TEK SOFTWARE MANUAL

Part No. 070-5317-01
Product Group 65

UTek

COMMAND REFERENCE

VOLUME 2

First Printing NOV 1984
Revised SEP 1985

Tektronix®
COMMITTED TO EXCELLENCE

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following individuals and institutions for their role in its development:

W. N. Joy
O. Babaouglu
R. S. Fabry
K. Sklower
Eric P. Allman
M. K. McKusick
E.Cooper
David Musher
S. J. Leffler

University of California at Berkeley
Department of Electrical Engineering and Computer Science

The MH Mail System is based on software developed by the Rand Corporation.

Portions of this document are based on the RCS Revision Control System, © 1982 Walter F. Tichy.

Revision
INFORMATION

PRODUCT: 6000 Family Intelligent Graphics Workstation
This manual supports the following versions of this product: V2.2

REV DATE	DESCRIPTION
NOV 1984	Original Issue
JAN 1985	Added: ERROR(3C), MKTEMP(3C), TIME(3F), FERROR(3S) Changed: HSEARCH(3C) - pg. 2
MAR 1985	Revised to support Version **2.1**.
SEP 1985	Revised to support Version **2.2**. Part number rolled to 070-5317-01.

Contents

Volume 2

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 "int" unless otherwise noted. An error number is also made available in the external variable *errno*. *Errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

The following is a complete list of the errors and their names as given in ⟨*errno.h*⟩. Only these symbolic names for error numbers should be used in programs, since the actual value of the error number may vary with the implementation. Certain implementations may contain extensions which prevent some errors from ever occurring.

0 Unused

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

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

3 ESRCH No such process
No such process can be found corresponding to that specified by the process ID.

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 beyond the limits of the device. It may also occur when, for example, a tape drive is not on line, or a disk pack is not loaded on a drive.

7 E2BIG Argument list too long
An argument list longer than NCARGS (defined in ⟨sys/param.h⟩) bytes is presented to a member of the *exec* family.

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, is not in the proper format for an executable object program. See *a.out*(5).

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

10 ECHILD No children from process
Wait and the process has no living or unwaited–for children.

11 EAGAIN No more processes
A *fork* failed because the system's process table is full or the user is not allowed to create any more processes. This may be a temporary condition and subsequent calls to the same routine may complete normally.

12 ENOMEM Not enough core or swap space
During an *execve* or *brk* or *sbrk*, 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 core size 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 a hardware fault in attempting to access an argument of a system call.

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

16 EBUSY Device busy
An attempt was made to access a device that was already in use, such as mounting a device that is already mounted. This error is also returned if an attempt is made to dismount a device on which there is an active file directory (open file, current directory, mounted–on file, active text segment).

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

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

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

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

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

22 EINVAL Invalid argument
Some invalid argument (e.g., mentioning an unknown signal in *kill,* reading or writing a file for which *lseek* has generated a negative pointer) has been used. Also set by math functions, see *intro*(3m).

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

24 EMFILE Too many open files
No process may have more than NOFILE (defined in ⟨sys/max.h⟩) file descriptors open at a time.

25 ENOTTY Request does not apply
A given request is not recognized by or does not apply to a specified file or device.

26 ETXTBSY Text file busy
An attempt was made to open for writing a shared–text file that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum file size.

28 ENOSPC No space left on device
During a *write* to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because no more disk blocks are available on the file system, or the allocation of an inode for a newly created file failed because no more inodes are available on the file system.

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 was made to make more than the legal limit of hard links to a file.

32 EPIPE Unconnected pipe
A write on a pipe or socket was attempted, 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 out of range
The argument of a function in the math package (see *intro(3m)*) is out of the domain of the function.

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

35 EWOULDBLOCK Operation would block
An operation which would cause a process to block was attempted on a object while in non–blocking mode (see *ioctl* (2)).

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 Disconnection 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 shouldn't 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 remote host
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
An request to send or receive data was disallowed because the socket is 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
Unused.

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 MAXNAMLEN characters, or an entire pathname exceeded MAXPATHLEN characters. These are defined in ⟨sys/max.h⟩.

64 EHOSTDOWN Host is down
A requested host is not responding.

65 EHOSTUNREACH Host is unreachable
A requested host is not reachable from the current node.

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
Unused.

68 EUSERS Too many users
Unused.

69 EDQUOT Disk quota exceeded
Unused.

70 ENOASCII Name contains byte with high-order bit set
A given pathname contains a non-ASCII character, a byte with the high-order bit set.

71 EMCOLLIDE Map onto something already there
New areas may only be added where there is currently no memory. If you want to replace an area, unmap it first.

72 EMRANGE Designated area out of range
Possible problems: Any part of the addressed area 1) crosses the P0:P1 boundary (on a VAX); 2) is in the u area; 3) is out of the user's address space.

73 EDFS_CD Too many chdir's to remote host
A limited number of chdir's can be made to a remote host at one time; this limit has been exceeded.

74 EDFSREF Reference is to remote file
A reference to a remote file was detected but is not supported for this system call. In some cases (link, rename) this error is returned if the two pathnames do not reference files on the same host.

75 EDFSBADRESP Response length incorrect
The response from a remote host to the system call sent to that host was of incorrect length. This is not a normal error. Contact your service representative.

76 EDFSBADCMD Bad command (invalid command or wrong length)
The Distributed File System Daemon on a remote host received a garbled command from the local host. This is not a normal error. Contact your service representative.

77 EDFSNOSUCHHOST No such host
You specified a pathname of the form //host/path, and the operating system was unable to find the location of 'host'. Check that you have the correct host name and that the host is operational on the network.

78 EDFSNOBUF Malloc failed on remote system; try smaller (⟨8k) read or write
You tried to do a read or write of more than 8192 bytes to a remote file, and the malloc(3) call to get a buffer of the size you specified failed. Try again with an 8kbyte or smaller request.

79 EDFSBADVER Remote system couldn't handle version of request
The version of the Distributed File System on the local system does not match the version of the daemon on the remote system.

80 EDFSNODAEMON DFS daemon is not running
A reference of the form "//hostname/pathname" was detected, but the attempt to convert the hostname to an internet address failed because the Distributed File System daemon on the local system was not running.

81 EDFSNOPROC No more processes on remote system
A Distributed File System reference failed because when the daemon on the remote host forked a process to handle your request, the remote system's process table was full. This may be a temporary condition.

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 NPROC.

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(1csh)*.

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(1csh)*, and *tty*(4).

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

  Each user is also a member of 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.

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

Effective User ID, 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's 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."

Super–user
: A process is recognized as a *super-user* 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 non–negative integer assigned by the system when a file is referenced by *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2), or when a socket is referenced by *socket*(2) or *socketpair*(2). The descriptor uniquely

identifies an access path to that file or socket from a given process or any of its children.

File Name

Names consisting of up to MAXNAMLEN 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 preferable to use only letters, numbers, underscores and periods within file names, since the use of non-printing and other special characters can be confusing or ambiguous in certain contexts.

Pathname and Path Prefix

A pathname is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name. The total length of a pathname must be less than MAXPATHLEN characters.

A path prefix is a pathname without the final file name.

If a pathname begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory. A slash by itself names the root directory. A null pathname refers to the current directory.

Directory

A directory is a special type of file which contains entries which are pointers to data files or other directories. 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. In the root directory, .. refers to the root directory itself.

Root Directory and Current Working Directory

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving pathname searches. A process' 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 at the time 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 indicate whether 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, those 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 class of use of the caller.

Read, write, and execute/search permissions on a file are granted to a process if:

The process's effective user ID is that of the super–user.

The process's effective user ID matches the user ID of the owner of the file and the "owner" access permissions allow the access.

The process's effective user ID does not match the user ID of the owner of the file, and either the process's effective group ID matches the group ID of the file, or the group ID of the file is in the process's group access list, and the "group" access permissions allow the access.

Neither the effective user ID nor effective group ID and group access list of the process match the corresponding user ID and group ID of the file, but the access permissions for "other users" allow access.

Otherwise, permission is denied.

Character and Block Special Files

Character and block special files are used to refer to physical devices. Certain restrictions may apply to the use of character and block special files which are implementation–dependent.

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; consult *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(3m), perror(3c).

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 which 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 which 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.

DIAGNOSTICS

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.

[EINVAL]
The options for this socket probably does not include accepting connections.

[ECONNABORTED]
Tried accepting connection on socket that has receiving shutdown.

RETURN VALUE

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

SEE ALSO

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

NAME

access — determine accessibility of file

SYNOPSIS

#include ⟨sys/file.h⟩

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

DESCRIPTION

Access checks the given file *path* for accessibility according to *mode*. *Mode* is the inclusive *or* of the following values, defined in ⟨sys/file.h⟩:

```
#define R_OK   4        * test for read permission *
#define W_OK   2        * test for write permission *
#define X_OK   1        * test for execute (search) permission *
#define F_OK   0        * test for presence of file *
```

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–user–id 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.

DIAGNOSTICS

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* is too long.

[ENOENT]
The named file does not exist.

[ENOASCII]
The argument *path* contains a byte with the high–order bit set.

[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. If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access.5n).*

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

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

RETURN VALUE

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

SEE ALSO

chmod(2), stat(2).

NAME

adjtime — correct the time to allow synchronization of the system clock

SYNOPSIS

#include ⟨sys/time.h⟩

adjtime(delta, mode)
struct timeval *delta;
int mode;

DESCRIPTION

Adjtime changes the system time, as returned by **gettimeofday(2),** in the way specified by the argument *mode.* If *mode* is T_ADJ then **adjtime** moves the time back or forward by a number of milliseconds corresponding to the timeval *delta* while keeping the monotonicity of the function. If *mode* is T_SET then *delta* milliseconds are added algebraicly to the time.

This call can be used in timeservers that synchronize the clocks of computers in a network to keep an accurate network time.

Only the super–user can call

adjtime(2). The time is normally incremented by a 20ms tick. If T_ADJ is passed and *delta* is negative, the clock is incremented with a smaller tick for the time necessary to correct the error. When *delta* is positive a larger tick is used. This way, the clock is always a monotonic function. With respect to this, **adjtime** with *mode* set to T_SET, should be used carefully and only at boot tome before any users can log on.

The T_SET mode has been introduced to make, at the process level, the operation of adding a value to the time an atomic one.

DIAGNOSTICS

Adjtime may set the following errors in *errno*:

[EINVAL]
mode is not valid.

[EFAULT]
An argument references invalid memory.

[EPERM]
The caller is not 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 the error.

SEE ALSO

gettimeofday(2), timed(8n).

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 the *name*, be assigned to the socket.

Binding a name in the Unix domain creates a socket in the file system which must be deleted by the caller when it is no longer needed (using *unlink*(2)). The file created is a side–effect of the current implementation, and will not be created in future versions of the UTek ipc domain.

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

Name for an internet domain socket has three components, an address family type, a port number, and an internet address. See */usr/include/netinet/in.h* for the appropriate data structures. A Unix domain socket has two components, an address family and a pathname to a "file" that serves as a communications rendezvous point. See */usr/include/sys/un.h* for the appropirate data structures.

Suppose you wish to set up a communications channel between two independent processes that models a client–server relationship. The server would do a *socket(2)* call (using type SOCK_STREAM) followed by a **bind,** a *listen(2),* and an *accept(2).* The client would do a *socket(2)* call, followed by a *connect(2)*

DIAGNOSTICS

The **bind** call will fail if:

[EBADF]
S is not a valid descriptor.

[ENOTSOCK]
S is not a socket.

[EADDRNOTAVAIL]
The specified address (in the internet domain) or name (in the Unix domain) is not available from the local machine.

[EADDRINUSE]
The specified address (in the internet domain) or name (in the Unix domain) is already in use.

[EMSGSIZE]
The specified address size (in the internet domain) or name (in the Unix domain) is too big for the protocol. The pathname of a Unix domain socket is limited to 108 bytes.

[EINVAL]
The socket is already bound to an address (in the internet domain) or name (in the Unix domain).

[EACCESS]
The requested address (in the internet domain) or address (in the Unix domain) is protected, and the current user has inadequate permission to access it. Internet address port numbers less than 1024 are privileged, meaning that you can't bind to them unless you are root (or, as in the case with *rcp(1n)*, owned by root with the setuid bit on).

[EFAULT]
The *name* parameter is not in a valid part of the user address space.

[EDFSREF]
Name for a Unix domain socket references a file on a remote system (which is not allowed). If you need that capability, use an internet domain socket.

RETURN VALUE

[0] **Bind** was successful.

[—1]
Bind was unsuccessful. The error is further specified in the global *errno*.

SEE ALSO

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

NAME

brk, sbrk — change data segment size

SYNOPSIS

#include ⟨sys/types.h⟩

caddr_t brk(addr)
caddr_t addr;

newaddr = sbrk(incr)
caddr_t newaddr;
int incr;

DESCRIPTION

Brk and **sbrk** are used to change dynamically the amount of space allocated for the calling process's contiguous heap. The change is made by resetting the process's break value. The break value is the address of the first location beyond the end of the contiguous heap. The amount of allocated space increases as the break value increases. **Brk** sets the break value to *addr* (rounded up to the next multiple of the system's page size) and changes the allocated space accordingly. Locations greater than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

Sbrk adds *incr* more bytes to the break value and changes the allocated space accordingly. A pointer to the start of the new area is returned in *newaddr*.

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*(3c) for the definition of *etext*.)

DIAGNOSTICS

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

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

[ENOMEM]
The maximum possible size of a data segment, text segment or stack would be exceeded. These limits are MAXTSIZ, MAXDSIZ and MAXSSIZ, defined in ⟨machine/vmparam.h⟩.

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

RETURN VALUE

Brk returns 0 if the break could be set, otherwise it returns –1. **Sbrk** returns a pointer to the new data area in *newaddr* if the break could be set, otherwise it returns –1. Both **brk** and **sbrk** set *errno* if there is an error.

CAVEATS

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*.

SEE ALSO

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

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 searches for pathnames not beginning with "/".

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

DIAGNOSTICS

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.

[ENAMETOOLONG]
The argument *path* is too long.

[ENOASCII]
The argument *path* contains a byte with the high–order bit set.

[EACCES]
Search permission is denied for any component of the path name. If the target directory is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n).*

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

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

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

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

[EDFS_CD]
There is a fixed limit for the number of chdirs that may be made to a remote host. When this limit is exceeded this error message is returned.

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.

CAVEATS

If you **chdir** to a directory on a remote host, and are inactive for four hours, the daemon serving you will exit.

SEE ALSO

chroot(2).

NAME

chmod, fchmod — change mode of file

SYNOPSIS

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

fchmod(fd, mode)
int fd, mode;

DESCRIPTION

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

04000 set user ID on execution
02000 set group ID on execution
01000 save text image after execution
00400 read by owner
00200 write by owner
00100 execute (search on directory) by owner
00070 read, write, execute (search) by group
00007 read, write, execute (search) by others

If an executable file is set up for sharing (this is the default) then mode 1000 prevents the system from abandoning the swap–space image of the program–text portion of the file when its last user terminates. Ability to set this bit is restricted to the super–user.

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

Writing or changing the group owner of a file(see *chgrp*(1)) 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.

DIAGNOSTICS

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

[ENOASCII]
The argument *path* contains a byte with the high–order bit set.

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

[ENAMETOOLONG]
The argument *path* is too long.

[ENOENT]
The named file does not exist.

[EACCES]
Search permission is denied on a component of the path prefix. If the file is located on a remote host, this error code will be returned if

the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n).*

[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.

Fchmod will fail if:

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

[EBADF]
The descriptor is not valid.

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

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

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

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

RETURN VALUE

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

SEE ALSO

chgrp(1), chown(2), open(2).

NAME

chown, fchown — change owner and group of a file

SYNOPSIS

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

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

DESCRIPTION

The file which is named by *path* or referenced by *fd* has its *owner* and *group* changed as specified. Only the super–user may execute this call, because if users were able to give files away, they could defeat the file–space accounting procedures.

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 (see *flock*(2)).

Only one of *owner* and *group* may be set by specifying the other as —1.

DIAGNOSTICS

Chown will fail and the file will be unchanged if:

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

[ENAMETOOLONG]
The argument pathname is too long.

[ENOASCII]
The argument *path* contains a byte with the high–order bit set.

[ENOENT]
The named file does not exist.

[EPERM]
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.

[EPERM]
The effective user ID is not the super–user.

[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.

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

Fchown will fail if:

[EBADF]
Fd does not refer to a valid descriptor.

[EPERM]
The effective user ID is not the super–user.

[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.

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

RETURN VALUE

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

SEE ALSO

chmod(2), flock(2).

NAME

chroot — change root directory

SYNOPSIS

chroot(path)
char *path;

DESCRIPTION

Path is the pathname of a directory. **Chroot** causes this directory to become the root directory, the starting point for path names beginning with "/". An exception is a directory name used as an argument to **chroot**; its root is "/".

This call is restricted to the super–user.

DIAGNOSTICS

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

[EPERM]
The effective user ID is not the super–user.

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

[ENAMETOOLONG]
The argument *path* is too long.

[ENOASCII]
The argument *path* contains a byte with the high–order bit set.

[ENOENT]
The named directory does not exist.

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

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

[EDFSREF]
Reference is to remote directory which is not supported for 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 an error.

SEE ALSO

chdir(2).

NAME

close — delete a descriptor

SYNOPSIS

close(fd)
int fd;

DESCRIPTION

The **close** call deletes the descriptor *fd* 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 (see *flock*(2)).

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 forks (see *fork*(2)), 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(fd, F_SETFD, 1)" is provided which arranges that a descriptor will be closed after a successful execve; the call "fcntl(fd, F_SETFD, 0)" restores the default, which is to not close the descriptor.

DIAGNOSTICS

Close will fail if:

[EBADF]
Fd is not an active 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.

SEE ALSO

accept(2), close(2), dup(2), execve(2), fcntl(2), flock(2), fork(2), open(2), pipe(2), socket(2), socketpair(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 permanently specifies the peer to which datagrams are to be sent; if it 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.

DIAGNOSTICS

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 (for internet domain sockets) or name (for UTek domain sockets) 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 (for internet domain sockets) or name (for UTek domain sockets) is already in use.

[EFAULT]
The *name* parameter specifies an area outside the process address space.

[EWOULDBLOCK]
The socket is non–blocking and the and the connection cannot be completed immediately. It is possible to *select*(2) the socket while it is connecting by selecting it for writing.

