
%3	like %3d
%.	like %c
%+x	adds *x* to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %
%n	exclusive or row and column with 0140 (DM2500)

%B BCD (16*(x/10)) + (x%10), no output.
%D 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 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 canceled with **xx@** where xx is the capability. For example, the entry

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

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

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

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

BUGS

ex allows only 256 characters for string capabilities, and the routines in *termcap*(3X) 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

tp-acct – tape allocation accounting file

DESCRIPTION

The file */usr/adm/tp-acct* contains a record of tape allocations and deallocations performed by *tpmount* and *tpunmount*. Each successful allocation or deallocation causes one line to be written to */usr/adm/tp-acct* containing the following fields in the order they are listed:

event	The word **allocated** or **deallocated.**
time	The time that the event occurred, as returned by *time*(3).
device	The physical device allocated. This field is present only if *event* is the word **allocated.**
uid	The user ID of the allocator at allocation time.
gid	The group ID of the allocator at allocation time.
aid	The activity ID of the allocator at allocation time.
drive name	The tape drive name as given in the configuration database maintained by *tpconfig*.

FILES

/usr/adm/tp-acct

SEE ALSO

tpmount(1), tpconfig(1), sumscripts(8),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

NAME

ttys – terminal initialization data

DESCRIPTION

The *ttys* file contains information that is used by various routines to initialize and control the use of terminal special files. This information is read with the *getttyent*(3) library routines. There is one line in the *ttys* file per special file. Fields are separated by tabs and/or spaces. Some fields may contain more than one word and should be enclosed in double quotes. Blank lines and comments can appear anywhere in the file; comments are delimited by '#' and new line. Unspecified fields default to null. The first field is the terminal's entry in the device directory, /dev. The second field of the file is the command to execute for the line, typically *getty*(8), which performs such tasks as baud-rate recognition, reading the login name, and calling *login*(1). It can be, however, any desired command, for example the start up for a window system terminal emulator or some other daemon process, and can contain multiple words if quoted. The third field is the type of terminal normally connected to that tty line, as found in the *termcap*(5) data base file. The remaining fields set flags in the *ty_status* entry (see *getttyent*(3)) or specify a window system process that *init*(8) will maintain for the terminal line. As flag values, the strings 'on' and 'off' specify whether *init* should execute the command given in the second field, while 'secure' in addition to 'on' allows root to login on this line. An optional 'dialup' or 'uucp' string can also be specified after the 'secure' string. The 'dialup' string on a line requires all users to enter a dialup password when logging on to that tty. The 'uucp' string allows a user with the same group id as the user 'uucp' to log in to that tty. Ttys designated as 'dialup' do not have to be named in the ttyd? format. These flag fields should not be quoted. The string 'window=' is followed by a quoted command string which *init* will execute before starting *getty*. If the line ends in a comment, the comment is included in the *ty_comment* field of the ttyent structure.

The *ttys* file can also be used to allow or restrict user or group access to a specific tty line. The restriction parameters are placed before the comment and on the same line as the tty that they affect. Users are denoted by their login names and groups by the group name placed between "<" and ">". An optional "!" can be placed before a user or group name to specify that access is denied to that user or group. If no user or group names appear on a line then by default all users are allowed to login on that terminal. If a group or user name appears on a line then only those users or members of that group are allowed to login on that tty. The user and group names have precedence from left to right.

Some examples:

```
console "/etc/getty std.1200"     vt100       on secure
tty00   "/etc/getty d1200"        vt100       on dialup # 555-1234
ttyh0   "/etc/getty std.9600"     hp2621-nl   on        # 254MC
ttyh1   "/etc/getty std.9600"     plugboard   on        # John's office
ttyp0   none                      network
ttyp1   none                      network     off
ttyv0   "/usr/new/xterm -L :0"    vs100       on        window="/usr/new/Xvs100 0"
tty01   "/etc/getty std.9600"     vt100       on secure smith <engr> # engineering tty
tty02   "/etc/getty std.9600"     vt100       on secure jones !<engr> # marketing tty
```

The first example permits root login on the console at 1200 baud, the second allows dialup at 1200 baud without root login and will prompt for a dialup password, the third and fourth allow login at 9600 baud with terminal types of "hp2621-nl" and "plugboard" respectively, the fifth and sixth line are examples of network pseudo ttys, which should not have *getty* enabled on them, and the seventh example shows a terminal emulator and window system startup entry. The last two examples are of giving or dening access to a particular tty. The eighth example gives the user smith and all members of the engr group access to tty00. The ninth example gives the user jones and all users except the ones in the engr group access to tty01.