[EDFSREF]
Name for a UTek domain socket references a file on a remote system (which is not allowed). If you need that capability, use an internet domain socket.

RETURN VALUE

[0] Successful binding or connection.

[—1]
Unsuccessful binding or connection. A more specific error code is stored in *errno*.

SEE ALSO

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

NAME

dup, dup2 — duplicate a descriptor

SYNOPSIS

newfd = dup(oldfd)
int newfd, oldfd;

newfd = dup2(oldfd, newfd)
int oldfd, newfd;

DESCRIPTION

Dup duplicates an existing object descriptor. The argument *oldfd* 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 *newfd* returned by the call is the lowest numbered descriptor which is not currently in use by the process.

The object referenced by the descriptor does not distinguish between references using *oldfd* and *newfd* in any way. Thus if *newfd* and *oldfd* 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 *newfd* 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.

DIAGNOSTICS

Dup and **dup2** fail if:

[EBADF]
Oldfd or *newfd* is not a valid active descriptor.

[EMFILE]
NOFILE (defined in ⟨sys/max.h⟩) descriptors are already active.

RETURN VALUE

Upon successful completion, **dup** and **dup2** return the new file descriptor in *newfd*. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

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

NAME

dup, dup2 — duplicate a descriptor

SYNOPSIS

newfd = dup(oldfd)
int newfd, oldfd;

newfd = dup2(oldfd, newfd)
int oldfd, newfd;

DESCRIPTION

Dup duplicates an existing object descriptor. The argument *oldfd* 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 *newfd* returned by the call is the lowest numbered descriptor which is not currently in use by the process.

The object referenced by the descriptor does not distinguish between references using *oldfd* and *newfd* in any way. Thus if *newfd* and *oldfd* 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 *newfd* 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.

DIAGNOSTICS

Dup and **dup2** fail if:

[EBADF]
Oldfd or *newfd* is not a valid active descriptor.

[EMFILE]
NOFILE (defined in <sys/max.h>) descriptors are already active.

RETURN VALUE

Upon successful completion, **dup** and **dup2** return the new file descriptor in *newfd*. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

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

NAME

execve — execute a file

SYNOPSIS

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

DESCRIPTION

Execve transforms the calling process into a new process. The new process is constructed from *path*, 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. See *a.out(5)*.

An interpreter file begins with a line of the form "#! *interpreter*". The length of this line cannot exceed SHSIZE, defined in ⟨sys/user.h⟩ (currently 32). When an interpreter file is **execve**'d, the system **execve**'s the specified interpreter. The original arguments are passed to the interpreter as one argument (arg 1) and *path* , the name of the originally **execve**'d file, is passed as an additional argument (arg 2).

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 *path*).

The argument *envp* is also an array of character pointers to null–terminated strings. These strings pass information to the new process which are not directly arguments to the command. See *environ(7)*.

Descriptors open in the calling process remain open in the new process, except for those for which the close–on–exec flag is set; see *close(2)*. 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 be undefined; see *sigvec(2)* for more information.

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 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)*
access groups	see *getgroups(2)*
working directory	see *chdir(2)*
root directory	see *chroot(2)*
control terminal	see *tty(2)*
resource usages	see *getrusage(2)*
interval timers	see *getitimer(2)*
resource limits	see *getrlimit(2)*
file mode mask	see *umask(2)*
signal mask	see *sigvec(2)*

When a "C" program is executed as a result of the call, 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(1sh)* passes an environment entry for each global shell variable defined when the program is called. See *environ(7)* for some conventionally used names.

DIAGNOSTICS

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

[ENAMETOOLONG]
The new process file's pathname is too long.

[ENOENT]
One or more components of the new process file's pathname do not exist, or the interpreter to be used to execute the new process file does not exist.

[ENOTDIR]
A component of the new process file's or the interpreter's pathname is not a directory.

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

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

[EACCES]
The new process file mode or the interpreter mode denies execute permission. If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

[ENOEXEC]
The new process file or the interpreter has the appropriate access permission, but has an invalid magic number in its header (see *a.out(5)*).

[ETXTBSY]
The new process file or the interpreter 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)*).

[E2BIG]
The number of bytes in the new process's argument list is larger than the system-imposed limit of NCARGS, defined in ⟨sys/param.h⟩.

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

[ENOEXEC]
The interpreter name is longer than SHSIZE, defined in ⟨sys/user.h⟩.

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

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

[ENOMEM]
Swap space is not available for the new process, or the new process file's textsize, datasize or stacksize exceed the system-imposed limits MAXTSIZ, MAXDSIZ or MAXSSIZ, defined in ⟨machine/vmparam.h⟩.

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

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.

CAVEATS

If a program is *setuid* to a non-super-user, but is executed when the real *uid* is "root", then the program has the powers of a super-user as well.

SEE ALSO

close(2), exit(2), fork(2), getrlimit(2), sigvec(2), execl(3c), 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; see *wait*(2).

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 (see *intro*(2)) inherits each of these processes as well.

Most "C" programs call the library routine *exit*(3c) 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(3c).

NAME

fcntl — file control

SYNOPSIS

#include <fcntl.h>

result = fcntl(fd, cmd, arg)
int result;
int fd, cmd, arg;

DESCRIPTION

Fcntl provides for control over open descriptors. The argument *fd* is an open descriptor. The value of *result* and *arg* depends on *cmd*; see below. *Cmd* is one of the following, defined in <fcntl.h>:

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 *execve*(2) system calls.

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

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

F_GETFL
Get descriptor status flags, as described below.

F_SETFL
Set descriptor status flags to *arg*. *Arg* is created by *or*'ing FNDELAY, FAPPEND and FASYNC; see below.

F_GETOWN
Get the process group currently receiving SIGIO and SIGURG signals; process groups are returned as negative values.

F_SETOWN
Set the process group to receive SIGIO and SIGURG signals. If *arg* is negative, it is interpreted as a process group number. If *arg* is positive, it is interpreted as a process ID and the associated process group is used.

The flags for the F_GETFL and F_SETFL flags are as follows, defined in <sys/file.h>:

FNDELAY
Non-blocking 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 EWOULDBLOCK.

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

FASYNC
Enable the SIGIO signal to be sent to the process group when I/O is possible (e.g., upon availability of data to be read) and enable the SIGURG signal to be sent when an exception occurs.

DIAGNOSTICS

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

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

[EINVAL]
Cmd or *arg* is an invalid value.

[EMFILE]
Cmd is F_DUPFD and NOFILE (defined in <sys/max.h>) file descriptors are currently open.

[EINVAL]
Cmd is F_DUPFD and *arg* is negative or greater than NOFILE (defined in <sys/max.h>) (see *getdtablesize*(2)).

[EINVAL]
Cmd is F_SETOWN and *arg* is not a valid process ID.

[ENOTTY]
Cmd is not a valid request for the type of object associated with *fd.*

[ENXIO]
Cmd was attempted on a special device which does not exist, or beyond the limits of the device.

[EIO]
An I/O error occurred while accessing the file system.

[ENODEV]
Fd is a device, and *cmd* is an inappropriate request for that device.

RETURN VALUE

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

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_GETFL	Value of flags, *or*'d together.

F_GETOWN	Value of file descriptor owner.
other	Value other than —1.

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

CAVEATS

The asynchronous I/O facilities of FNDELAY and FASYNC are currently available only for tty operations. No SIGIO signal is sent upon draining of output sufficiently for non–blocking writes to occur.

SEE ALSO

close(2), execve(2), getdtablesize(2), open(2), sigvec(2), execl(3c).

NAME

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

SYNOPSIS

#include ⟨sys/file.h⟩

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*. *Operation* is constructed by *or*'ing together some combination of the following, defined in ⟨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 */
```

A lock is applied by specifying either LOCK_SH or LOCK_EX, possibly *or*'d with LOCK_NB. LOCK_UN will unlock an existing lock.

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.

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.

DIAGNOSTICS

The **flock** call fails if:

[EWOULDBLOCK]	The file is locked and LOCK_NB was specified.
[EBADF]	The argument *fd* is an invalid descriptor.
[EOPNOTSUPP]	The argument *fd* refers to a socket, not to a file.
[EINVAL]	The argument *operation* is an invalid request.

RETURN VALUE

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

SEE ALSO

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

NAME

mmap — map pages of memory

SYNOPSIS

#include ⟨sys/types.h⟩
#include ⟨sys/mman.h⟩

mmap(pid, fromaddr, toaddr, len, prot, share)
int pid;
caddr_t fromaddr, toaddr;
u_int len, prot, share;

DESCRIPTION

The mapping routine **mmap** allows a process to access areas of other processes through its own address space. It causes the calling process' pages starting at *toaddr* and continuing for *len* bytes to map onto the process with id *pid*, starting at the object's pages *fromaddr*.

If *pid* is *M_SELF*, an area of the process is mapped to itself. If *pid* is *M_PHYS*, an area of the process is mapped to physical memory (in which case *share* is ignored). If *pid* is *M_ZFILL*, an area of the process is made zero filled (in which case *fromaddr* and *share* are ignored).

If the parameter *share* is true, both mappings will share the same memory. Otherwise, a *private* copy of the area is made, and changes through one mapping are not visible through the other.

PRIVATE	make a private copy for the new map
SHARED	share the area between the mappings

The parameter *prot* specifies the accessibility of the pages through the new mapping. Read and write access may be given on the basis of processes of the same user, same process group, same group, and world. A process may also protect its pages against itself. The protection for a page is specified by *or*'ing together the following values.

M_R_SELF	read, process
M_W_SELF	write, process
M_R_USER	read, user
M_W_USER	write, user
M_R_PGROUP	read, process group
M_W_PGROUP	write, process group
M_R_GROUP	read, group
M_W_GROUP	write, group
M_R_WORLD	read, world
M_W_WORLD	write, world

Note that the protection is associated with the mapping, and not with the actual memory.

If the process must change the protection of a mapping, it may map the area to itself, with the new protection. Doing this with *share* cleared will disassociate the area with all other mappings.

The *toaddr*, *fromaddr* and *len* parameters must be multiples of the system cluster size (found using the *getpagesize*(2) call).

DIAGNOSTICS

Mmap will fail when one of the following occurs:

[EINVAL]
An address is not on a cluster boundary.

[EMCOLLIDE]
Portions of the new area are already mapped.

[EMRANGE]
An area is outside the possible user's address space or includes part of the uarea.

[EACCES]
The required permissions (for reading and/or writing) are denied for the named file or area of a process.

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

[EPERM]
The area of the object to be mapped is protected against the desired operation.

RETURN VALUE

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

SEE ALSO

getpagesize(2), mremap(2), munmap(2).

NAME

fork — create a new process

SYNOPSIS

pid = fork()
int pid;

DESCRIPTION

Fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent 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, so that, 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.

The child processes resource utilizations are set to 0; see *setrlimit*(2).

DIAGNOSTICS

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

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

[EAGAIN]
The system–imposed limit on the total number of processes under execution by a single user, MAXUPRC, defined in <sys/param.h>, would be exceeded.

[ENOMEM]
Insufficient space exists in the swap area for the child process.

RETURN VALUE

Upon successful completion, **fork** returns a value of 0 in *pid* to the child process and returns the process ID of the child process in *pid* 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.

SEE ALSO

execve(2), setrlimit(2), vfork(2), wait(2).

NAME

stat, lstat, fstat — get file status

SYNOPSIS

#include ⟨sys/types.h⟩
#include ⟨sys/stat.h⟩

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

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

fstat(fd, buf)
int fd;
struct stat *buf;

DESCRIPTION

Stat obtains information about the file *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 reachable.

Lstat is like **stat** 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.

Fstat obtains the same information about an open file referenced by *fd*, 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 structure is defined in ⟨*sys/stat.h*⟩ as:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing a directory entry */
                          /* for this file */
    ino_t     st_ino;     /* this inode's number */
    u_short   st_mode;    /* file mode; see below */
    short     st_nlink;   /* number of hard links to the file */
    short     st_uid;     /* user ID of the file's owner */
    short     st_gid;     /* group ID of the file's group */
    dev_t     st_rdev;    /* ID of device -- this entry is defined only */
                          /* for character special or block special files */
    off_t     st_size;    /* total size of file */
    time_t    st_atime;   /* time of last access */
    int       st_spare1;
    time_t    st_mtime;   /* time of last data modification */
    int       st_spare2;
    time_t    st_ctime;   /* time of last file status change */
    int       st_spare3;
    long      st_blksize; /* optimal blocksize for file system I/O ops */
    long      st_blocks;  /* actual number of blocks allocated */
    long      st_hostid;  /* hostid of machine where file is located */
```

```
        long        st_spare4;
    };
```

st_atime — Time when file data was last read or modified. Changed by the following system calls: *mknod*(2), *utimes*(2), and *read*(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).

st_ctime — Time when file status was last changed. It is set both 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).

The status information word *st_mode* has these bits:

```
#define S_IFMT    0170000  /* type of file */
#define S_IFDIR   0040000  /* directory */
#define S_IFCHR   0020000  /* character special */
#define S_IFBLK   0060000  /* block special */
#define S_IFREG   0100000  /* regular */
#define S_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  /* save swapped text even after use */
#define S_IREAD   0000400  /* read permission, owner */
#define S_IWRITE  0000200  /* write permission, owner */
#define S_IEXEC   0000100  /* execute/search permission, owner */
```

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

When *fd* is associated with a pipe, **fstat** reports an ordinary file with an i-node number, restricted permissions, and a not necessarily meaningful length.

DIAGNOSTICS

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

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

[ENOASCII]
The pathname contains a character with the high-order bit set.

[ENAMETOOLONG]
The pathname is too long.

[ENOENT]
The named file does not exist.

[EACCES]
Search permission is denied for a component of the path prefix. If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

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

[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.

Fstat will fail if one of the following are true:

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

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

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

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.

CAVEATS

The fields in the stat 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 which depend on the time stamps being contiguous (in calls to *utimes*(2)).

Applying **fstat** to a socket returns a zeroed buffer.

SEE ALSO

chmod(2), chown(2), utimes(2).

NAME

fsync — synchronize a file's in-core 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-core 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.

DIAGNOSTICS

The **fsync** fails if:

[EBADF]
Fd is not a valid descriptor.

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

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.

CAVEATS

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

SEE ALSO

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

NAME

getdtablesize — get descriptor table size

SYNOPSIS

size = getdtablesize()
int size;

DESCRIPTION

Each process has a fixed size descriptor table. **Getdtablesize** returns the size of this table in *size*. The table has NOFILE slots; NOFILE is defined in <sys/max.h> and is guaranteed to be at least 20. The entries in the descriptor table are numbered with small integers starting at 0.

RETURN VALUE

Getdtablesize returns the value NOFILE in *size*.

SEE ALSO

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

NAME

getuid, geteuid — get user identity

SYNOPSIS

uid = getuid()
int uid;

euid = geteuid()
int euid;

DESCRIPTION

Getuid returns the real user ID of the current process in **uid**; **geteuid** returns the effective user ID in **euid.**

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, which use **getuid** to determine the real user ID of the process which invoked them.

RETURN VALUE

Getuid returns the real user ID; **geteuid** returns the effective user ID.

SEE ALSO

getgid(2), setreuid(2).

NAME

getgid, getegid — get group identity

SYNOPSIS

gid = getgid()
int gid;

egid = getegid()
int egid;

DESCRIPTION

Getgid returns the real group ID of the current process in **gid. Getegid** returns the effective group ID in **egid.**

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. It is for such processes that **getgid** is most useful.

RETURN VALUE

Getgid returns the real group ID of the current process in **gid. Getegid** returns the effective group ID in **egid.**

SEE ALSO

getuid(2), setregid(2), setgid(3c).

NAME

getgroups — get group access list

SYNOPSIS

#include <sys/param.h>

getgroups(ngroups, gidset)
int *ngroups, *gidset;

DESCRIPTION

Getgroups gets the current group access list of the user process and stores it in the array *gidset*. The parameter *ngroups* points to the the number of entries which may be placed in *gidset* and its contents are modified on return to indicate the actual number of groups returned. No more than NGROUPS, as defined in *<sys/param.h>*, will ever be returned in *ngroups*.

DIAGNOSTICS

The possible errors for **getgroups** are:

[EFAULT]
The argument *gidset* or *ngroups* specifies an invalid address.

[EINVAL]
The size of *gidset,* as specified by the contents of *ngroups*, is too small to accommodate the entire group access list.

RETURN VALUE

Upon successful completion, a value of 0 is returned, and *gidset* and *ngroups* are modifed as described above. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

setgroups(2), initgroups(3c).

NAME

getgroups — get group access list

SYNOPSIS

#include ⟨sys/param.h⟩

getgroups(ngroups, gidset)
int *ngroups, *gidset;

DESCRIPTION

Getgroups gets the current group access list of the user process and stores it in the array *gidset*. The parameter *ngroups* points to the the number of entries which may be placed in *gidset* and its contents are modified on return to indicate the actual number of groups returned. No more than NGROUPS, as defined in ⟨*sys/param.h*⟩, will ever be returned in *ngroups*.

DIAGNOSTICS

The possible errors for **getgroups** are:

[EFAULT]
The argument *gidset* or *ngroups* specifies an invalid address.

[EINVAL]
The size of *gidset,* as specified by the contents of *ngroups*, is too small to accommodate the entire group access list.

RETURN VALUE

Upon successful completion, a value of 0 is returned, and *gidset* and *ngroups* are modifed as described above. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

setgroups(2), initgroups(3c).

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 intended to be unique among all UTek systems in existence. 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.

RETURN VALUE

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

CAVEATS

32 bits for the identifier is too small.

SEE ALSO

hostid(1), gethostname(2).

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 super–user and is normally used only when the system is bootstrapped.

DIAGNOSTICS

The following errors may be returned by these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller was not the super–user. Applies only to *sethostname*.

RETURN VALUE

[0]	Successful call.
[—1]	Unsucessful call. An error code is placed in the global location *errno*.

CAVEATS

Host names are limited to 255 characters.

SEE ALSO

gethostid(2).

NAME

getitimer, setitimer — get/set value of interval timer

SYNOPSIS

#include ⟨sys/time.h⟩

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*⟩:

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

The **getitimer** call returns in *value* the current value for the timer specified in *which*. The **setitimer** call sets the value of the timer specified in *which* to *value*, returning the previous value of the timer in *ovalue*.

A timer value is defined by the *itimerval* structure, defined in ⟨sys/time.h⟩:

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

The *timeval* structure, defined in ⟨sys/time.h⟩, is:

```
struct  timeval {
        long            tv_sec;         /* seconds */
        long            tv_usec;        /* and microseconds */
}
```

For **getitimer**, if *it_value* is non-zero, it indicates the time to the next timer expiration. For example, if *it_value* is set to 30 seconds, then in 30 seconds the timer will expire and a SIGALRM signal will be sent to the process. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires.

For **setitimer**, setting *it_value* to non-zero sets the time to the next timer expiration. Setting *it_interval* to non-zero specifies the 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 this resolution.

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.

DIAGNOSTICS

Getitimer and **setitimer** will fail if one or more of the following are true:

[EFAULT]
The *value* parameter specifies a bad address.

[EINVAL]
Which is an invalid argument.

Setitimer will also fail if the following is true:

[EINVAL]
The *value* parameter specifies an invalid time. This could mean that either the seconds or microseconds field of the timeval structure is negative, or the seconds field is greater than 100000000 (over 3 years), or if the microseconds field is greater than 1000000 (1 sec).

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.

CAVEATS

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).

SEE ALSO

gettimeofday(2), sigvec(2).

NAME

getpagesize — get system page size

SYNOPSIS

pagesize = getpagesize()
int pagesize;

DESCRIPTION

Getpagesize returns the number of bytes in a page, (NBPG * CLSIZE), in *pagesize*. NBPG and CLSIZE are defined in ⟨machine/param.h⟩. 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.

RETURN VALUE

The number of bytes in a page is returned.

SEE ALSO

sbrk(2).

NAME

getpeername — get name of connected peer

SYNOPSIS

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).

DIAGNOSTICS

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.

RETURN VALUE

[0]	Sucessful call.
[—1]	Unsucessful call.

CAVEATS

Names bound to sockets in the UTek domain are inaccessible; **getpeername** returns a zero length name.

SEE ALSO

bind(2), socket(2), getsockname(2).

NAME

getpgrp — get process group

SYNOPSIS

pgrp = getpgrp(pid)
int pgrp;
int pid;

DESCRIPTION

Getpgrp returns the process group of the process specified by *pid* in *pgrp*. If *pid* is zero, then the call applies to the current process.

Process groups are used for distribution of signals, and by terminals to arbitrate requests for their input; processes which have the same process group as the terminal are foreground and may read, while others will block with a signal if they attempt to read.

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

DIAGNOSTICS

Getpgrp will fail if the following is true:

[ESRCH]
Pid is an invalid process ID.

RETURN VALUE

Upon successful completion, **getpgrp** returns the process group in *pgrp*. If an error occurs, —1 is returned and *errno* is set to indicate the error.

SEE ALSO

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

NAME

getpid, getppid — get process identification

SYNOPSIS

```
pid = getpid()
long pid;

ppid = getppid()
long ppid;
```

DESCRIPTION

Getpid returns the process ID of the current process in **pid.** 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 in **ppid.**

RETURN VALUE

Getpid returns the current process ID; **getppid** returns the current process' parent's process ID.

SEE ALSO

gethostid(2).

NAME

getpid, getppid — get process identification

SYNOPSIS

```
pid = getpid()
long pid;

ppid = getppid()
long ppid;
```

DESCRIPTION

Getpid returns the process ID of the current process in **pid.** 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 in **ppid.**

RETURN VALUE

Getpid returns the current process ID; **getppid** returns the current process' parent's process ID.

SEE ALSO

gethostid(2).

NAME

getpriority, setpriority — get/set program scheduling priority

SYNOPSIS

#include ⟨sys/time.h⟩
#include ⟨sys/resource.h⟩

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 returned in *prio* with the **getpriority** call and set to *prio* with the **setpriority** call.

Which is one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, defined in ⟨sys/resource.h⟩:

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

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.

A value of 0 for *who*, in either **getpriority** or **setpriority**, will indicate the operations are to apply to the current process, process group, or user.

The **getpriority** call returns in *prio* the highest priority (lowest numerical value) enjoyed by any of the specified processes. Here, *prio* will be one of 40 values in the range —20 to 19. The default priority is 0; lower priorities cause more favorable scheduling.

The **setpriority** call sets to *prio* the priorities of all of the specified processes to the specified value. Only the super–user may lower priorities.

DIAGNOSTICS

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

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

[EINVAL]
Which is 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 is located, but neither its effective nor real user ID matched the effective user ID of the caller, and the caller is not the super–user.

[EACCES]
A non super-user is attempting to change a process priority to a negative value.

RETURN VALUE

Setpriority returns 0 if there is no error, or —1 if there is, setting *errno* to indicate the error. **Getpriority** returns the process' priority. 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.

CAVEATS

If a *prio* larger than 19 is given to **setpriority,** it will be changed to 19 and the priority set accordingly.

SEE ALSO

fork(2), nice(1).

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.

Getrlimit returns the limits on the current process in the rlimit structure pointed to by *rlp*; **setrlimit** uses the values in the structure to set the process limits.

The *resource* parameter is one of the following, defined in ⟨sys/resource.h⟩:

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_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.

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, defined in ⟨sys/resource.h⟩:

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

Only the super-user may raise the hard limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

An "infinite" value for a limit is defined as RLIMIT_INFINITY (0x7fffffff) in ⟨sys/resource.h⟩.

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(1csh)*.

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.

DIAGNOSTICS

The possible errors are:

[EFAULT]
The address specified for *rlp* is invalid.

[EPERM]
The limit specified to **setrlimit** would have raised the maximum limit value, and the caller is not the super-user.

[EINVAL]
The *resource* argument is not a valid value.

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*.

SEE ALSO

csh(1csh), sh(1sh).

NAME

getrusage — get information about resource utilization

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>
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, defined in <sys/resouce.h>:

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

The buffer pointed to by *rusage* will be filled in with the following structure, defined in <sys/resource.h>:

```
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 */
};
```

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 kilobytes * seconds–of–execution and is calculated by summing the number of shared memory pages in use each time the internal system clock ticks and then averaging over 1 second intervals.

ru_idrss
: an integral value of the amount of unshared memory residing in the data segment of a process (expressed in units of kilobytes * seconds–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 kilobytes * seconds–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/O 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 input.

ru_outblock
: the number of times the file system had to perform 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.

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

DIAGNOSTICS

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

[EINVAL]
The *who* argument is an invalid value.

[EFAULT]
The argument *rusage* refers to an invalid address.

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.

CAVEATS

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

SEE ALSO

gettimeofday(2), wait(2).

NAME

getsockname — get socket name

SYNOPSIS

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

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.

RETURN VALUE

[0] **Getsockname** was successful.

[—1]
Getsockname was unsuccessful.

CAVEATS

Names bound to sockets in the UTek domain are inaccessible; **getsockname** returns a zero length name.

SEE ALSO

bind(2), socket(2).

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.

To manipulate options at the "socket" level, *level* is specified as SOL_SOCKET.

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

Optname and any specified option value are passed uninterpreted to the appropriate protocol module for interpretation. The include file *<sys/socket.h>* contains definitions for "socket" level options; see *socket*(2). At this level, *optname* is a single option; that is, only one option can be specified per call to **getsockopt** or **setsockopt**. Also, **setsockopt** will fail if an *optval* of size greater than the mbuf data length (defined in *<sys/mbuf.h>*) is specified.

DIAGNOSTICS

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 specified in **getsockopt** is not set.

[EINVAL]
Optname or *level* is unknown; size of *optval* is too large (**setsockopt**).

[EFAULT]
The options are not in a valid part of the process address space.

[ENOBUFS]
No system buffer space is available.

RETURN VALUE

[0] Successful call. In the case of **getsockopt**, the option specified by *optname* is set.

[—1]
Unsuccessful call or *optname* is not set. An error code is stored into the global variable *errno*.

CAVEATS

At present, only "socket" level options are allowed. Of these, only SO_LINGER accepts an *optval* argument of integer size to **setsockopt**.

There is no provision for resetting an option, once set.

SEE ALSO

socket(2), getprotoent(3n).

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 in the structures pointed to by *tp* and *tzp*. **Settimeofday** sets the time, using the contents of the structures.

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 ⟨*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 *tz_dsttime*, 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.

DIAGNOSTICS

Gettimeofday and **settimeofday** may set the following errors in *errno*:

[EFAULT]
An argument address references invalid memory.

In addition, **settimeofday** may set the following error:

[EPERM]
The caller is not 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 the error.

CAVEATS

Time is never correct enough to believe the microsecond values. There should a mechanism by which, at least, local clusters of systems might synchronize their clocks to millisecond granularity. If *tzp* is NULL, the time zone information will not be returned or set.

SEE ALSO

date(1), ctime(3c).

NAME

getuid, geteuid — get user identity

SYNOPSIS

uid = getuid()
int uid;

euid = geteuid()
int euid;

DESCRIPTION

Getuid returns the real user ID of the current process in **uid**; **geteuid** returns the effective user ID in **euid.**

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, which use **getuid** to determine the real user ID of the process which invoked them.

RETURN VALUE

Getuid returns the real user ID; **geteuid** returns the effective user ID.

SEE ALSO

getgid(2), setreuid(2).

NAME

ioctl — control device

SYNOPSIS

#include <sys/ioctl.h>

ioctl(fd, request, argp)
int fd;
long request;
char *argp;

DESCRIPTION

Ioctl performs a variety of functions, specified by *request*, on the open descriptor *fd*. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl** requests.

An **ioctl** *request* has encoded in it whether the argument *argp* is an "in" parameter or "out" parameter, and the size of *argp* in bytes. Macros and defines used in specifying an **ioctl** *request* are located in the file *<sys/ioctl.h>*.

The writeups of various devices in section 4 discuss how **ioctl** applies to them. See that section for details.

DIAGNOSTICS

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

[EBADF]
Fd is not a valid open descriptor, or the object it references is not readable or writable.

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

[ENOTTY]
The specified request does not apply to the kind of object which the descriptor *fd* references.

[EINVAL]
Request or *argp* is not valid.

[EFAULT]
Argp references an invalid address.

[EFAULT]
The size encoded in *request* is invalid.

RETURN VALUE

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

SEE ALSO

execve(2), fcntl(2), tty(4), intro(4n).

NAME

kill — send signal to a process

SYNOPSIS

kill(pid, sig)
int pid, sig;

DESCRIPTION

Kill sends the signal *sig* to a process specified by the process number *pid*. *Sig* may be one of the signals specified in *sigvec*(2), or it may be 0. If *sig* is 0, error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super–user. A single exception is the signal SIGCONT which may always be sent to any child or grandchild of the current process.

If *pid* is greater than 0, *sig* is is sent to the process whose process ID is equal to *pid*.

If *pid* is 0, *sig* is sent to all other processes in the sender's process group; this is a variant of *killpg*(2).

If *pid* is —1, and the user is the super–user, *sig* is broadcast universally except to system processes and the process sending the signal.

Processes may send signals to themselves.

DIAGNOSTICS

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*.

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

[EINVAL]
Pid is 0, but there is no process group associated with *pid*.

RETURN VALUE

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

SEE ALSO

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 *pgrp* 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.

DIAGNOSTICS

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 *pid*.

[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.

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.

SEE ALSO

getpgrp(2), kill(2), sigvec(2).

NAME

link — make a hard link to a file

SYNOPSIS

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

DESCRIPTION

Path1 names an existing file. *Path2* names a new directory entry to be created. **Link** creates a new link (directory entry) for the existing file, named *path2*.

With hard links, both *path1* and *path2* must be in the same file system. Unless the caller is the super–user, *path1* must not be a directory. Both *path1* and *path2* share equal access and rights to the underlying object.

DIAGNOSTICS

Link will fail and no link will be created if one or more of the following are true:

[ENOASCII]
Either pathname contains a byte with the high–order bit set.

[ENOSPC]
The directory in which the entry for the new link is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENAMETOOLONG]
Either pathname is too long.

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

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

[EEXIST]
The link named by *path2* does exist.

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

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

[EACCES]
A component of either path prefix denies search permission or the requested link requires writing in a directory with a mode that denies write permission. If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

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

[EFAULT]
One of the pathnames specified is outside the process's allocated address space.

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

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

[EDFSREF]
Both *path1* and *path2* must reference files on the same host or this error will be returned.

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

RETURN VALUE

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

SEE ALSO

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 backlog for incoming connections is specified with *listen(2)* and then the connections are accepted with *accept(2)*. The **listen** call applies only to sockets of type SOCK_STREAM.

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 will receive an error with an indication of ECONNREFUSED.

DIAGNOSTICS

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**.

RETURN VALUE

[0] **Listen** was successful.

[—1]
Listen was unsuccessful.

CAVEATS

The *backlog* is currently limited (silently) to 5.

SEE ALSO

accept(2), connect(2), socket(2).

NAME

lseek — move read/write pointer

SYNOPSIS

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

pos = lseek(fd, offset, whence)
int pos;
int fd;
off_t offset;
int whence;
```

DESCRIPTION

Lseek sets the file pointer of the file referenced by *fd*, and returns the new value of the file pointer in *pos*. *Fd* refers to a file or device open for reading and/or writing. *Whence* is one of the following values, defined in <sys/file.h>:

```
#define L_SET   0       /* set the seek pointer */
#define L_INCR  1       /* increment the seek pointer */
#define L_XTND  2       /* extend the file size */
```

The use of **offset** is described below.

Lseek sets the file pointer

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

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

If *whence* is L_XTND, 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 in *pos* . Some devices are incapable of seeking. The value of the pointer associated with such a device is undefined.

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

DIAGNOSTICS

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

[EBADF]
Fd is not an open file descriptor.

[ESPIPE]
Fd is associated with a pipe or a socket.

[EINVAL]
Whence is not a proper value.

[EINVAL]
The new pointer would be negative.

RETURN VALUE

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

CAVEATS

This document's use of *whence* is incorrect English, but maintained for historical reasons.

SEE ALSO

dup(2), open(2).

NAME

stat, lstat, fstat — get file status

SYNOPSIS

#include ⟨sys/types.h⟩
#include ⟨sys/stat.h⟩

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

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

fstat(fd, buf)
int fd;
struct stat *buf;

DESCRIPTION

Stat obtains information about the file *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 reachable.

Lstat is like **stat** 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.

Fstat obtains the same information about an open file referenced by *fd*, 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 structure is defined in ⟨*sys/stat.h*⟩ as:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing a directory entry */
                          /* for this file */
    ino_t     st_ino;     /* this inode's number */
    u_short   st_mode;    /* file mode; see below */
    short     st_nlink;   /* number of hard links to the file */
    short     st_uid;     /* user ID of the file's owner */
    short     st_gid;     /* group ID of the file's group */
    dev_t     st_rdev;    /* ID of device -- this entry is defined only */
                          /* for character special or block special files *
    off_t     st_size;    /* total size of file */
    time_t    st_atime;   /* time of last access */
    int       st_spare1;
    time_t    st_mtime;   /* time of last data modification */
    int       st_spare2;
    time_t    st_ctime;   /* time of last file status change */
    int       st_spare3;
    long      st_blksize; /* optimal blocksize for file system I/O ops */
    long      st_blocks;  /* actual number of blocks allocated */
    long      st_hostid;  /* hostid of machine where file is located */
```

```
        long        st_spare4;
    };
```

st_atime — Time when file data was last read or modified. Changed by the following system calls: *mknod*(2), *utimes*(2), and *read*(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).

st_ctime — Time when file status was last changed. It is set both 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).

The status information word *st_mode* has these bits:

```
#define S_IFMT   0170000   /* type of file */
#define S_IFDIR  0040000   /* directory */
#define S_IFCHR  0020000   /* character special */
#define S_IFBLK  0060000   /* block special */
#define S_IFREG  0100000   /* regular */
#define S_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   /* save swapped text even after use */
#define S_IREAD  0000400   /* read permission, owner */
#define S_IWRITE 0000200   /* write permission, owner */
#define S_IEXEC  0000100   /* execute/search permission, owner */
```

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

When *fd* is associated with a pipe, **fstat** reports an ordinary file with an i-node number, restricted permissions, and a not necessarily meaningful length.

DIAGNOSTICS

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

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

[ENOASCII]
The pathname contains a character with the high-order bit set.

[ENAMETOOLONG]
The pathname is too long.

[ENOENT]
The named file does not exist.

[EACCES]
Search permission is denied for a component of the path prefix. If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

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

[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.

Fstat will fail if one of the following are true:

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

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

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

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.

CAVEATS

The fields in the stat 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 which depend on the time stamps being contiguous (in calls to *utimes*(2)).

Applying **fstat** to a socket returns a zeroed buffer.

SEE ALSO

chmod(2), chown(2), utimes(2).

NAME

mkdir — make a directory file

SYNOPSIS

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

DESCRIPTION

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

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

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

DIAGNOSTICS

Mkdir will fail and no directory will be created if:

[ENOASCII]
The *path* argument contains a byte with the high-order bit set.

[ENAMETOOLONG]
The argument *path* is too long.

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

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

[EACCES]
You do not have write permission in the directory in which you want to create the new directory, or you do not have search permission in one of the components of the path prefix. If the directory is to be created on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

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

[EEXIST]
The named directory *path* already exists.

[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.

[ENOSPC]
The file system is out of inodes.

[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]
The new directory cannot be created because there is no space left on the file system which will contain the directory.

[ENFILE]
The system inode table is full.

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

RETURN VALUE

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

SEE ALSO

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

NAME

mknod — make a special file

SYNOPSIS

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

DESCRIPTION

Mknod creates a new special file whose name is *path*. The mode of the new file (including special file bits) is initialized from *mode*. (The protection part of the mode is modified by the process's mode mask; see *umask*(2)). The first block pointer of the i–node is initialized from *dev* and is used to specify which device the special file refers to.

For a list of modes, see *stat*(2).

If mode indicates a block or character special file, *dev* is a configuration–dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

Mknod may be invoked only by the super–user.

DIAGNOSTICS

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

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

[ENOASCII]
The pathname contains a character with the high–order bit set.

[ENAMETOOLONG]
The argument *path* is too long.

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

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

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

[EEXIST]
The named file already exists.

[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.

[EACCES]
Search permission is denied for any component of the path name. If the target directory is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n).*

[ENFILE]
The system inode table is full.

[ENOSPC]
The file system is out of inodes.

[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.