FILES

/etc/ttys

SEE ALSO

login(1), getttyent(3), gettytab(5), init(8), getty(8)

NAME
ttytype – data base of terminal types by port

SYNOPSIS
/etc/ttytype

DESCRIPTION
This file has been obsoleted by the new *ttys(5)*.

SEE ALSO
ttys(5)

NAME

types – primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

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

```
/*      $CHeader: types.h 1.6 91/03/01 14:43:19 $*/
/*      Copyright 1984-1990 Convex Computer Corp.*/
/*      types.h   7.1     (Berkeley) 6/4/86*/

#ifndef _SYS_TYPES_H_ /* { _SYS_TYPES_H_*/
#define _SYS_TYPES_H_

/*
 * Warning:  the contents of this file are somewhat regulated by POSIX;
 * therefore, "random" nested includes (outside the kernel) are not allowed
 * as they might "pollute" the namespace...
 */

/*
 * We are allowed to add new types ending with '_t' here.  The
 * following should gradually replace the old u_char, u_int, etc.
 */

typedef  unsigned char     uchar_t;
typedef  unsigned short    ushort_t;
typedef  unsigned int      uint_t;
typedef  unsigned long     ulong_t;

/*
 * The 64 bit long long types are available neither in the
 * cross compiler nor in strictly conforming modes of the native
 * convex 4.0 C compiler.
 */
#ifndef _LONGLONG     /* { _LONGLONG*/
#define _LONGLONG
#if       defined(__STDC__) || defined(__mc68k__)/* strict/cross*/
#define _LL_HI    0
#define _LL_LO    1
typedef  long                 s64_t[2];
typedef unsigned long         u64_t[2];
#else                         /* strict/cross*/
typedef  long long            s64_t;
typedef  unsigned long longu64_t;
#endif                        /* strict/cross */
#endif                        /* } __LONGLONG*/

typedef  ushort_t  nlink_t;
typedef  ulong_t              ino_t;
typedef  short                dev16_t;
typedef  int                  dev_t;
typedef float                 pri_t;

#ifndef __UID_T               /* { __UID_T */
/* also defined in pwd.h and unistd.h */
typedef  ushort_t  uid_t;
#define __UID_T
#endif                        /* } __UID_T */

#define _SAME_UID       ((uid_t)(-1))
```

```
#ifndef __GID_T                 /* { __GID_T */
/* also defined in grp.h, pwd.h and unistd.h */
typedef  ushort_t  gid_t;
#define __GID_T
#endif                          /* } __GID_T */

#define  _SAME_GID      ((gid_t)(-1))

#ifndef __MODE_T                /* { __MODE_T */
/* also defined in unistd.h and fcntl.h */
typedef  ushort_t  mode_t;
#define __MODE_T
#endif                          /* } __MODE_T */

#ifndef __OFF_T                 /* { __OFF_T */
/* also defined in unistd.h and fcntl.h */
typedef  long                   off_t;
#define __OFF_T
#endif                          /* } __OFF_T */

#ifndef __PID_T                 /* { __PID_T*/
/* This occurs also in wait.h, signal.h, fcntl.h, and unistd.h */
#define __PID_T
typedef  short                  pid_t;
#endif                          /* } __PID_T*/

#ifndef __TIME_T                /* { __TIME_T*/
/* This occurs also in time.h */
#define __TIME_T
typedef  long                   time_t;
#endif                          /* } __TIME_T*/

#ifndef __SSIZE_T               /* { __SSIZE_T */
/* also defined in unistd.h */
typedef  int            ssize_t;
#define __SSIZE_T
#endif                          /* } __SSIZE_T */

#ifndef __SIZE_T  /* { __SIZE_T */
/* also defined in unistd.h */
typedef unsigned int            size_t;
#define __SIZE_T
#endif                          /* } __SIZE_T */

typedef unsigned long           priv_t;
typedef  unsigned long          mask_t;

#ifdef   _AUDIT                 /* { _AUDIT*/
typedef  unsigned long          uint;
typedef  unsigned long          ulong;