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

RETURN VALUE

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

SEE ALSO

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

NAME

mmap, fmap — map pages of memory

SYNOPSIS

#include ⟨sys/types.h⟩
#include ⟨sys/mman.h⟩

mmap(pid, fromaddr, toaddr, len, prot, share)
int pid;
caddr_t fromaddr, toaddr;
u_int len, prot, share;

DESCRIPTION

The mapping routine **mmap** allows a process to access areas of other processes through its own address space. It causes the calling process' pages starting at *toaddr* and continuing for *len* bytes to map onto the process with id *pid*, starting at the object's pages *fromaddr*.

If *pid* is *M_SELF*, an area of the process is mapped to itself. If *pid* is *M_PHYS*, an area of the process is mapped to physical memory (in which case *share* is ignored). If *pid* is *M_ZFILL*, an area of the process is made zero filled (in which case *fromaddr* and *share* are ignored).

If the parameter *share* is true, both mappings will share the same memory. Otherwise, a *private* copy of the area is made, and changes through one mapping are not visible through the other.

PRIVATE	make a private copy for the new ma
SHARED	share the area between the mappin

The parameter *prot* specifies the accessibility of the pages through the new mapping. Read and write access may be given on the basis of processes of the same user, same process group, same group, and world. A process may also protect its pages against itself. The protection for a page is specified by *or*'ing together the following values.

M_R_SELF	read, process
M_W_SELF	write, process
M_R_USER	read, user
M_W_USER	write, user
M_R_PGROUP	read, process group
M_W_PGROUP	write, process group
M_R_GROUP	read, group
M_W_GROUP	write, group
M_R_WORLD	read, world
M_W_WORLD	write, world

Note that the protection is associated with the mapping, and not with the actual memory.

If the process must change the protection of a mapping, it may map the area to itself, with the new protection. Doing this with *share* cleared will disassociate the area with all other mappings.

The *toaddr*, *fromaddr* and *len* parameters must be multiples of the system cluster size (found using the *getpagesize*(2) call).

DIAGNOSTICS

Mmap will fail when one of the following occurs:

[EINVAL]
An address is not on a cluster boundary.

[EMCOLLIDE]
Portions of the new area are already mapped.

[EMRANGE]
An area is outside the possible user's address space or includes part of the uarea.

[EACCES]
The required permissions (for reading and/or writing) are denied for the named file or area of a process.

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

[EPERM]
The area of the object to be mapped is protected against the desired operation.

RETURN VALUE

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

SEE ALSO

getpagesize(2), mremap(2), munmap(2).

NAME

mount, umount — mount or remove file system

SYNOPSIS

mount(special, path, rwflag)
char *special, *path;
int rwflag;

umount(special)
char *special;

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block special file *special*. After successful completion, references to file *path* will refer to the root file on the newly mounted file system. *Special* and *path* are pointers to null–terminated strings containing the appropriate pathnames.

Path must exist already. *Path* must be a directory. Its old contents are inaccessible while the file system is mounted.

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

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

These calls are limited to the super–user.

DIAGNOSTICS

Mount and **umount** will fail when one of the following occurs:

[EPERM]
The caller is not the super–user.

[ENOENT]
Special or *path* does not exist.

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

[ENAMETOOLONG]
The argument *special* or *path* is too long.

[ENOTBLK]
Special is not a block device.

[ENXIO]
The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).

[ENOASCII]
The pathname *special* or *path* contains a character with the high–order bit set.

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

[EDFSREF]
Path may not reference a file system on another host.

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

In addition, **mount** will fail when one or more of the following occurs:

[ENOTDIR]
Path is not a directory.

[EBUSY]
Another process currently holds a reference to *path.*

[ENOMEM]
No space remains in the mount table.

[EINVAL]
The super block for the file system has a bad magic number or an out–of–range block size.

[ENOMEM]
Not enough memory is available to read the cylinder group information for the file system.

[EIO]
An I/O error occurred while reading the super block or cylinder group information.

[EIO]
An I/O error occurred while accessing the device.

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

[EFAULT]
Special or *path* points outside the process's allocated address space.

In addition, **umount** will fail when one or more of the following occurs:

[EINVAL]
The requested device is not in the mount table.

[EBUSY]
A process is holding a reference to a file located on the file system.

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

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

RETURN VALUE

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

SEE ALSO

mount(8), umount(8).

NAME

mremap — remap pages of memory

SYNOPSIS

#include ⟨sys/types.h⟩
#include ⟨sys/mman.h⟩

mremap(fromaddr, toaddr, len, prot)
caddr_t fromaddr, toaddr;
u_int len, prot;

DESCRIPTION

Mremap causes the process pages starting at *fromaddr* and continuing for *len* bytes to be mapped to the address *toaddr*. The parameter *prot* specifies the accessibility of the newly mapped pages (see *mmap*(2)).

The *fromaddr*, *toaddr*, and *len* parameters must be multiples of the system page size (obtained with the *getpagesize*(2) call), which may be larger than the underlying hardware page size.

DIAGNOSTICS

[EINVAL]
An address is not on a cluster boundary.

[EPERM]
The area of the object to be mapped is protected against the desired operation.

[EMCOLLIDE]
Portions of the new area are already mapped. This check is made as if the old area were gone, so overlapping moves work.

[EMRANGE]
An area is outside the possible user's address space or includes part of the uarea.

RETURN VALUE

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

SEE ALSO

fmap(2), getpagesize(2), mmap(2), mremap(2), munmap(2).

NAME

mstat — find out about process clusters

SYNOPSIS

#include <sys/types.h>

mstat(type, addr, len, vec)
caddr_t addr;
int len;
char *vec;

DESCRIPTION

N.B.: **Mstat** will not be implemented in the first release of the system.

Mstat returns information, in the character array *vec*, describing the process clusters beginning at *addr* and continuing for *len* bytes. *Type* is one of the following:

M_S_INCORE	in core
M_S_LOCKED	locked into core
M_S_MAPPED	mapped
M_S_ADVISED	the advised strategy

Each entry in *vec* corresponds to a single cluster.

DIAGNOSTICS

[EFAULT]
Part of *vec* lies outside of the process' allocated address space.

RETURN VALUE

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

SEE ALSO

getpagesize(2), fmap(2), madvise(2), mmap(2), munmap(2).

NAME

munmap — unmap pages of memory

SYNOPSIS

#include ⟨sys/type.h⟩
#include ⟨sys/mman.h⟩

munmap(addr, len)
caddr_t addr;
u_int len;

DESCRIPTION

Munmap causes the process pages starting at *addr* and continuing for *len* bytes to be removed from the legal address space of the process.

DIAGNOSTICS

[EINVAL]
addr is not on a cluster boundary or *len* is not a multiple of the pagesize.

[EMRANGE]
The area to be unmapped is outside the possible user's address space or includes part of the uarea.

RETURN VALUE

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

SEE ALSO

fmap(2), getpagesize(2), mmap(2), mremap(2).

NAME

open — open a file for reading or writing, or create a new file

SYNOPSIS

#include ⟨sys/file.h⟩

fd = open(path, flags, mode)
int fd;
char *path;
int flags, mode;

DESCRIPTION

Open opens the file named by *path* as specified by the *flags* argument and returns a descriptor for that file in *fd*.

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 mode *mode* as described in *chmod*(2) and modified by the process' umask value (see *umask*(2)).

Path is a null-terminated pathname (the address of a string of ASCII characters representing a pathname, terminated by a null character). *Flags* is constructed by *or*'ing the following values, defined in ⟨sys/file.h⟩:

O_RDONLY
Open for reading only.

O_WRONLY
Open for writing only.

O_RDWR
Open for reading and writing.

O_NDELAY
Do not block on open.

If the open call would result in the process being blocked for some reason (e.g., waiting for carrier on a dialup line), the open returns immediately.

O_APPEND
Append on each write.

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

O_CREAT
Create file if it does not exist.

O_TRUNC
Truncate size to 0.

If the file exists, it is truncated to zero length.

O_EXCL
Error if create and file exists.

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

Upon successful completion a non–negative integer *fd*, termed a file descriptor, is returned. The file pointer used to mark the current position within the file is set to the beginning of the file.

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

There is a limit on the number of file descriptors a process may have open simultaneously. This number is NOFILE, defined in ⟨sys/max.h⟩. The *getdtablesize(2)* call returns the current value of NOFILE.

DIAGNOSTICS

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

[ENAMETOOLONG]
The argument *path* is too long.

[ENOASCII]
The pathname contains a character with the high–order bit set.

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

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

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

[EACCES]
The required permissions (for reading and/or writing) are denied for the named flag. If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

[EISDIR]
The named file is a directory, and the arguments specify it is to be opened for writing.

[EROFS]
The named file resides on a read–only file system, and the file is to be modified.

[EMFILE]
NOFILE files are currently open (see *getdtablesize*(2)).

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

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

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

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

[EEXIST]
O_EXCL is specified and the file exists.

[ENOSPC]
O_CREAT is specified, and the file system is out of inodes.

[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, the file does not exist and O_CREAT is specified.

[ENFILE]
O_CREAT is specified, and the system inode table is full.

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

[ENXIO]
The O_NDELAY flag is given, and the file is a communications device on which their is no carrier present.

[EBUSY]
An exclusively–opened port is already opened.

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

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

RETURN VALUE

If no error occurred, *open* returns the file descriptor in *fd*. Otherwise, —1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), close(2), dup(2), getdtablesize(2), lseek(2), read(2), write(2), umask(2), unlink(2).

NAME

pipe — create an interprocess communication channel

SYNOPSIS

pipe(fd)
int fd[2];

DESCRIPTION

The **pipe** system call creates an I/O mechanism called a pipe. **Pipe** returns two file descriptors in *fd*[]. *Fd*[0] is opened for reading, and *fd*[1] is opened for writing. When the pipe is written using *fd*[1] up to MINBSIZE (defined in ⟨sys/fs.h⟩) bytes of data are buffered before the writing process is blocked. A read using *fd*[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. See *sh(1sh)*.

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

Pipes are really a special case of the *socketpair(2)* call and, in fact, are implemented as such in the system.

A signal (SIGPIPE) is generated if a *write* on a pipe with only one end is attempted.

DIAGNOSTICS

The **pipe** call will fail if:

[EMFILE]
More than NOFILE − 2 (defined in ⟨sys/max.h⟩) descriptors are already open in this process.

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

[ENFILE]
The system file table is full.

[ENOBUF]
No buffer space is available for the pipe.

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.

CAVEATS

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

SEE ALSO

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

NAME

profil — execution time profile

SYNOPSIS

profil(buf, bufsiz, offset, scale)
char *buf;
int bufsiz, offset, scale;

DESCRIPTION

Buf 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 a word inside *buf*, that word is incremented.

The scale is interpreted as an unsigned, fixed–point fraction with binary point at the left: 0x10000 gives a 1–1 mapping of pc's to words in *buf*; 0x8000 maps each pair of instruction words together. 0x2 maps all instructions onto the beginning of *buf* (producing a non–interrupting core clock).

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

RETURN VALUE

A 0, indicating success, is always returned.

SEE ALSO

gprof(1), setitimer(2), monitor(3c).

NAME

ptrace — process trace

SYNOPSIS

#include <signal.h>

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

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. The child process must be started by using *exect* (see *execl(3c)*).

Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process.

A process being traced behaves normally until it encounters some signal whether internally generated like "illegal instruction" or externally generated like "interrupt". See *sigvec(2)* for the list. Then the traced process enters a stopped state and its parent is notified via *wait(2)*. When the child is in the stopped state, its core image can be examined and modified using **ptrace**. If desired, another **ptrace** request can then cause the child either to terminate or to continue, possibly ignoring the signal.

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

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

1,2 The word in the child process's address space at *addr* is returned. If I and D space are separated (e.g. historically on a pdp-11), request 1 indicates I space, 2 D space. *Addr* must be even. The child must be stopped. The input *data* is ignored.

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

4,5 The given *data* is written at the word in the process's address space corresponding to *addr,* which must be even. No useful value is returned. If I and D space are separated, request 4 indicates I space, 5 D space. Attempts to write in pure procedure fail if another process is executing the same file.

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

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

8 The traced process terminates.

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. This is part of the mechanism for implementing breakpoints.

10 A memory breakpoint is modified. *Addr* is the breakpoint address. *Data* is a bit mask that is used to determine what kind of memory breakpoint action to take. Bit 0 will force a breakpoint if *addr* is written, bit 1 will force a breakpoint if *addr* is read, and bit 2 determines which breakpoint register to use. (If bit 2 is set, breakpoint register 1 (BPR1) is used; otherwise breakpoint register 0 (BPR0) is used.) The breakpoint is removed if neither bit 0 or bit 1 is set.

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

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

DIAGNOSTICS

[ESRCH]
The specified process does not exist.

[EPERM]
The specified process cannot be traced.

[EIO]
Request is an invalid argument.

[EIO]
An I/O error occurred while performing the requested action.

RETURN VALUE

A 0 value is returned if the call succeeds. If the call fails then a —1 is returned and the global variable *errno* is set to indicate the error.

CAVEATS

Ptrace is unique and arcane; it should be replaced with a special file which can be opened and read and written. The control functions could then be implemented with *ioctl(2)* calls on this file. This would be simpler to understand and have much higher performance.

The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use "illegal instruction" signals at a very high rate) could be efficiently debugged.

The error indication, —1, is a legitimate function value; *errno,* see *intro(2)*, can be used to disambiguate.

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

SEE ALSO

adb(1), sigvec(2), wait(2), execl(3c).

NAME

read, readv — read input

SYNOPSIS

cc = read(fd, buf, nbytes)
int cc, fd;
char *buf;
int nbytes;

#include ⟨sys/types.h⟩
#include ⟨sys/uio.h⟩

cc = readv(fd, iov, iovcnt)
int cc, fd;
struct iovec *iov;
int iovcnt;

DESCRIPTION

Read attempts to read *nbytes* of data from the object referenced by the descriptor *fd* into the buffer pointed to by *buf*. **Readv** performs the same action, but scatters the input data into *iovcnt* buffers specified by the members of the *iovec* array: iov[0], iov[1], ..., iov[iovcnt—1].

Readv is not supported for raw devices (e.g. raw disks, terminals) nor is it supported for reading files on remote hosts.

Read and **readv** return in *cc* the number of bytes read.

For **readv**, the *iovec* structure is defined in ⟨sys/uio.h⟩ 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 **read** starts at a position given by the pointer associated with *fd*, 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 a object is undefined.

Upon successful completion, **read** and **readv** return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a file which has that many bytes left before the end-of-file, but in no other cases.

If *cc* = 0, then end-of-file has been reached.

DIAGNOSTICS

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

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

[EFAULT]
Buf points outside the allocated address space.

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

[ENOBUFS]
Fd is a socket, and the system lacks sufficient buffer space to do the **read.**

[ENOTCONN]
Fd is a socket which is not connected.

[EWOULDBLOCK]
Fd is in non–blocking mode, and doing the **read** would cause a process to block.

In addition, **readv** may return one of the following errors:

[EINVAL]
Iovcnt is less than or equal to 0, or greater than 16.

[EINVAL]
One of the *iov_len* values in the *iov* array is negative.

[EINVAL]
The sum of the *iov_len* values in the *iov* array overflowed a 32–bit integer.

[ENXIO]
Readv was attempted on an unsupported raw device (see above).

[EDFSNOBUF]
Malloc failed on remote system; try smaller (8k or less) read.

RETURN VALUE

If successful, the number of bytes actually read is returned in *cc.* Otherwise, a —1 is returned and the global variable *errno* is set to indicate the error.

SEE ALSO

dup(2), lseek(2), open(2), pipe(2), socket(2), socketpair(2).

NAME

readcbcs — read compute engine configuration switch

SYNOPSIS

switch = readcbcs()
int switch;

DESCRIPTION

Readcbcs is a system call specific to the 6100 series workstations. It returns the value of the 8 bit computer board configuration dipswitch located at the rear of the workstation. To use this system call the argument **—l61** with *cc(1).*

FILES

/usr/lib/lib61.a

CAVEATS

Care should be taken when modifying the setting of this switch as it is read at powerup and boot.

SEE ALSO

writecbd(2).

REFERENCES

6130 System User's Guide section 2 for switch settings.

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 *path* in the buffer *buf* which has size *bufsiz*. *Cc*, the number of characters copied into *buf*, is returned. Only *bufsiz* bytes are copied; if the contents of the link named by *path* are longer than *bufsiz*, *buf* will contain a truncated copy of the contents of the symbolic link.

DIAGNOSTICS

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

[ENOASCII]
The *path* argument contains a byte 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.

[EACCES]
Search permission is denied on a component of the path prefix. If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

[EINVAL]
The named file *path* 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.

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

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

RETURN VALUE

The call returns in *cc* 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*.

SEE ALSO

lstat(2), stat(2), symlink(2).

NAME

read, readv — read input

SYNOPSIS

```
cc = read(fd, buf, nbytes)
int cc, fd;
char *buf;
int nbytes;

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

cc = readv(fd, iov, iovcnt)
int cc, fd;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

Read attempts to read *nbytes* of data from the object referenced by the descriptor *fd* into the buffer pointed to by *buf*. **Readv** performs the same action, but scatters the input data into *iovcnt* buffers specified by the members of the *iovec* array: iov[0], iov[1], ..., iov[iovcnt—1].

Readv is not supported for raw devices (e.g. raw disks, terminals) nor is it supported for reading files on remote hosts.

Read and **readv** return in *cc* the number of bytes read.

For **readv**, the *iovec* structure is defined in <sys/uio.h> 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 **read** starts at a position given by the pointer associated with *fd*, 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 a object is undefined.

Upon successful completion, **read** and **readv** return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a file which has that many bytes left before the end–of–file, but in no other cases.

If *cc* = 0, then end–of–file has been reached.

DIAGNOSTICS

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

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

[EFAULT]
Buf points outside the allocated address space.

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

[ENOBUFS]
Fd is a socket, and the system lacks sufficient buffer space to do the **read.**

[ENOTCONN]
Fd is a socket which is not connected.

[EWOULDBLOCK]
Fd is in non–blocking mode, and doing the **read** would cause a process to block.

In addition, **readv** may return one of the following errors:

[EINVAL]
Iovcnt is less than or equal to 0, or greater than 16.

[EINVAL]
One of the *iov_len* values in the *iov* array is negative.

[EINVAL]
The sum of the *iov_len* values in the *iov* array overflowed a 32–bit integer.

[ENXIO]
Readv was attempted on an unsupported raw device (see above).

[EDFSNOBUF]
Malloc failed on remote system; try smaller (8k or less) read.

RETURN VALUE

If successful, the number of bytes actually read is returned in *cc.* Otherwise, a —1 is returned and the global variable *errno* is set to indicate the error.

SEE ALSO

dup(2), lseek(2), open(2), pipe(2), socket(2), socketpair(2).

NAME

reboot — reboot system or halt processor

SYNOPSIS

#include <sys/reboot.h>

reboot(howto)
int howto;

DESCRIPTION

N.B.: **Reboot** is not implemented with the first release of the system. Its interface, as documented here, is likely to change when it becomes available.

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" in the root file system of unit 0 of a disk chosen in a processor–specific way. An automatic consistency check of the disks is then normally performed.

The bits of *howto*, defined in <sys/reboot.h>, 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 "xx(0,0)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 superuser may **reboot** a machine.

DIAGNOSTICS

[EPERM]
: The caller is not the superuser.

RETURN VALUE

If successful, this call never returns. Otherwise, a —1 is returned and an error is returned in the global variable *errno*.

SEE ALSO

crash(8), halt(8), init(8), reboot(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 may be 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 send call is formed by **or**'ing one or more of the values,

```
#define MSG_PEEK   0x1        /* peek at incoming message */
#define MSG_OOB        0x2    /* process out-of-band data */
```

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  iov *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). Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*.

DIAGNOSTICS

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.

RETURN VALUE

These calls return the number of bytes received, or —1 if an error occurred.

SEE ALSO

read(2), send(2), socket(2).

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 may be 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 send call is formed by **or**'ing one or more of the values,

```
#define MSG_PEEK   0x1      /* peek at incoming message */
#define MSG_OOB    0x2      /* process out-of-band data */
```

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  iov *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). Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*.

DIAGNOSTICS

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.

RETURN VALUE

These calls return the number of bytes received, or —1 if an error occurred.

SEE ALSO

read(2), send(2), socket(2).

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 may be 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 send call is formed by **or**'ing one or more of the values,

```
#define MSG_PEEK  0x1     /* peek at incoming message */
#define MSG_OOB   0x2     /* process out–of–band data */
```

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  iov *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). Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*.

DIAGNOSTICS

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.

RETURN VALUE

These calls return the number of bytes received, or —1 if an error occurred.

SEE ALSO

read(2), send(2), socket(2).

NAME

rename — change the name of a file

SYNOPSIS

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

DESCRIPTION

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

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

DIAGNOSTICS

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

[ENOASCII]
Either pathname contains a byte with the high–order bit set.

[ENAMETOOLONG]
The argument *from* or *to* is too long.

[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.

[ENOSPC]
The directory in which the entry for the new name is being placed cannot be extended because there is no space left on the file system containing the directory.

[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.

[EINVAL]
From is "." or "..", or the parent of *from* is the same as *from*.

[ENOTEMPTY]
To exists, and is a non–empty directory.

[ENOTDIR]
From is not a directory, but *to* is.

[EISDIR]
From is a directory, but *to* is not.

[EEXIST]
From is an ancestor of *to* (allowing this would make *to* the ancestor of *from* and would make a loop).

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

[EACCES]
The requested link requires writing in a directory with a mode that denies write permission. If the *from* is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access.5n).*

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

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

[EIO]
An I/O error occurred while accessing the file system.

[EDFSREF]
Both *from* and *to* must reference files on the same host or this error will be returned.

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

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.

CAVEATS

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", say "a/foo", being a hard link to directory "b", and an entry in directory "b", say "b/bar", 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.

SEE ALSO

open(2).

NAME

rmdir — remove a directory file

SYNOPSIS

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 "..".

DIAGNOSTICS

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.

[ENOASCII]
The pathname 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.

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

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

[EACCES]
If the directory is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

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

[EINVAL]
Path is ".".

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

[EDFSNOSUCHHOST]
The pathname referenced a remote directory, but when we broadcast a request for its address, no host responded.

RETURN VALUE

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

SEE ALSO

mkdir(2), unlink(2).

NAME

rw_nvram — read or write the contents of a nonvolatile memory

SYNOPSIS

#include <machine/nvram.h>

rw_nvram(nvram_number, rw, bufp, size)
int nvram_number, rw;
u_char *bufp;
int size;

DESCRIPTION

Rw_nvram reads or writes *size* bytes of the nonvolatile memory (nvram) specified by *nvram_number* from/to the buffer at *bufp*. The first byte of the nvram is always an 8 bit checksum. The checksum value provided by the user on a write is ignored. The second byte of the nvram is a version number used to interpret the rest of the contents of the nvram.

A value of NV_GLOBAL in *nvram_number* selects the global system nvram. Other nvrams are identified by the slot number of the board on which they reside. If *rw* is NV_READ, the nvram is to be read. A value of NV_WRITE indicates the nvram is to be written.

DIAGNOSTICS

Rw_nvram will fail when one of the following occurs:

[EBADF]
The nvram specified by *nvram_number* does not exist.

[EFAULT]
Bufp points outside the allocated address space.

[ENOTTY]
The function requested in *rw* is not valid.

[EINVAL]
Size is negative or larger than the size of the nvram.

[EPERM]
The caller requesting a write operation is not the super–user.

[EIO]
An I/O error occurred while accessing the device.

RETURN VALUE

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

NAME

brk, sbrk — change data segment size

SYNOPSIS

#include <sys/types.h>

caddr_t brk(addr)
caddr_t addr;

newaddr = sbrk(incr)
caddr_t newaddr;
int incr;

DESCRIPTION

Brk and **sbrk** are used to change dynamically the amount of space allocated for the calling process's contiguous heap. The change is made by resetting the process's break value. The break value is the address of the first location beyond the end of the contiguous heap. The amount of allocated space increases as the break value increases. **Brk** sets the break value to *addr* (rounded up to the next multiple of the system's page size) and changes the allocated space accordingly. Locations greater than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

Sbrk adds *incr* more bytes to the break value and changes the allocated space accordingly. A pointer to the start of the new area is returned in *newaddr*.

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*(3c) for the definition of *etext*.)

DIAGNOSTICS

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

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

[ENOMEM]
The maximum possible size of a data segment, text segment or stack would be exceeded. These limits are MAXTSIZ, MAXDSIZ and MAXSSIZ, defined in <machine/vmparam.h>.

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

RETURN VALUE

Brk returns 0 if the break could be set, otherwise it returns –1. **Sbrk** returns a pointer to the new data area in *newaddr* if the break could be set, otherwise it returns –1. Both **brk** and **sbrk** set *errno* if there is an error.

CAVEATS

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*.

SEE ALSO

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

NAME

select — synchronous I/O multiplexing

SYNOPSIS

#include <sys/time.h>

nfound = select(nfd, readfd, writefd, execptfd, timeout)
int nfound, nfd;
unsigned long readfd[], writefd[], execptfd[];
struct timeval *timeout;

DESCRIPTION

Select examines the I/O descriptors specified by the arrays of bit masks *readfd*, *writefd*, and *execptfd* to see if they are ready for reading, writing, or have an exceptional condition pending, respectively. These mask arrays must be at least as long as "(*nfd* + 31)/32", or one element for every 32 file descriptors. File descriptor *f* is represented in the mask by:

mask[f/32] |= 1 << (f % 32)

Nfd descriptors are checked, i.e. the bits from 0 through *nfd*–1 in the masks are examined. **Select** returns, in place, a mask of those descriptors which are ready. The total number of ready descriptors is returned in *nfound*.

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 *readfd*, *writefd*, and *execptfd* may be given as 0 if no descriptors are of interest.

DIAGNOSTICS

An error return from **select** indicates:

[EBADF]
One of the bit masks specifies an invalid descriptor.

[EINTR]
An signal was delivered before any of the selected–for events occurred or the time limit expired.

[EINVAL]
Timeout does not point to a reasonable value.

[EFAULT]
An argument specifies an invalid address.

RETURN VALUE

Select returns the number of descriptors contained in the bit masks, or —1 if an error occurred. If the time limit expires then **select** returns 0.

CAVEATS

The descriptor masks (up to "(*nfd* + 31)/32") are always modified on return, even if the call returns as the result of the timeout.

The return value of *mask* for descriptors greater than specified by *nfd* is undefined. In other words, the bits specified by

mask [f/32] & (1 << (f % 32))

where *f* is greater than or equal to *nfd* should not be assumed to have any meaningful value.

The magic constant *32* mentioned above is the number of bits in a **long.**

SEE ALSO

accept(2), connect(2), read(2), recv(2), send(2), write(2).

NAME

send, sendto, sendmsg — send a message from 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** although 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 be set to MSG_OOB to send "out-of-band" data on sockets which support this notion (e.g. SOCK_STREAM).

See *recv(2)* for a description of the *msghdr* structure.

DIAGNOSTICS

[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.

RETURN VALUE

The call returns the number of characters sent, or —1 if a local error occurred.

SEE ALSO

recv(2), socket(2).

NAME

send, sendto, sendmsg — send a message from 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** although 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 be set to MSG_OOB to send "out–of–band" data on sockets which support this notion (e.g. SOCK_STREAM).

See *recv(2)* for a description of the *msghdr* structure.

DIAGNOSTICS

[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.

RETURN VALUE

The call returns the number of characters sent, or —1 if a local error occurred.

SEE ALSO

recv(2), socket(2).

NAME

send, sendto, sendmsg — send a message from 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** although 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 be set to MSG_OOB to send "out–of–band" data on sockets which support this notion (e.g. SOCK_STREAM).

See *recv(2)* for a description of the *msghdr* structure.

DIAGNOSTICS

[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.

RETURN VALUE

The call returns the number of characters sent, or —1 if a local error occurred.

SEE ALSO

recv(2), socket(2).

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 NGRPS, as defined in *⟨sys/param.h⟩*.

Only the super-user may set new groups.

DIAGNOSTICS

The **setgroups** call will fail if:

[EPERM]
The caller is not the super-user.

[EFAULT]
The address specified for *gidset* is outside the process address space.

[EINVAL]
Ngroups is too large a value.

RETURN VALUE

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

SEE ALSO

getgroups(2), initgroups(3c).

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 UTek systems in existence. 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.

RETURN VALUE

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

CAVEATS

32 bits for the identifier is too small.

SEE ALSO

hostid(1), gethostname(2).

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 super–user and is normally used only when the system is bootstrapped.

DIAGNOSTICS

The following errors may be returned by these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller was not the super–user. Applies only to *sethostname.*

RETURN VALUE

[0]	Successful call.
[—1]	Unsucessful call. An error code is placed in the global location *errno.*

CAVEATS

Host names are limited to 255 characters.

SEE ALSO

gethostid(2).

NAME

getitimer, setitimer — get/set value of interval timer

SYNOPSIS

#include ⟨sys/time.h⟩

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*⟩:

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

The **getitimer** call returns in *value* the current value for the timer specified in *which*. The **setitimer** call sets the value of the timer specified in *which* to *value*, returning the previous value of the timer in *ovalue*.

A timer value is defined by the *itimerval* structure, defined in ⟨sys/time.h⟩:

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

The *timeval* structure, defined in ⟨sys/time.h⟩, is:

```
struct  timeval {
        long            tv_sec;         /* seconds */
        long            tv_usec;        /* and microseconds */
}
```

For **getitimer**, if *it_value* is non–zero, it indicates the time to the next timer expiration. For example, if *it_value* is set to 30 seconds, then in 30 seconds the timer will expire and a SIGALRM signal will be sent to the process. If *it_interval* is non–zero, it specifies a value to be used in reloading *it_value* when the timer expires.

For **setitimer**, setting *it_value* to non–zero sets the time to the next timer expiration. Setting *it_interval* to non–zero specifies the 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 this resolution.

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.

DIAGNOSTICS

Getitimer and **setitimer** will fail if one or more of the following are true:

[EFAULT]
The *value* parameter specifies a bad address.

[EINVAL]
Which is an invalid argument.

Setitimer will also fail if the following is true:

[EINVAL]
The *value* parameter specifies an invalid time. This could mean that either the seconds or microseconds field of the timeval structure is negative, or the seconds field is greater than 100000000 (over 3 years), or if the microseconds field is greater than 1000000 (1 sec).

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.

CAVEATS

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).

SEE ALSO

gettimeofday(2), sigvec(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.

If the invoker is not the super–user, then the affected process must have the same effective user ID as the invoker or be a descendant of the invoking process.

DIAGNOSTICS

Setpgrp will fail and the process group will not be altered if one or more of the following occur:

[ESRCH]
The requested process does not exist.

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

RETURN VALUE

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

SEE ALSO

getpgrp(2).

NAME

getpriority, setpriority — get/set program scheduling priority

SYNOPSIS

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

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 returned in *prio* with the **getpriority** call and set to *prio* with the **setpriority** call.

Which is one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, defined in <sys/resource.h>:

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

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.

A value of 0 for *who*, in either **getpriority** or **setpriority**, will indicate the operations are to apply to the current process, process group, or user.

The **getpriority** call returns in *prio* the highest priority (lowest numerical value) enjoyed by any of the specified processes. Here, *prio* will be one of 40 values in the range —20 to 19. The default priority is 0; lower priorities cause more favorable scheduling.

The **setpriority** call sets to *prio* the priorities of all of the specified processes to the specified value. Only the super–user may lower priorities.

DIAGNOSTICS

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

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

[EINVAL]
Which is 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 is located, but neither its effective nor real user ID matched the effective user ID of the caller, and the caller is not the super–user.

[EACCES]
A non super–user is attempting to change a process priority to a negative value.

RETURN VALUE

Setpriority returns 0 if there is no error, or —1 if there is, setting *errno* to indicate the error. **Getpriority** returns the process' priority. 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.

CAVEATS

If a *prio* larger than 19 is given to **setpriority,** it will be changed to 19 and the priority set accordingly.

SEE ALSO

fork(2), nice(1).

NAME

setregid — set real and effective group ID

SYNOPSIS

setregid(rgid, egid)
int rgid, egid;

DESCRIPTION

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

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

DIAGNOSTICS

[EPERM]

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

RETURN VALUE

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

SEE ALSO

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

NAME

setreuid — set real and effective user ID's

SYNOPSIS

setreuid(ruid, euid)
int ruid, euid;

DESCRIPTION

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

DIAGNOSTICS

[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.

RETURN VALUE

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

SEE ALSO

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

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.

Getrlimit returns the limits on the current process in the rlimit structure pointed to by *rlp*; **setrlimit** uses the values in the structure to set the process limits.

The *resource* parameter is one of the following, defined in <sys/resource.h>:

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_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.

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, defined in <sys/resource.h>:

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

Only the super-user may raise the hard limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

An "infinite" value for a limit is defined as RLIMIT_INFINITY (0x7fffffff) in ⟨sys/resource.h⟩.

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(1csh)*.

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.

DIAGNOSTICS

The possible errors are:

[EFAULT]
The address specified for *rlp* is invalid.

[EPERM]
The limit specified to **setrlimit** would have raised the maximum limit value, and the caller is not the super-user.

[EINVAL]
The *resource* argument is not a valid value.

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*.

SEE ALSO

csh(1csh), sh(1sh).

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.

To manipulate options at the "socket" level, *level* is specified as SOL_SOCKET.

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

Optname and any specified option value are passed uninterpreted to the appropriate protocol module for interpretation. The include file ⟨*sys/socket.h*⟩ contains definitions for "socket" level options; see *socket*(2). At this level, *optname* is a single option; that is, only one option can be specified per call to **getsockopt** or **setsockopt**. Also, **setsockopt** will fail if an *optval* of size greater than the mbuf data length (defined in ⟨*sys/mbuf.h*⟩) is specified.

DIAGNOSTICS

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 specified in **getsockopt** is not set.

[EINVAL]
Optname or *level* is unknown; size of *optval* is too large (**setsockopt**).

[EFAULT]
The options are not in a valid part of the process address space.

[ENOBUFS]
No system buffer space is available.

RETURN VALUE

[0] Successful call. In the case of **getsockopt**, the option specified by *optname* is set.

[—1]
Unsuccessful call or *optname* is not set. An error code is stored into the global variable *errno*.

CAVEATS

At present, only "socket" level options are allowed. Of these, only SO_LINGER accepts an *optval* argument of integer size to **setsockopt**.

There is no provision for resetting an option, once set.

SEE ALSO

socket(2), getprotoent(3n).

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 in the structures pointed to by *tp* and *tzp*. **Settimeofday** sets the time, using the contents of the structures.

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 ⟨*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 *tz_dsttime*, 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.

DIAGNOSTICS

Gettimeofday and **settimeofday** may set the following errors in *errno*:

[EFAULT]
An argument address references invalid memory.

In addition, **settimeofday** may set the following error:

[EPERM]
The caller is not 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 the error.

CAVEATS

Time is never correct enough to believe the microsecond values. There should a mechanism by which, at least, local clusters of systems might synchronize their clocks to millisecond granularity. If *tzp* is NULL, the time zone information will not be returned or set.

SEE ALSO

date(1), ctime(3c).

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.

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

sigblock — block signals

SYNOPSIS

omask = sigblock(mask);
int omask;
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. Bits are numbered beginning with 1.

It is not possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

RETURN VALUE

The previous set of masked signals is returned in *omask*.

CAVEATS

As noted above, bits are numbered beginning with 1, not 0.

SEE ALSO

kill(2), sigsetmask(2), sigvec(2).

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. **Sigmask** is usually 0 to indicate that no signals are now to be blocked. **Sigpause** always terminates by being interrupted, returning EINTR.

In normal usage, a signal is blocked using *sigblock(2)*, to begin a critical section, variables modified on the occurance 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**.

RETURN VALUE

Sigpause always terminates by being interrupted, returning EINTR.

SEE ALSO

sigblock(2), sigvec(2).

NAME

sigsetmask — set current signal mask

SYNOPSIS

omask = sigsetmask(mask);
int omask;
int mask;

DESCRIPTION

Sigsetmask sets the current signal mask (those signals which are blocked from delivery) to *mask*. Signal *i* is blocked if the *i*–th bit in *mask* is a 1. Bits are numbered beginning at 1.

The system quietly disallows SIGKILL, SIGSTOP, or SIGCONT to be blocked.

RETURN VALUE

The previous set of masked signals is returned in *omask*.

CAVEATS

As noted above, bits are numbered beginning at 1, not 0.

SEE ALSO

kill(2), sigblock(2), sigpause(2), sigvec(2).

NAME

sigstack — set and/or get signal stack context

SYNOPSIS

#include <signal.h>

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. The signal stack structure is defined in <signal.h> as:

```
struct sigstack {
        caddr_t ss_sp;
        int     ss_onstack;
};
```

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.

Signal stacks are not "grown" automatically, as is done for the normal stack. If the stack overflows unpredictable results may occur.

DIAGNOSTICS

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.

RETURN VALUE

If *oss* is non–zero, upon successful completion the current signal stack state is returned in *oss*. If *oss* is NULL, upon successful completion a value of 0 is returned. If an error occurs, a value of —1 is returned and *errno* is set to indicate the error.

SEE ALSO

sigvec(2), setjmp(3c), setjmp(3f).

NAME

sigvec — software signal facilities

SYNOPSIS

#include ⟨signal.h⟩

sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;

DESCRIPTION

Sigvec assigns a handler for a specific signal *sig*, using the following structure, defined in ⟨*signal.h*⟩:

```
struct  sigvec {
        int     (*sv_handler)();
        int     sv_mask;
        int     sv_onstack;
};
```

If *vec* is non–zero, it specifies a handler routine *sv_handler()* and mask *sv_mask* to be used when delivering *sig*. Further, if *sv_onstack* is 1, 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. For an explanation of these terms, see below.

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, may reinstate the *default action* for a signal, or may specify that a signal is to be *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 initilized 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.

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 (not reset when caught)
SIGTRAP	5*	trace trap (not reset when caught)
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal from kill
SIGURG	16 ●	urgent condition present on socket, exception condition present on a device
SIGSTOP	17†	stop (cannot be caught or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19 ●	continue after stop (cannot be blocked)
SIGCHLD	20 ●	to parent on child stop or exit
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))
SIGUSR1	28	user–defined signal 1
SIGUSR2	29	user–defined signal 2
SIGPWR	31	power fail

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 (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 *sv_handler* is SIG_IGN the signal is subsequently ignored, and pending instances of the signal are discarded.

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* or *write(2)* on a slow device (such as a terminal; but not a file) and during a *wait(2)*.

After a *fork(2)* or *vfork(2)* the child inherits all signals, the signal mask, and the signal stack.

The system call *execve(2)* resets all caught signals to default action; ignored signals remain ignored; the signal mask remains the same; the signal stack state is reset.

The mask specified in *vec* is not allowed to block SIGKILL, SIGSTOP, or SIGCONT. This is done silently by the system.

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, a constant as given below. *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 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
Floating inexact result	SIGFPE	FPE_INEXCT_TRAP
Length access control	SIGSEGV	
Protection violation	SIGBUS	
Undefined instruction trap	SIGILL	ILL_UNDEF_TRAP
Privileged instruction trap	SIGILL	ILL_PRIVIN_TRAP
Floating reserved operand trap	SIGILL	ILL_RESOP_TRAP
Floating illegal instruction trap	SIGILL	ILL_FLOAT_TRAP
Customer-reserved instr.	SIGEMT	
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	

DIAGNOSTICS

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).

RETURN VALUE

If *ovec* is non–zero, upon successful completion the previous handling information is returned in *ovec*. If *ovec* is NULL, upon successful completion a 0 is returned. If an error occurs, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), kill(2), ptrace(2), sigblock(2), sigpause(2) sigsetmask(2), sigstack(2), sigvec(2), setjmp(3c), setjmp(3f), tty(4).

NAME

socket — create an endpoint for communication

SYNOPSIS

#include ⟨sys/types.h⟩
#include ⟨sys/socket.h⟩

s = socket(af, type, protocol)
int s, af, type, protocol;

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The *af* parameter specifies an address format with which addresses specified in later operations using the socket should be interpreted. These formats are defined in the include file ⟨*sys/socket.h*⟩. The currently understood formats are

AF_UNIX	(UTek path names),
AF_INET	(ARPA Internet addresses),
AF_PUP	(Xerox PUP-I Internet addresses), an
AF_IMPLINK	(IMP "host at IMP" addresses).

The socket has the indicated *type* which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams with an out-of-band data transmission mechanism. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network interfaces. SOCK_RAW is available only to the super-user.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type using a given address format. 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.

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_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send(2)* calls. It is also possible to receive datagrams at such a socket with *recv(2)*.

An *fcntl(2)* call can be used to specify a process group to receive a SIGURG signal when the out–of–band data arrives.

The operation of sockets is controlled by socket level *options*. These options are defined in the file ⟨*sys/socket.h*⟩ and explained below. *Setsockopt(2)* and *getsockopt(2)* are used to set and get options, respectively.

SO_DEBUG	turn on recording of debugging information.
SO_REUSEADDR	allow local address reuse.
SO_KEEPALIVE	keep connections alive.
SO_DONTROUTE	do no apply routing on outgoing messages.
SO_LINGER	linger on close if data present.
SO_DONTLINGER	do not linger on close.

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates 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 and SO_DONTLINGER control the actions 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_DONTLINGER is specified and a **close** is issued, the system will process the **close** in a manner which allows the process to continue as quickly as possible.

DIAGNOSTICS

The **socket** call fails if:

[EAFNOSUPPORT]
The specified address family is not supported in this version of the system.

[ESOCKTNOSUPPORT]
The specified socket type is not supported in this address family.

[EPROTONOSUPPORT]
The specified protocol is not supported.

[EMFILE]
The per-process descriptor table is full.

[ENOBUFS]
No buffer space is available. The socket cannot be created.

RETURN VALUE

A —1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

CAVEATS

The use of keepalives is a questionable feature for this layer.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), recv(2), select(2), send(2), shutdown(2), socketpair(2).

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

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.

[EOPNOSUPPORT]
The specified protocol does not support creation of socket pairs.

[EFAULT]
The address *sv* does not specify a valid part of the process address space.

RETURN VALUE

[0] **Socketpair** was successful.

[—1]
Socketpair was unsuccessful.

CAVEATS

This call is currently implemented only for the UTek domain.

SEE ALSO

read(2), write(2), pipe(2).

NAME

stat, lstat, fstat — get file status

SYNOPSIS

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

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

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

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

DESCRIPTION

Stat obtains information about the file *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 reachable.

Lstat is like **stat** 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.

Fstat obtains the same information about an open file referenced by *fd*, 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 structure is defined in ⟨*sys/stat.h*⟩ as:

```
struct stat {
   dev_t    st_dev;     /* ID of device containing a directory entry */
                        /* for this file */
   ino_t    st_ino;     /* this inode's number */
   u_short  st_mode;    /* file mode; see below */
   short    st_nlink;   /* number of hard links to the file */
   short    st_uid;     /* user ID of the file's owner */
   short    st_gid;     /* group ID of the file's group */
   dev_t    st_rdev;    /* ID of device -- this entry is defined only */
                        /* for character special or block special files *
   off_t    st_size;    /* total size of file */
   time_t   st_atime;   /* time of last access */
   int      st_spare1;
   time_t   st_mtime;   /* time of last data modification */
   int      st_spare2;
   time_t   st_ctime;   /* time of last file status change */
   int      st_spare3;
   long     st_blksize; /* optimal blocksize for file system I/O ops */
   long     st_blocks;  /* actual number of blocks allocated */
   long     st_hostid;  /* hostid of machine where file is located */
   long     st_spare4;
  };
```

st_atime Time when file data was last read or modified. Changed by the following system calls: *mknod*(2), *utimes*(2), and *read*(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).

st_ctime Time when file status was last changed. It is set both 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).

The status information word *st_mode* has these bits:

```
#define S_IFMT      0170000   /* type of file */
#define S_IFDIR     0040000   /* directory */
#define S_IFCHR     0020000   /* character special */
#define S_IFBLK     0060000   /* block special */
#define S_IFREG     0100000   /* regular */
#define S_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   /* save swapped text even after use */
#define S_IREAD     0000400   /* read permission, owner */
#define S_IWRITE    0000200   /* write permission, owner */
#define S_IEXEC     0000100   /* execute/search permission, owner */
```

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

When *fd* is associated with a pipe, **fstat** reports an ordinary file with an i–node number, restricted permissions, and a not necessarily meaningful length.

DIAGNOSTICS

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

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

[ENOASCII]
The pathname contains a character with the high–order bit set.

[ENAMETOOLONG]
The pathname is too long.

[ENOENT]
The named file does not exist.

[EACCES]
Search permission is denied for a component of the path prefix. If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

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

[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.

Fstat will fail if one of the following are true:

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

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

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

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.

CAVEATS

The fields in the stat 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 which depend on the time stamps being contiguous (in calls to *utimes*(2)).

Applying **fstat** to a socket returns a zeroed buffer.

SEE ALSO

chmod(2), chown(2), utimes(2).

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. The size of the swap area on *special* is calculated at the time the device is first made available for swapping.

DIAGNOSTICS

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

[ENOENT]
Special does not exist.

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

[ENAMETOOLONG]
The argument *special* is too long.

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

[ENOASCII]
The pathname *special* contains a character with the high-order bit set.

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

[EFAULT]
special points to an invalid address.

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

[ENOTBLK]
Special is not a block device.

[ENXIO]
The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).

[EBUSY]
Swapping is already being done on the device.

[ENODEV]
The device is not in the swap table.

[EDFSREF]
Special is a reference to a remote device and this system call does not support that capability.

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.

CAVEATS

There is no way to stop swapping on a disk so that the pack may be dismounted.

SEE ALSO

config(8), swapon(8).

NAME

symlink — make symbolic link to a file

SYNOPSIS

symlink(path1, path2)
char *path1, *path2;

DESCRIPTION

A symbolic link *path2* is created to *path1* (*path2* is the name of the file created, *path1* is the string used in creating the symbolic link). Either name may be an arbitrary pathname; the files need not be on the same file system.

DIAGNOSTICS

The symbolic link is made unless on or more of the following are true:

[ENOASCII]
Path2 contain a character with the high–order bit set.

[ENAMETOOLONG]
One of the pathnames specified is too long.

[ENOTDIR]
A component of the *path2* prefix is not a directory.

[EEXIST]
Name2 already exists.

[EACCES]
A component of the *path2* path prefix denies search permission. If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

[EROFS]
The file *path2* would reside on a read–only file system.

[ENOSPC]
The file system is out of inodes.

[ENOSPC]
The directory in which the entry for the new symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC]
The new symbolic link cannot be ecreated because there is no space left on the file system which will contain the link.

[ENFILE]
The system inode table is full.

[EFAULT]
Path1 or *path2* points outside the process's allocated address space.

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

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

[EDFSNOSUCHHOST]
Path2 referenced a remote host, but when we broadcast a request for its address, no host responded.

RETURN VALUE

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

SEE ALSO

ln(1), link(2), 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.

CAVEATS

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

SEE ALSO

fsync(2), sync(8), update(8).

NAME

syscall — indirect system call

SYNOPSIS

syscall(number, arg, ...)

DESCRIPTION

Syscall performs the system call whose assembly language interface has the specified *number,* register arguments *r0* and *r1* and further arguments *arg.*

The r0 value of the system call is returned.

RETURN VALUE

When the flag bit is set, **syscall** returns —1 and sets the external variable *errno* (see *intro*(2)).

CAVEATS

There is no way to simulate system calls such as *pipe*(2), which return values in register r1.

Unless signals are being blocked or ignored, an invalid *number* will send cause **syscall** to send signal SIGSYS (see *sigvec*(2)).

SEE ALSO

intro(2).

NAME

truncate, ftruncate — truncate a file to a specified length

SYNOPSIS

truncate(path, length)
char *path;
long length;

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

DESCRIPTION

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

DIAGNOSTICS

Truncate succeeds unless:

[ENOASCII]
The pathname contains a character with the high–order bit set.

[ENAMETOOLONG]
The pathname *path* is too long.

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

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

[ENOENT]
The named file *path* does not exist.

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

[EACCES]
Write permission is denied for *path*.

[EACCES]
If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

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

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

[EINVAL]
Length value given was negative.

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

Ftruncate succeeds unless:

[EBADF]
Fd is not a valid descriptor.

[EACCES]
Write permission is denied for the file referenced by *fd*.

[EINVAL]
Fd references a socket, not a file.

[EROFS]
Fd resides on a read–only file system.

[EINVAL]
Length value given was negative.

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.

CAVEATS

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.

SEE ALSO

open(2).

NAME

umask — set file creation mode mask

SYNOPSIS

oumask = umask(numask)
int oumask, numask;

DESCRIPTION

Umask sets the process's file mode creation mask to *numask* and returns the previous value of the mask in *oumask*. The low–order 9 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 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).

NAME

mount, umount — mount or remove file system

SYNOPSIS

mount(special, path, rwflag)
char *special, *path;
int rwflag;

umount(special)
char *special;

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block special file *special*. After successful completion, references to file *path* will refer to the root file on the newly mounted file system. *Special* and *path* are pointers to null–terminated strings containing the appropriate pathnames.

Path must exist already. *Path* must be a directory. Its old contents are inaccessible while the file system is mounted.

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

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

These calls are limited to the super–user.

DIAGNOSTICS

Mount and **umount** will fail when one of the following occurs:

[EPERM]
The caller is not the super–user.

[ENOENT]
Special or *path* does not exist.

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

[ENAMETOOLONG]
The argument *special* or *path* is too long.

[ENOTBLK]
Special is not a block device.

[ENXIO]
The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).

[ENOASCII]
The pathname *special* or *path* contains a character with the high–order bit set.

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

[EDFSREF]
Path may not reference a file system on another host.

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

In addition, **mount** will fail when one or more of the following occurs:

[ENOTDIR]
Path is not a directory.

[EBUSY]
Another process currently holds a reference to *path.*

[ENOMEM]
No space remains in the mount table.

[EINVAL]
The super block for the file system has a bad magic number or an out-of-range block size.

[ENOMEM]
Not enough memory is available to read the cylinder group information for the file system.

[EIO]
An I/O error occurred while reading the super block or cylinder group information.

[EIO]
An I/O error occurred while accessing the device.

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

[EFAULT]
Special or *path* points outside the process's allocated address space.

In addition, **umount** will fail when one or more of the following occurs:

[EINVAL]
The requested device is not in the mount table.

[EBUSY]
A process is holding a reference to a file located on the file system.

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

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

RETURN VALUE

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

SEE ALSO

mount(8), umount(8).

NAME

uname — get name of current system

SYNOPSIS

#include ⟨sys/utsname.h⟩

uname(name)
struct utsname *name;

DESCRIPTION

Uname stores information identifying the current operating system in the structure pointed to by *name*.

Uname uses the structure defined in ⟨*sys/utsname.h*⟩ whose members are:

```
char    sysname[9];
char    nodename[9];
char    release[9];
char    version[9];
char    machine[9];
```

Uname returns a null–terminated character string naming the current system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Nodename* contains the first 8 characters of the *hostname* that was set using *sethostname(2)*. The complete name is available using *gethostname(2)*. *Release* and *version* further identify the operating system. *Machine* contains a standard name that identifies the hardware that the system is running on.

DIAGNOSTICS

Uname will fail is the following is true:

[EFAULT]
Name points to an invalid address.

RETURN VALUE

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

SEE ALSO

uname(1), gethostname(2), sethostname(2).

NAME

unlink — remove directory entry

SYNOPSIS

unlink(path)
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. If, however, the file was open in any process, the actual resource reclamation is delayed until it is closed, even though the directory entry has disappeared.

Only the super-user may **unlink** a directory.

DIAGNOSTICS

The **unlink** succeeds unless:

[ENOASCII]
The path contains a character with the high-order bit set.

[ENAMETOOLONG]
The pathname is too long.

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

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

[ENOENT]
The named file does not exist.

[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.

[EACCES]
If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

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

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

[ETXTBUSY]
Path is a shared text file that is being executed.

[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.

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

RETURN VALUE

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

SEE ALSO

close(2), link(2), open(2), rmdir(2), creat(3c).

NAME

utimes — set file times

SYNOPSIS

#include ⟨sys/time.h⟩

utimes(path, tvp)
char *path;
struct timeval tvp[2];

DESCRIPTION

The **utimes** call uses the values in the *tvp* array to set the "accessed" and "modified" times (in that order) for the file named by *path*. The *timeval* structure is defined in ⟨sys/time.h⟩ as:

```
struct timeval {
        long    tv_sec; /* seconds */
        long    tv_usec;        /* and microseconds */
};
```

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.

DIAGNOSTICS

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

[ENOASCII]
The pathname contains a character with the high-order bit set.

[ENAMETOOLONG]
The pathname is too long.

[ENOENT]
The named file does not exist.

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

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

[EACCES]
A component of the path prefix denies search permission. If the file is located on a remote host, this error code will be returned if the local host name and local user name does not appear in */etc/hosts.dfs.access* on the remote machine. See *hosts.dfs.access(5n)*.

[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.

[EFAULT]
File or *tvp* points outside the process's allocated address space.

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

[EDFSNOSUCHHOST]
The pathname referenced a remote host, but when we broadcast a request for its address, no host responded.

RETURN VALUE

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

SEE ALSO

stat(2).

NAME

vfork — spawn new process in a virtual memory efficient way

SYNOPSIS

pid = vfork()
int pid;

DESCRIPTION

Vfork is identical to *fork*(2). It is provided here for compatibility with other systems. On those systems, **vfork** creates new processes without copying the address space of the old process, borrowing 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).

Here, with both **vfork** and *fork* the address space is not copied; data and stack are made copy–on–write so neither the parent nor the child can modify the other's memory. The new process is created without the overhead of copying the whole process.

Vfork returns 0 in *pid* in the child's context and (later) in *pid* 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 child's 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.)

DIAGNOSTICS

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

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

[EAGAIN]
The system–imposed limit on the total number of processes under execution by a single user, MAXUPRC, defined in ⟨sys/param.h⟩, would be exceeded.

[ENOMEM]
Insufficient space exists in the swap area for the child process.

RETURN VALUE

Upon successful completion, **vfork** returns a value of 0 in *pid* to the child process and returns the process ID of the child process in *pid* 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.

CAVEATS

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.

SEE ALSO

execve(2), fork(2), sigvec(2), wait(2).

NAME

wait, wait3 — wait for process to terminate

SYNOPSIS

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

pid = wait(status)
int pid;
union wait *status;

pid = wait(0)
int pid;

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

pid = wait3(status, options, rusage)
int pid;
union wait *status;
int options;
struct rusage *rusage;
```

DESCRIPTION

Wait suspends the calling process until it receives a signal or one of its child processes terminates. If any child is terminated prior to the call on **wait**, return is immediate, returning in *pid* the process ID and in *status* the exit status of one of the terminated children. If there are no children, return is immediate with *pid* set to —1.