typedef  union obj {
        char    *o_file;
        int             o_pid;
        int             o_soclkid;
        int             o_fdes;
}       obj_t;
#endif                          /* } _AUDIT*/

#if     !__stdc__ || defined(_CONVEX_SOURCE)/* { !__stdc__ || _C_S*/

/*
 * Basic system types and major/minor device constructing/busting macros.
 */

/* major part of a device */
```

```
#define major_dev16t(x) ((int)(((unsigned)(x)>>8)&0377))
#define major(x) ((int)( ( (unsigned)(x) >> 20 ) & 0xfffff) )

/* minor part of a device */
#define minor_dev16t(x) ((int)((x)&0377))
#define minor(x) ((int)((x)&0xfffff))

/* make a device number */
#define makedev_dev16t(x,y)((dev_t)(((x)<<8) | (y)))
#define makedev(x,y)    ((dev_t)( ((x) << 20 ) | (y) ))

        /* The following should succumb to..*/
typedef unsigned char   u_char;/* uchar_t*/
typedef unsigned short  u_short;/* ushort_t*/
typedef unsigned int    u_int;/* uint_t*/
typedef unsigned long   u_long;/* ulong_t*/
typedef unsigned short  ushort;/* ushort_t*/

#ifdef  __convex__      /* { __convex__*/
typedef struct  _physadr { int r[1]; } *physadr;/* FIXME */
typedef struct  label_t {
        int     val[20];/* Enough to save a0-a7, s0-s7, and pc */
} label_t;
#endif                  /* } __convex__*/

typedef struct  kabel_t{
        int     pc;
        int     psw;
        int     fp;
        int     ap;
        int     sp;
} kabel_t;
typedef struct  _quad { long val[2]; } quad;
typedef long    daddr_t;
typedef char *  caddr_t;
typedef long    segsize_t;

typedef union {
        struct {
                uid_t   uid;
                gid_t   gid;
                mode_t mode;
        }       unixmode;
        caddr_t ptr;
        uint_t  count;
}       dac_t;


#define NBBY    8       /* number of bits in a byte */
/*
 * Select uses bit masks of file descriptors in longs.
 * These macros manipulate such bit fields (the filesystem macros use chars).
 * FD_SETSIZE may be defined by the user, but the default here
 * should be >= NOFILE (param.h).
 */
#ifndef FD_SETSIZE
#define FD_SETSIZE      256
#endif

typedef long    fd_mask;
#define NFDBITS (sizeof(fd_mask) * NBBY)/* bits per mask */
#ifndef howmany
#define howmany(x, y)   (((x)+((y)-1))/(y))
#endif
```

```
typedef  struct fd_set {
        fd_mask  fds_bits[howmany(FD_SETSIZE, NFDBITS)];
} fd_set;

#define  FD_SET(n, p)    ((p)->fds_bits[(n)/NFDBITS] |= (1 << ((n) % NFDBITS)))
#define  FD_CLR(n, p)    ((p)->fds_bits[(n)/NFDBITS] &= ~(1 << ((n) % NFDBITS)))
#define  FD_ISSET(n, p)  ((p)->fds_bits[(n)/NFDBITS] & (1 << ((n) % NFDBITS)))
#define FD_ZERO(p)       bzero((char *)(p), sizeof(*(p)))
/* Sun... typedef  struct  fd_set { int fds_bits[1]; } fd_set; */


/*      Differentiate between the different types of "block numbers" */
typedef  long       dkblk_t;/* disk block numbers */
typedef  long       fsblk_t;/* file system block numbers */
typedef  long       swblk_t;/* swap block numbers */

/*      Binary semaphores & queues  */
typedef unsigned short     bsema_t;/* indivisible lock type        */

/* Shared memory semaphore */
struct   semaphore        {/* BSD semaphore - machine dependent */
         char       lock;   /* the test and set lock byte */
         char       awakened;/* tas wakeup has been done byte */
         };
typedef  struct semaphore semaphore;
typedef int     iobsema_t;      /* index into tas semaphore page */

/*
 * Expanded offsets for new stat structure
 */
#ifndef __OFF64_T
     typedef s64_t   off64_t;
#    define __OFF64_T
#endif

typedef s64_t       ino64_t;

#endif                          /* } !__stdc__ || _C_S*/
#endif                          /* } _SYS_TYPES_H_*/
```