On return from a successful **wait** call, *status* is nonzero, and the high byte of *status* contains the low byte of the argument to *exit* supplied by the child process; the low byte of *status* contains the termination status of the process. A more precise definition of the *status* word is given in <sys/wait.h>:

```
union wait      {
        int     w_status;               /* used in syscall */
        /*
         * Terminated process status.
         */
        struct {
                unsigned short  w_Termsig:7;    /* termination signal */
                unsigned short  w_Coredump:1;   /* core dump indicator */
                unsigned short  w_Retcode:8;    /* exit code if w_termsig=
        } w_T;
        /*
         * Stopped process status.  Returned
         * only for traced children unless requested
         * with the WUNTRACED option bit.
         */
        struct {
                unsigned short  w_Stopval:8;    /* == W_STOPPED if stopped
                unsigned short  w_Stopsig:8;    /* signal that stopped us
        } w_S;
        };
```

If **wait** is called with an argument of 0, no status information is returned.

Wait3 provides an alternate interface for programs which must not block when collecting the status of child processes. The *status* parameter is defined as above. The *options* parameter is one of the following, defined in <sys/wait.h>:

```
#define WNOHANG      1      /* don't hang in wait */
#define WUNTRACED    2      /* tell about stopped, untraced children */
```

Options is used to indicate the call should not block if there are no processes which wish to report status (WNOHANG), and/or that only children of the current process which are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal should have their status reported (WUNTRACED). If *rusage* is non–zero, a summary of the resources used by the terminated process and all its children is returned in *rusage* (this information is currently not available for stopped processes). See *getrusage(2)*.

If *rusage* is NULL, no resource information is returned.

When the WNOHANG option is specified and no processes wish to report status, **wait3** returns a *pid* of 0. The WNOHANG and WUNTRACED options may be combined by *or*'ing the two values.

See *sigvec(2)* for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted; see *ptrace(2)*. If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

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

Wait and *wait3* are automatically restarted when a process receives a signal while awaiting termination of a child process.

DIAGNOSTICS

Wait and **wait3** will fail and return immediately if one or more of the following are true:

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

[EFAULT]
The *status* or *rusage* arguments point to an illegal address.

RETURN VALUE

If **wait** or **wait3** return due to a stopped or terminated child process, the process ID of the child is returned to the calling process in *pid* and the exit status of the child is returned in *status*. Otherwise, a value of —1 is returned in *pid* and *errno* is set to indicate the error.

Wait3 returns 0 if WNOHANG is specified and there are no stopped or exited children.

SEE ALSO

exit(2), getrusage(2), ptrace(2), sigvec(2).

NAME

wait, wait3 — wait for process to terminate

SYNOPSIS

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

pid = wait(status)
int pid;
union wait *status;

pid = wait(0)
int pid;

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

pid = wait3(status, options, rusage)
int pid;
union wait *status;
int options;
struct rusage *rusage;
```

DESCRIPTION

Wait suspends the calling process until it receives a signal or one of its child processes terminates. If any child is terminated prior to the call on **wait**, return is immediate, returning in *pid* the process ID and in *status* the exit status of one of the terminated children. If there are no children, return is immediate with *pid* set to —1.

On return from a successful **wait** call, *status* is nonzero, and the high byte of *status* contains the low byte of the argument to *exit* supplied by the child process; the low byte of *status* contains the termination status of the process. A more precise definition of the *status* word is given in <sys/wait.h>:

```
union wait      {
int     w_status;               /* used in syscall */
/*
 * Terminated process status.
 */
struct {
        unsigned short  w_Termsig:7;    /* termination signal */
        unsigned short  w_Coredump:1;   /* core dump indicator */
        unsigned short  w_Retcode:8;    /* exit code if w_termsig==0 */
} w_T;
/*
 * Stopped process status.  Returned
 * only for traced children unless requested
 * with the WUNTRACED option bit.
 */
struct {
        unsigned short  w_Stopval:8;    /* == W_STOPPED if stopped */
        unsigned short  w_Stopsig:8;    /* signal that stopped us */
} w_S;

};
```

If **wait** is called with an argument of 0, no status information is returned.

Wait3 provides an alternate interface for programs which must not block when collecting the status of child processes. The *status* parameter is defined as above. The *options* parameter is one of the following, defined in <sys/wait.h>:

```
#define WNOHANG      1      /* don't hang in wait */
#define WUNTRACED    2      /* tell about stopped, untraced children */
```

Options is used to indicate the call should not block if there are no processes which wish to report status (WNOHANG), and/or that only children of the current process which are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal should have their status reported (WUNTRACED). If *rusage* is non–zero, a summary of the resources used by the terminated process and all its children is returned in *rusage* (this information is currently not available for stopped processes). See *getrusage(2)*.

If *rusage* is NULL, no resource information is returned.

When the WNOHANG option is specified and no processes wish to report status, **wait3** returns a *pid* of 0. The WNOHANG and WUNTRACED options may be combined by *or*'ing the two values.

See *sigvec(2)* for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted; see *ptrace(2)*. If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

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

Wait and *wait3* are automatically restarted when a process receives a signal while awaiting termination of a child process.

DIAGNOSTICS

Wait and **wait3** will fail and return immediately if one or more of the following are true:

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

[EFAULT]
The *status* or *rusage* arguments point to an illegal address.

RETURN VALUE

If **wait** or **wait3** return due to a stopped or terminated child process, the process ID of the child is returned to the calling process in *pid* and the exit status of the child is returned in *status*. Otherwise, a value of —1 is returned in *pid* and *errno* is set to indicate the error.

Wait3 returns 0 if WNOHANG is specified and there are no stopped or exited children.

SEE ALSO

exit(2), getrusage(2), ptrace(2), sigvec(2).

NAME

write, writev — write on a file

SYNOPSIS

```
cc = write(fd, buf, nbytes)
int cc;
int fd;
char *buf;
int nbytes;

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

cc = writev(fd, iov, iovcnt)
int cc;
int fd;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

Write attempts to write *nbytes* of data to the object referenced by the descriptor *fd* from the buffer pointed to by *buf*. **Writev** performs the same action, but gathers the output data from *iovcnt* buffers specified by the members of the *iovec* array: iov[0], iov[1], ..., iov[iovcnt–1].

Writev is not supported for raw devices (for example, raw disks, terminals) nor is it supported for a file located on a remote host.

Write and **writev** return the number of bytes written in *cc*.

For **writev**, the *iovec* structure is defined in <sys/uio.h> 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 is gathered.

On objects capable of seeking, the **write** starts at a position given by the pointer associated with *fd*, see *lseek*(2). Upon return from **write**, the pointer is incremented by the number of bytes actually written.

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 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.

DIAGNOSTICS

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

[EBADF]
Fd is not a valid descriptor open for writing.

[EPIPE]
An attempt is made to write to a pipe that is not open for reading by any process.

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

[EPIPE]
An attempt is made to write to a socket of type SOCK_STREAM which is not connected to a peer socket.

[EFBIG]
An attempt is 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.

[EMSGSIZE]
Fd is a socket, and the message sent on it was larger than the internal message buffer.

[ENOTCONN]
Fd is a socket which is not connected.

[EDESTADDREQ]
Fd is a socket, and a required address was omitted from the **write** request on the socket.

[EWOULDBLOCK]
Fd is in non-blocking mode, and the **write** would cause a process to block.

[ENOBUFS]
Fd is a socket, and the system lacks sufficient buffer space to do the **write.**

In addition, **writev** will fail is one or more of the following are true:

[EINVAL]
One of the *iov_len* values in the *iov* array is negative.

[EINVAL]
The sum of the *iov_len* values in the *iov* array overflows a 32–bit integer.

[EINVAL]
Iovcnt is less than or equal to 0, or greater than 16.

[ENXIO]
Writev was attempted on an unsupported raw device (see above).

[EDFSNOBUF]
Malloc failed on remote system; try smaller (8k or less) write.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned in *cc*. Otherwise a —1 is returned and *errno* is set to indicate the error.

SEE ALSO

lseek(2), open(2), pipe(2), read(2).

NAME

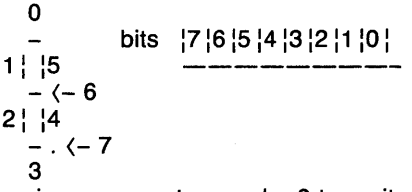
writecbd — write to computer board diagnostic display

SYNOPSIS

writecbd(pattern)
char pattern;

DESCRIPTION

Writecbd is a system call specific to the 6100 series workstations. It writes **pattern** to the seven-segment diagnostic display located at the rear of the workstation. The segments are mapped to the bits in pattern in the folowing manner:



a 1 turns the given segment on and a 0 turns it off. To use this system call the argument **—l61** must be used with *cc(1)*.

FILES

/usr/lib/lib61.a

SEE ALSO

readcbcs(2)

REFERENCES

6130 System User's Guide section 2 for switch settings.

NAME

write, writev — write on a file

SYNOPSIS

cc = write(fd, buf, nbytes)
int cc;
int fd;
char *buf;
int nbytes;

#include ⟨sys/types.h⟩
#include ⟨sys/uio.h⟩

cc = writev(fd, iov, iovcnt)
int cc;
int fd;
struct iovec *iov;
int iovcnt;

DESCRIPTION

Write attempts to write *nbytes* of data to the object referenced by the descriptor *fd* from the buffer pointed to by *buf*. **Writev** performs the same action, but gathers the output data from *iovcnt* buffers specified by the members of the *iovec* array: iov[0], iov[1], ..., iov[iovcnt–1].

Writev is not supported for raw devices (for example, raw disks, terminals) nor is it supported for a file located on a remote host.

Write and **writev** return the number of bytes written in *cc*.

For **writev**, the *iovec* structure is defined in ⟨sys/uio.h⟩ 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 is gathered.

On objects capable of seeking, the **write** starts at a position given by the pointer associated with *fd*, see *lseek*(2). Upon return from **write**, the pointer is incremented by the number of bytes actually written.

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 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.

DIAGNOSTICS

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

[EBADF]
Fd is not a valid descriptor open for writing.

[EPIPE]
An attempt is made to write to a pipe that is not open for reading by any process.

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

[EPIPE]
An attempt is made to write to a socket of type SOCK_STREAM which is not connected to a peer socket.

[EFBIG]
An attempt is 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.

[EMSGSIZE]
Fd is a socket, and the message sent on it was larger than the internal message buffer.

[ENOTCONN]
Fd is a socket which is not connected.

[EDESTADDREQ]
Fd is a socket, and a required address was omitted from the **write** request on the socket.

[EWOULDBLOCK]
Fd is in non–blocking mode, and the **write** would cause a process to block.

[ENOBUFS]
Fd is a socket, and the system lacks sufficient buffer space to do the **write.**

In addition, **writev** will fail is one or more of the following are true:

[EINVAL]
One of the *iov_len* values in the *iov* array is negative.

[EINVAL]
The sum of the *iov_len* values in the *iov* array overflows a 32–bit integer.

[EINVAL]
Iovcnt is less than or equal to 0, or greater than 16.

[ENXIO]
Writev was attempted on an unsupported raw device (see above).

[EDFSNOBUF]
Malloc failed on remote system; try smaller (8k or less) write.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned in *cc*. Otherwise a —1 is returned and *errno* is set to indicate the error.

SEE ALSO

lseek(2), open(2), pipe(2), read(2).

NAME

intro — introduction to library functions

DESCRIPTION

This section describes functions that may be found in various libraries. The library functions are those other than the functions which directly invoke UTek system primitives, described in section two. This section has the libraries physically grouped together. This is a departure from older versions of the *UNIX Programmer's Reference Manual,* which did not group functions by library. The functions described in this section are grouped into various libraries:

(3c)
The 3c functions are the standard C library functions.

(3f) The 3f functions are all functions callable from FORTRAN. These functions perform the same jobs as do the 3c functions.

(3m)
These functions constitute the math library, *libm*. They are automatically loaded as needed by the FORTRAN compiler *f77(1)*. The link editor searches this library under the **—lm** option. Declarations for these functions may be obtained from the *include* file ⟨*math.h*⟩.

(3n)
These functions constitute the internet network library.

(3s) These functions constitute the standard I/O package; see *stdio(3s)*. These functions are in the C library. Declarations for these functions may be obtained from the *include* file ⟨*stdio.h*⟩.

(3t) These functions constitute the curses and termcap libraries, and contain routines for screen management. These functions are loaded by using the arguments **—lcurses** and **—ltermlib** (or **—ltermcap**) with *cc(1)*.

(3d)
These functions constitute the database management library.

(3mp)
These functions constitute the multiple precision math library.

The functions in the (3c), (3n), and (3s) routines , constitute library *libc*, which is automatically loaded by the C compiler *cc(1)* and the FORTRAN compiler *f77(1)*. The link editor *ld(1)* searches this library under the —**lc** option. Declarations for some of these functions may be obtained from *include* files indicated on the appropriate pages.

FILES

/lib/libc.a	Standard C library
/usr/lib/libc_p.a	Standard C library for profiling
/usr/lib/libm.a	Math library

/usr/lib/libF77.a	f77 intrinsic functions
/usr/lib/libF77_p.a	f77 intrinsic functions for profiling
/usr/lib/libI77.a	f77 input/output functions
/usr/lib/libI77_p.a	f77 input/output functions for profiling
/usr/lib/libU77.a	f77 system call interface functions
/usr/lib/libU77_p.a	f77 system call interface functions for profiling
/usr/lib/libcurses.a	Curses library
/usr/lib/libtermcap.a	Terminal capability library
/usr/lib/libtermcap_p.a	Terminal capability library for profiling
/usr/lib/libtermlib.a	Terminal capability library
/usr/lib/libtermlib_p.a	Terminal capability library for profiling
/usr/lib/libdbm.a	Database management library
/usr/lib/libmp.a	Multiple precision math library

DIAGNOSTICS

Functions in the math library (3m) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable **errno** (see *intro(2)*) is set to the value EDOM (domain error) or ERANGE (range error). The values of EDOM and ERANGE are defined in the *include* file ⟨*math.h*⟩.

SEE ALSO

cc(1), f77(1), ld(1), nm(1), intro(2), curses(3t), dbm(3d), intro(3f), intro(3m), intro(3n), intro(3s), madd(3mp), stdio(3s), termcap(3t).

NAME

ERROR — an error handling routine

SYNOPSIS

#include <errdefs.h>
ERROR (exit_code, tag, out_code, format, args)
int exit_code
char *tag
out_code is a macro described in the section **Output Codes.**
char *format

DESCRIPTION

ERROR prints error messages in standard formats and keeps track of warnings. This subroutine works with special versions of *exit(3c)* and *crt0(3c)*. Each of the arguments is described in the sections below.

Exit codes

The first argument to *ERROR* is the exit code number. Some special exit codes are defined in the include file *errdefs.h*. The following paragraphs describe the actions and error message formats for different exit codes. In these sections, *program* refers to the basename of the program calling *ERROR, user_message* refers to the message text given by the **format** and **args** arguments given to *ERROR*, and **system_message** refers to the message from the system error list which corresponds to the system error which occurred before the call to *ERROR* (see *intro(2)* for a more detailed description of these messages). The exit code descriptions also refer to the "warning code". This is described in the *Exit Interface* section.

The exit code **NO_ERRS** has the value 0. This causes the warning code to be set to 0, and no message is printed.

Exit codes from 1 to 120 are not special to *ERROR* and should be used to give useful information to the user via the exit code. These codes cause the warning code to be set to the value of the exit code and an error message to be printed. The error messages produced for these codes are in the format:

program : *user_message* (*tag*)

The exit code **NO_CMD** is used by the program *sh(1sh)* to signal that the given command could not be executed.

The exit code **NP_WARN** causes the warning code to be set to the value of **NP_WARN** and a message to be printed in the format:

program : *user_message* (*tag*)

The exit code **NP_ERR** causes the program to exit with the value of **NP_WARN** after printing a message in the format:

program : *user_message* (*tag*)

The exit code **P_WARN** is used to print a warning message when a system call such as *open(2)* fails. This code causes the warning code to be set to the value of **P_WARN** and a message to be printed in the format:

program : *user_message* : *system_message* (*tag*)

The exit code **P_ERR** is used to print an error message when a system call such as *open(2)* fails. This code causes the program to exit with the value of **P_ERR** after printing a message in the format:

program : *user_message* : *system_message* (*tag*)

The exit code **USAGE** is used to print a synopsis of the command line syntax of the program. This code causes the program to exit with the value of **USAGE** after printing a message in the format:

program : **usage** : *program user_message*

The exit code **INTERNAL** is used to print error messages when an error occurs that should never occur. This code causes the program to dump core and exit after printing a message in the format:

program : **INTERNAL ERROR** : *user_message* (*tag*)
At line *line_number* **in Source file** *source_file*

Tags
The second argument to *ERROR* is a special 'tag' which is printed in parentheses after the error message. This tag is an index into the verbose error message utilities, about which very little is currently known.

If the *exit_code* argument is either **P_WARN** or **P_ERR** and the *tag* argument has a value of NULL or "" (the null string), the tag printed will be '(sys#)', where '#' is the system error number (errno). Otherwise, a null tag will cause no tag to be printed.

If the *exit_code* argument is **USAGE**, the *tag* argument is ignored.

If the environment variable NOTAGS is set, no tags will be printed.

Output Codes
The output code argument tells *ERROR* whether or not to print a message, and where to print it. The output codes are described in the include file *errdefs.h*. These codes are macros which send the output code number, the source code file name and the line number to *ERROR*, and only these macros should be used. The defined output codes and corresponding actions are described in the next three paragraphs.

The output code **O_ERR** tells *ERROR* to print the error message on standard error.

The output code **O_ERROUT** tells *ERROR* to print the error message on the standard output.

The output code **NO_OUT** tells *ERROR* not to print any error message.

User Error Messages (format and **args)**
The user may specify the error message to be printed by providing a format and arguments in the same way as with *printf(3f)*. Unless the output code is **NO_OUT**, a format must be specified, even if it is null.

Debugging with ERROR
ERROR has a special feature which can be useful while debugging programs. When the environment variable PRLINE is set, each error message is followed by a line of the format:

At line *line_number* **in Source file** *source_file*

The source file is the file containing the call to *ERROR* and the line number is the line on which *ERROR* is called. The exit code **INTERNAL** always causes this message to be printed.