The form *daddr_t* is used for disk addresses except in an i-node on disk, see *fs*(5). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

SEE ALSO

fs(5), time(3c), lseek(2), adb(1)

NAME

utmp, wtmp – login records

SYNOPSIS

#include <utmp.h>

DESCRIPTION

The *utmp* file records information on who is currently using the system. The file is a sequence of entries with the following structure declared in the include file:

```
/*      $CHeader: utmp.5 0.12 91/10/08 21:06:40 $       */
/*      Copyright 1984 Convex Computer Corp.            */

/*
 * Structure of utmp and wtmp files.
 *
 * Assuming the number 8 is unwise.
 */
struct utmp {
        char    ut_line[8];             /* tty name */
        char    ut_name[8];             /* user ID */
        char    ut_host[16];            /* host name, if remote */
        long    ut_time;                /* time on */
};

/*
 * This is a utmp entry that does not correspond to a genuine user
 */
#define nonuser(ut) ((ut).ut_host[0] == 0 && \
        strncmp((ut).ut_line, "tty", 3) == 0 && ((ut).ut_line[3] == 'p' \
        || (ut).ut_line[3] == 'q' || (ut).ut_line[3] == 'r' \
        || (ut).ut_line[3] == 's'))
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of *time*(3C).

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

wtmp is maintained by *login*(1) and *init*(8). Neither of these programs creates the file, so if it is removed, record-keeping is turned off. It is summarized by *ac*(8) and *last*(1). These programs rely on *wtmp* for their data, so if *wtmp* is removed, they will complain that there is no *wtmp* file.

FILES

/etc/utmp
/usr/adm/wtmp

SEE ALSO

last(1), login(1), who(1), bill-acct(5), ac(8), connecttime(8), init(8),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

NAME

uuencode – format of an encoded uuencode file

DESCRIPTION

Files output by *uuencode(1C)* consist of a header line, followed by a number of body lines, and a trailer line. *Uudecode(1C)* 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(1C), uudecode(1C), uusend(1C), uucp(1C), mail(1)

NAME

utmp, wtmp – login records

SYNOPSIS

#include <utmp.h>

DESCRIPTION

The *utmp* file records information on who is currently using the system. The file is a sequence of entries with the following structure declared in the include file:

```
/*      $CHeader: utmp.5 0.12 91/10/08 21:06:40 $   */
/*      Copyright 1984 Convex Computer Corp.        */

/*
 * Structure of utmp and wtmp files.
 *
 * Assuming the number 8 is unwise.
 */
struct utmp {
        char    ut_line[8];             /* tty name */
        char    ut_name[8];             /* user ID */
        char    ut_host[16];            /* host name, if remote */
        long    ut_time;                /* time on */
};

/*
 * This is a utmp entry that does not correspond to a genuine user
 */
#define nonuser(ut) ((ut).ut_host[0] == 0 && \
        strncmp((ut).ut_line, "tty", 3) == 0 && ((ut).ut_line[3] == 'p' \
        || (ut).ut_line[3] == 'q' || (ut).ut_line[3] == 'r' \
        || (ut).ut_line[3] == 's'))
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of *time*(3C).

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

wtmp is maintained by *login*(1) and *init*(8). Neither of these programs creates the file, so if it is removed, record-keeping is turned off. It is summarized by *ac*(8) and *last*(1). These programs rely on *wtmp* for their data, so if *wtmp* is removed, they will complain that there is no *wtmp* file.

FILES

/etc/utmp
/usr/adm/wtmp

SEE ALSO

last(1), login(1), who(1), bill-acct(5), ac(8), connecttime(8), init(8),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

NAME

verify – verify databases

SYNOPSIS

/usr/lib/verify/*

DESCRIPTION

The directory */usr/lib/verify* contains database files used by *verify*(8). The database files are ASCII format and contain entries naming files and information about them. There is one entry on each line. There are two types of entries. The first type specifies a file (referred to as a "file specification") and certain characteristics of that file. The second type specifies a directory name (referred to as a "directory specification").

A file specification consists of a file name, its mode, owner, and group, in that order, each separated by a single colon. A file specification may optionally include other attributes; see the description of "Optional Fields" given below.

The mode consists of 1 to 4 octal digits, with the same meaning and format as used by *chmod*(2). This is also the same as the "absolute" format described by *chmod*(1).

The owner and group may be given as either the symbolic owner/group name, or numerically, by the uid or gid, respectively.

If the first character of the owner, group, or mode is "*", then that field is not checked. If the first character of the owner or group is "?", then the file may have any owner that is listed in the password file or any group that is listed in the group file.

A directory specification specifies a new directory name to prepend to the files following it. A directory specification consists of the string **:dir:** followed by a directory name. All of the files following a **:dir:** specifier (up until the next **:dir:**) are expected to be found in the given directory. A new **:dir:** specifier changes the prepended directory.

Lines beginning with "#" are comment lines. Blank lines may also be present; they are ignored.

OPTIONAL FIELDS

Following the required fields there may be one or more optional fields. These fields are also separated by colons, and are as follows:

File type field

The type of a file may be given as **t=**=*c*, where *c* is one of the following characters:

Character	File Type
-	regular file
b	block special
c	character special
d	directory
l	symbolic link
p	named pipe
s	socket

These are the same as used in *ls*(1). If no type field is present, then the type is assumed to be "regular file."

Symbolic link field
If the file is a symbolic link, the "pointed to" file name may be specified as **l**=*file*.

Size field
The size of the file in bytes may be specified as **s**=*size*.

Checksum field
The checksum of the file (as determined by *sum*(1)) may be specified as **c**=*checksum*.

Modification time field
The modification time of the file (as determined by *stat*(2)) may be specified as **T**=*mtime*.

Version field
The version of the file (as determined by *vers*(1)) may be specified as **v**=*version*.

EXAMPLE

```
:dir:/etc
passwd:644:root:bin
# The above two lines have specified that the file
# /etc/passwd is owned by "root", its group membership
# is "bin", and its mode is 644.
:dir:/usr
ucb:775:root:bin:t=d
# The previous two lines indicate that "ucb" is a
# directory in /usr, with the owner, group, and mode
# indicated.
```

FILES

/usr/lib/verify standard database files

SEE ALSO

ls(1), sum(1), vers(1), stat(2), group(5), passwd(5), verify(8)

NAME

vfont – font formats for the Benson-Varian or Versatec

SYNOPSIS

/usr/lib/vfont/*

DESCRIPTION

The fonts for the printer/plotters have the following format. Each file contains a header, an array of 256 character description structures, and then the bit maps for the characters themselves. The header has the following format:

```
struct header {
        short           magic;
        unsigned short  size;
        short           maxx;
        short           maxy;
        short           xtnd;
} header;
```

The *magic* number is 0436 (octal). The *maxx, maxy,* and *xtnd* fields are not used at the current time. *Maxx* and *maxy* are intended to be the maximum horizontal and vertical size of any glyph in the font, in raster lines. The *size* is the size of the bit maps for the character, in bytes. Before the maps for the characters, is an array of 256 structures for each of the possible characters in the font. Each element of the array has the form:

```
struct dispatch {
        unsigned short  addr;
        short           nbytes;
        char            up;
        char            down;
        char            left;
        char            right;
        short           width;
};
```

The *nbytes* field is nonzero for characters which actually exist. For such characters, the *addr* field is an offset into the rest of the file where the data for that character begins. There are *up+down* rows of data for each character, each of which has *left+right* bits, rounded up to a number of bytes. The *width* field is not used by *vcat*, although it is used to make width tables for *troff*. It represents the logical width of the glyph, in raster lines, and shows where the base point of the next glyph would be.

FILES

/usr/lib/vfont/*

SEE ALSO

troff(1), pti(1), vpr(1), vtroff(1), vfontinfo(1)

Index

Description	Man Page
change root directory	CHROOT(2)
change the name of a file	RENAME(2)
change working directory	CD(1)
channel control unit pseudo-device	CCU(4)
character testing and mapping macros	CTYPE(3)
chdir — change current working directory	CHDIR(2)
check for new notesfile articles	CHECKNOTES(1)
check in RCS revisions	CI(1)
check nroff/troff files	CHECKNR(1)
check out RCS revisions	CO(1)
checkeq — typeset mathematics	NEQN(1)
checknotes — check for new notesfile articles	CHECKNOTES(1)
checknr — check nroff/troff files	CHECKNR(1)
checkpoint a process or process family	CHKPNT(1)
checkpoint a process or process family	CHKPNT(3)
checkpoint a process or process family	CHKPNT(3F)
chfn — change finger entry	CHFN(1)
chgrp — change group	CHGRP(1)
chkpnt — checkpoint a process or process family	CHKPNT(1)
chkpnt — checkpoint a process or process family	CHKPNT(3)
chkpnt — checkpoint a process or process family	CHKPNT(3F)
chkpnt — process checkpoint file format	CHKPNT(5)
chmagic — change magic number of executable file	CHMAGIC(1)
chmod — change mode	CHMOD(1)
chmod — change mode of file	CHMOD(2)
chown — change owner and group of a file	CHOWN(2)
chroot — change root directory	CHROOT(2)
chsh — change default login shell	CHSH(1)
ci — check in RCS revisions	CI(1)
circle — graphics interface	PLOT(3X)
clear — clear terminal screen	CLEAR(1)
clear terminal screen	CLEAR(1)
clearerr — stream status inquiries	FERROR(3S)
clock — get processor time used	CLOCK(3)
close — delete a descriptor	CLOSE(2)
close or flush a stream	FCLOSE(3S)
closedir — directory operations	DIRECTORY(3)
closelog — control system log	SYSLOG(3)
closepl — graphics interface	PLOT(3X)
cmp — compare two files	CMP(1)
co — check out RCS revisions	CO(1)
col — filter reverse line feeds	COL(1)
colcrt — filter nroff output for CRT previewing	COLCRT(1)
colrm — remove columns from a file	COLRM(1)
comm — select or reject lines common to two sorted files	COMM(1)
command language	SH(1)
compact — compress and uncompress files and cat them	COMPACT(1)
compact list of users who are on the system	USERS(1)
compare RCS revisions	RCSDIFF(1)

Description	Man Page
get processor time used	CLOCK(3)
get protocol entry	GETPROTOENT(3N)
get service entry	GETSERVENT(3N)
get socket name	GETSOCKNAME(2)
get system information	GETSYSINFO(2)
get system information	UNAME(2)
get system page size	GETPAGESIZE(2)
get system time	TIME(3C)
get terminal name	TTY(1)
get the current working directory for the process	GETCWD(3)
get ttys file entry	GETTTYENT(3)
get user identity	GETUID(2)
get user name	CUSERID(3)
getactaid — get activity file entry	GETACTENT(3)
getactent — get activity file entry	GETACTENT(3)
getactnam — get activity file entry	GETACTENT(3)
getacwent — get actwho file entry	GETACWENT(3)
getaid — get a process' activity ID	GETAID(2)
getc — get character or word from stream	GETC(3S)
getchar — get character or word from stream	GETC(3S)
getcwd — get the current working directory for the process	GETCWD(3)
getdirentries — gets directory entries in a filesystem independent format	GETDIRENTRIES(2)
getdiskbyname — get disk description by its name	GETDISKBYNAME(3X)
getdomainname — get/set name of current yellow pages domain	GETDOMAINNAME(2)
getdtablesize — get descriptor table size	GETDTABLESIZE(2)
getegid — get group identity	GETGID(2)
getenv — manipulate environmental variables	GETENV(3)
geteuid — get user identity	GETUID(2)
getfpmode — get current floating point mode	GETFPMODE(3)
getfsent — get file system descriptor file entry	GETFSENT(3X)
getfsfile — get file system descriptor file entry	GETFSENT(3X)
getfsspec — get file system descriptor file entry	GETFSENT(3X)
getfstype — get file system descriptor file entry	GETFSENT(3X)
getgid — get group identity	GETGID(2)
getgrent — get group file entry	GETGRENT(3)
getgrgid — group database access routines	GETGRGID(3)
getgrnam — group database access routines	GETGRGID(3)
getgroups — get group access list	GETGROUPS(2)
gethostbyaddr — get network host entry	GETHOSTBYNAME(3N)
gethostbyname — get network host entry	GETHOSTBYNAME(3N)
gethostent — get network host entry	GETHOSTBYNAME(3N)
gethostid — get/set unique identifier of current host	GETHOSTID(2)
gethostname — get/set name of current host	GETHOSTNAME(2)
getitimer — get/set value of interval timer	GETITIMER(2)
getlogin — get login name	GETLOGIN(3)
getlogin — get user name	CUSERID(3)
getmntent — get file system descriptor file entry	GETMNTENT(3)
getnetbyaddr — get network entry	GETNETENT(3N)

Description	Man Page
set process id	SETPID(2)
set program priority	NICE(3C)
set real and effective group ID	SETREGID(2)
set real and effective user ID's	SETREUID(2)
set terminal options	STTY(1)
set terminal tabs	TABS(1)
set time zone information	TZSET(3)
set user and group ID	SETUID(3)
setactent — get activity file entry	GETACTENT(3)
setacwent — get actwho file entry	GETACWENT(3)
setaid — set a process' activity ID	SETAID(2)
setbuf — assign buffering to a stream	SETBUF(3S)
setbuffer — assign buffering to a stream	SETBUF(3S)
set/display version numbers	VERS(1)
setdomainname — get/set name of current yellow pages domain	GETDOMAINNAME(2)
setenv — manipulate environmental variables	GETENV(3)
setfpmode — set floating point mode during program execution	SETFPMODE(3)
setfsent — get file system descriptor file entry	GETFSENT(3X)
setgid — set user and group ID	SETUID(3)
setgrent — get group file entry	GETGRENT(3)
setgroups — set group access list	SETGROUPS(2)
sethostent — get network host entry	GETHOSTBYNAME(3N)
sethostid — get/set unique identifier of current host	GETHOSTID(2)
sethostname — get/set name of current host	GETHOSTNAME(2)
setitimer — get/set value of interval timer	GETITIMER(2)
setjmp — non-local goto	SETJMP(3)
setkey — encryption operations	CRYPT(3)
setlinebuf — assign buffering to a stream	SETBUF(3S)
setlocale — functions for locale manipulation	SETLOCALE(3)
setlogmask — control system log	SYSLOG(3)
setmntent — get file system descriptor file entry	GETMNTENT(3)
setnetent — get network entry	GETNETENT(3N)
setpattr — get/set process attributes	GETPATTR(2)
setpflags — get/set process entry flags	GETPFLAGS(2)
setpgid — set process group	SETPGID(2)
setpgrp — set process group	SETPGRP(2)
setpid — set process id	SETPID(2)
setpriority — get/set program scheduling priority	GETPRIORITY(2)
setprotoent — get protocol entry	GETPROTOENT(3N)
setpwent — get password file entry	GETPWENT(3)
setpwrestent — get entry from password restrictions file	GETPWRESTENT(3)
setregid — set real and effective group ID	SETREGID(2)
setreuid — set real and effective user ID's	SETREUID(2)
setrlimit — control maximum system resource consumption	GETRLIMIT(2)
setservent — get service entry	GETSERVENT(3N)
setsid — create session and set process group ID	SETSID(2)
setsockopt — get and set options on sockets	GETSOCKOPT(2)
setstate — better random number generator; routines for changing generators	RANDOM(3)

Order Number
DSW-332

Document Number
710-015830-001