Exit Interface
When *ERROR* is called with an exit code that does not cause an immediate exit, such as **NP_WARN,** an internal warning code is set. When the program exits by calling the subroutine *exit* with a value of **NO_ERRS,** or via the C statement *return* with a value of **NO_ERRS,** the exit code is replaced by the value of the internal warning code. This way, programs do not need to keep track of warnings and programs that print warning messages do not always exit with a value of 0.

EXAMPLES
The following example shows a use of *ERROR* with the **P_WARN** exit code. Assume that the program is called "example".

```
char *file;
FILE *fp;
...
        if ((fp = fopen(file, "r")) == NULL) {
                ERROR (P_WARN, "open1", O_ERR, "%s", file);
        }
...
        exit (NO_ERRS);
}
```

In this case, if the program tried to open the file "foo" for reading and the file did not exist, the message:

example : foo : No such file or directory

would be printed. When the program exits, the exit code would be the value of **P_WARN.**

VARIABLES

PRLINE	Causes the source file name and line number to be printed after any message.
NOTAGS	Suppresses the printing of tags.

CAVEATS
If the format or its arguments are invalid, errors will occur. No attempt is made to check the validity of these arguments. For example, the call:

```
ERROR (P_ERR, "open1", O_ERR, file);
```

may cause problems if the string *file* contains any '%' characters.

If a null format is given to *ERROR* with the exit codes **P_WARN** or **P_ERR**, the error message will contain two colons separated by a space. For example, the code fragment :

```
if ((fp = fopen ("file", "r")) = = NULL) {
        ERROR (P_WARN, "open1", O_ERR, "");
}
```

Will print the error message :

```
program : : No such file or directory
```

SEE ALSO

msghlp(1), sh(1sh), intro(2), abort(3c), crt0(3c), exit(3c), fopen(3s), printf(3s), errtag(5).

NAME

getdiskbyname — get disk description by its name

SYNOPSIS

#include <disktab.h>

struct disktab *
getdiskbyname(name)
char *name;

DESCRIPTION

Getdiskbyname takes a disk name (for example, **rm03**) and returns a structure describing its geometry information and the standard disk partition tables. All information is obtained from the *disktab(5)* file.

<disktab.h> has the following form:

```
/*      @(#)disktab.h     4.2 (Berkeley) 3/6/83       */

/*
 * Disk description table, see disktab(5)
 *
 * $Header: disktab.h,v 1.3 84/05/11 16:01:18 dce Stable $
 * $Locker:  $
 *
 * Modifications from 4.2bsd
 * Copyright (c) 1984, Tektronix Inc.
 * All Rights Reserved
 */

#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_rpm;                    /* revolutions/minute */
         struct   partition {
                  int      p_size;          /* #sectors in partition */
                  short    p_bsize; /* block size in bytes */
                  short    p_fsize; /* frag size in bytes */
         } d_partitions[8];
};

struct   disktab *getdiskbyname();
```

CAVEATS

This information should be obtained from the system for locally available disks (in particular, the disk partition tables).

SEE ALSO

disktab(5).

NAME

regcmp, regex — compile and execute regular expression

SYNOPSIS

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

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

extern char *_ _loc1;

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. A NULL return from **regcmp** indicates an incorrect argument. *Regcmp(1)* has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. **Regex** returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer _ *_loc1* points to where the match began. **Regmap** and **regex** were mostly borrowed from the editor, *ed(1)*; however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings:

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

$ Matches the end of the string; **\n** matches the newline.

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

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

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

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

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

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

EXAMPLES

This example will match a leading newline in the subject string pointed at by *cursor*:

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

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

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

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

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

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

CAVEATS

The user program may run out of memory if **regcmp** is called iteractively without freeing the vectors no longer required. This is because **regcmp** uses *malloc(3c)* which does not use free space. The following user-supplied replacement for *malloc(3c)* reuses the same vector saving time and space:

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

SEE ALSO

ed(1), regcmp(1), malloc(3c).

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(3c)* 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.

CAVEATS

Vfree is not implemented.

SEE ALSO

brk(2), malloc(3c).

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(3s)*) 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 it 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 run–time.

Va_end(*pvar*) is used to finish up.

Multiple traversals, each bracketted by **va_start** ... **va_end**, are possible.

EXAMPLES

The following subroutine executes the given filename with the given arguments. The first argument must be a filename, and the last argument must be a 0.

```
#include <varargs.h>
execl(va_alist)
va_dcl
{
va_list ap;
char *filename;
char *args[100];
int argno = 0;
va_start(ap);
filename = va_arg(ap, char *);
while (args[argno++] = va_arg(ap, char *))
;
va_end(ap);
return execv(filename, args);
}
```

CAVEATS

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.

SEE ALSO

nargs(3c).

NAME

abort — generate a fault

DESCRIPTION

Abort executes an instruction which is illegal in user mode. This causes a signal that normally terminates the process with a core dump, which may be used for debugging.

DIAGNOSTICS

IOT trap — core dumped	This response usually comes from the shell.

CAVEATS

The **abort()** function does not flush standard I/O buffers. Use *flush(3f).*

SEE ALSO

adb(1), sigvec(2), exit(2).

NAME

abs — integer absolute value

SYNOPSIS

abs(i)
int i;

DESCRIPTION

Abs returns the absolute value of its integer operand.

CAVEATS

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.

SEE ALSO

floor(3m).

NAME

alarm — schedule signal after specified time

SYNOPSIS

alarm(seconds)
unsigned seconds;

DESCRIPTION

NOTE: This interface is made obsolete by *setitimer(2)*.

Alarm causes signal SIGALRM, see *signal(3c)*, 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.

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

SEE ALSO

sigpause(2), sigvec(2), signal(3c), sleep(3c).

NAME

argscan, argsbad — formatted conversion of command arguments

SYNOPSIS

```
int
argscan(argc, argv, format [, pointer] . . . )
int argc;
char *argv[];
char *format;

argsbad(error)
char *error
```

DESCRIPTION

Argscan converts the given argument list according to the specified format, removing converted arguments from the list and storing the results of the conversions in the locations provided. If successful, **argscan** returns the number of arguments remaining in the list (excluding **argv[0]**); otherwise, it will return a –1 after printing messages describing the error encountered and the correct usage of the calling program.

Argscan expects as its parameters an argument count *argc,* a pointer to an argument list *argv* (see *execve(2)*), a control string *format,* described below, and a set of *pointer* arguments indicating where the converted arguments should be stored.

The format string consists of a set of *fields* and *comments* separated by spaces or tabs. Each *comment* is a string not containing percent (%) or exclamation (!) and is printed ver batum in the usage message. It does not affect argument conversion. Each *field* describes the format of an acceptable argument (or arguments) and has the following structure:

```
key conversion flag(s) whitespace fieldname
```

Key may be either of

% Means the argument is optional – its absence is ignored.

! Indicates a required argument – if absent, an error return ensues.

Conversion is a single character which specifies the type of argument expected; the corresponding *pointer* parameter(s) must be of the appropriate type.

Flag(s) consists of the alphabetic character(s) from a command argument acceptable under this field.

Whitespace is any number of blanks and tabs. It separates *flag(s)* from *fieldname.*

Fieldname is any string not containing blank, tab, percent (%), or exclamation (!). It is a mnemonic for this field and will appear in the usage message generated from this format string.

The following conversion characters are supported:

s A character string is expected in the command arguments; the corresponding parameter should be a pointer to a *char pointer*. If such a string is found, the char pointer will be set to the address of the string. The *flag(s)* part of this field must be empty.

- A dash (-) followed by one and only one of the characters in *flag(s)* is expected in the command arguments; the corresponding parameter should be an **int** pointer. Each bit of the integer pointed to corresponds to one character in *flag(s),* the leftmost character corresponding to the integer's least significant bit. When processed, only the bit corresponding to the flag specified is set; none of the other bits of the integer are modified. *Whitespace* and *fieldname* must be empty.

+ A dash (-) followed by the single character in *flag(s)* followed in the next argument by an unsigned decimal string is expected in the command arguments; the corresponding parameters should be two **int** pointers. If the proper information appears in the command arguments, the first integer will be set to 1 and the second to the converted decimal number; otherwise, neither integer is modified.

a The command arguments are expected to contain the same information as **+**, but a character string, rather than a decimal number, is expected; the corresponding parameters should be an **int** pointer and a pointer to a *char pointer*. If the proper information appears in the command arguments, the integer is set to 1 and the *char pointer* is set to the address of the string; otherwise, neither parameter is changed.

The scanner will process the format string from left to right, searching for command arguments that match the specified fields.

An argument list that does not match the requirements of the control string will cause the printing of a short message telling why, and a message telling what the correct usage is. This usage is gleaned from the control string, and the fieldnames are used directly. The fieldnames should be both terse and descriptive.

Consider the following example of a call to **argscan** for the **diff** command:

```
int blanks; int flags; char *filename1; char *filename2;
argscan(argc, argv, "%-b !-efh !s filename1 !s filename2",
                &blanks, &flags, &filename1, &filename2 );
```

This would require one and only one of either **—e**, **—f**, or **—h** to be chosen, with **-b** as an independent option. *Filename1* and *filename2* are both required. The usage message for this version of **diff** would be

```
Usage:
diff [-b] -{efh} filename1 filename2
```

Argsbad is the subroutine used by **argscan** to print an error message followed by a usage message. It is made available so that a program can call **argscan** to make a preliminary check of the command line, then call **argsbad** passing an appropriate error message (or a pointer to a null string) if any further command line error is discovered.

Since **argsbad** uses the format string and **argv[0]** passed to **argscan**, **argscan** must be called sometime prior to calling **argsbad**.

CAVEATS

By its nature a call to **argscan** defines a syntax which may be ambiguous, and although the results may be surprising, they are predictable.

To prevent string and number parameters (conversions **a** and +) from being gobbled up as string arguments (**s** conversion), all string conversion fields must follow all string and number parameter fields in the format string. For example:

```
"%-xyz !s zapstr %as string" is illegal;
"%-xyz %as string !s zapstr" is legal.
```

No check is made of the correctness of the format string.

SEE ALSO

execve(2).

NAME

assert — program verification

SYNOPSIS

#include ⟨assert.h⟩

assert(expression)

DESCRIPTION

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

DIAGNOSTICS

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

SEE ALSO

intro(3f).

NAME

atof, atoi, atol — convert ASCII to numbers

SYNOPSIS

double atof(nptr)
char *nptr;

atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;

DESCRIPTION

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

Atof recognizes an optional string of spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optionally signed integer.

Atoi and **atol** recognize an optional string of spaces, then an optional sign, then a string of digits.

DIAGNOSTICS

Atof calls *ldexp(3c),* which sets **errno** on overflow or underflow.

CAVEATS

There are no provisions for overflow in **atol** or **atoi.**

SEE ALSO

ldexp(3c), scanf(3s).

NAME

bsearch — binary search

SYNOPSIS

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

DESCRIPTION

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function.

Key points to the datum to be sought in the table.

Base points to the element at the base of the table.

Nel is the number of elements in the table.

Compar is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

DIAGNOSTICS

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

CAVEATS

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

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

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

SEE ALSO

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

NAME

bcopy, bcmp, bzero, ffs — bit and byte string operations

SYNOPSIS

bcopy(b1, b2, length)
char *b1, *b2;
int length;

int 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 *string(3c)* do.

Bcopy copies *length* bytes from string *b1* to the string *b2*.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, and nonzero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* 0 bytes in the string *b1*.

Ffs find the first bit set in the argument passed it and returns the index of that bit. A return value of zero means that no bits are set. Bits are numbered starting at 1 (i.e., **ffs** (1) returns 1, and **ffs** (8) returns 3).

CAVEATS

The **bcmp** and **bcopy** routines take parameters backwards from **strcmp** and **strcpy**.

Unlike **strcmp, bcmp** only returns an indication of equality, and not of relative order.

SEE ALSO

string(3c).

NAME

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

SYNOPSIS

#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;

DESCRIPTION

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

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

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

SEE ALSO

ctype(3c), stdio(3s).

NAME

creat — create a new file

SYNOPSIS

fd = creat(pathname, mode)
int fd;
char *pathname;
int mode;

DESCRIPTION

Note: This interface is made obsolete by *open(2)*.

Creat creates a new file or prepares to rewrite an existing file whose pathname is *pathname*. A file descriptor for the file is returned in **fd**. If the file does not exist, it is given mode *mode*, as modified by the process's mode mask (see *umask(2)*). Also see *chmod(2)* for the construction of the *mode* argument.

If the file does exist, its mode and owner remain unchanged but it is truncated to zero length; the file is also opened for writing.

DIAGNOSTICS

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

[ENOASCII]
The argument contains a byte with the high–order bit set.

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

[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.

[EISDIR]
The file is a directory.

[EMFILE]
NOFILE files are already open.

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

[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.

[ENOENT]
A component of the pathname which must exist does not exist.

[EIO]
An I/O error occurred while reading from or writing to the 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]
Pathname points outside the process's allocated address space.

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

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

RETURN VALUE

[—1] This value is returned if an error occurs.

If there is no error, the call returns a nonnegative descriptor which only permits writing.

SEE ALSO

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

NAME

crt0 — C-program startup routine.

SYNOPSIS

extern int _last_err
extern char *_pgmname

DESCRIPTION

Crt0 is an object file which contains the startup actions for C language programs. It is loaded by default by the compilers. It builds the argument lists, sets up a pointer to the program name, and initializes the global exit code (set by *ERROR(3c)* and used by *exit(3c)*), and calls the routine *exit(3c)* explicitly in case it is not called to exit the program.

The global exit code is stored in the variable *_last_err*. This value may be set by user programs in cases where *ERROR* can not be called.

The variable *_pgmname* is a pointer to the basename of the name of the program being executed. This value may be used in order to find out what name the program was called by.

CAVEATS

The variable *_pgmname* is a pointer into the first element of the argument list. Programs that change the value of the first element of the argument list may trash the name of the program, making messages from *ERROR* contain garbage.

SEE ALSO

cc(1), ld(1), ERROR(3c), exit(3c).

NAME

crypt, setkey, encrypt — DES encryption

SYNOPSIS

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

setkey(key)
char *key;

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

DESCRIPTION

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

The first argument to **crypt** is normally a user's typed password. The second is a two-character string chosen from the set **a–z**, **A–Z**, **0–9**, **.**, and **/**. 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 eight, 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 zeros and ones. 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 **setkeys**. If **edflag** is 0, the argument is encrypted; if nonzero, it is decrypted.

CAVEATS

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

SEE ALSO

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

NAME

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

SYNOPSIS

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

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)
```

DESCRIPTION

Ctime converts a time pointed to by **clock** such as returned by *gettimeofday(2)* into ASCII and returns a pointer to a 26–character string in the following form. (All the fields have constant width.)

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

Localtime and **gmtime** return pointers to structures containing the broken–down time. **Localtime** corrects for the time zone and possible daylight savings time; **gmtime** converts directly to GMT, which is the time UTek uses. **Asctime** converts a broken–down time to ASCII and returns a pointer to a 26–character string.

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 savings time is in effect.

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

Timezone returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used; otherwise, it is the daylight savings version. If the required name does not appear in a table built into the routine, the difference from GMT is produced. For example, 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.

CAVEATS

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

SEE ALSO

gettimeofday(2).

NAME

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

SYNOPSIS

#include ⟨ctype.h⟩

isalpha(c)

. . .

DESCRIPTION

These macros classify ASCII–coded integer values by table **lookup**. Each is a predicate returning nonzero for true, zero for false. **Isascii** is defined on all integer values; the rest are defined only where **isascii** is true and on the single non–ASCII value EOF (see *stdio(3s)*).

isalpha
c is a letter

isupper
c is an uppercase letter

islower
c is a lowercase letter

isdigit
c is a digit

isalnum
c is an alphanumeric character

isspace
c is a space, tab, carriage return, newline, or formfeed

ispunct
c is a punctuation character (neither control nor alphanumeric)

isprint
c is a printing character, code 040(8) (space) through 0176 (tilde)

iscntrl
c is a delete character (0177) or ordinary control character (less than 040, except for space, horizontal and vertical tab, space, linefeed, carriage return, and formfeed)

isascii
c is an ASCII character, code less than 0200

SEE ALSO

conv(3c), ascii(7).

NAME

opendir, readdir, telldir, seekdir, rewinddir, closedir — directory operations

SYNOPSIS

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

DIR *opendir(filename)
char *filename;

struct direct *readdir(dirp)
DIR *dirp;

long telldir(dirp)
DIR *dirp;

seekdir(dirp, loc)
DIR *dirp;
long loc;

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(3c)* 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.

Telldir returns the current location associated with the named *directory stream.*

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**.

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.

EXAMPLES

A sample code which searchs a directory for entry *filename* is:

```
len = strlen(filename);
dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
if (dp->d_namlen == len && !strcmp(dp->d_name, filename)) {
closedir(dirp);
return FOUND;
}
closedir(dirp);
return NOT_FOUND;
```

CAVEATS

Old UNIX programs which examine directories should be converted to use this package, as the new directory format is not obvious.

Opendir will open any file as a directory, so the file should be checked to make sure it is a directory by using *stat(2)* .

SEE ALSO

lseek(2), open(2), close(2), read(2), stat(2).

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 nonzero; 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, and get it ready for printing. Trailing zeros may be suppressed.

CAVEATS

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

SEE ALSO

printf(3s).

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(3c)*, standard input/output (*stdio(3s)*), the profile (**—p**) option of *cc(1),* and so forth. The current value of the program break is reliably returned by **sbrk(0)**; see *brk(2)*.

SEE ALSO

brk(2), malloc(3c).

NAME

ERROR — an error handling routine

SYNOPSIS

#include <errdefs.h>
ERROR (exit_code, tag, out_code, format, args)
int exit_code
char *tag
out_code is a macro described in the section **Output Codes.**
char *format

DESCRIPTION

ERROR prints error messages in standard formats and keeps track of warnings. This subroutine works with special versions of *exit(3c)* and *crt0(3c)*. Each of the arguments is described in the sections below.

Exit codes

The first argument to *ERROR* is the exit code number. Some special exit codes are defined in the include file *errdefs.h*. The following paragraphs describe the actions and error message formats for different exit codes. In these sections, *program* refers to the basename of the program calling *ERROR, user_message* refers to the message text given by the **format** and **args** arguments given to *ERROR,* and **system_message** refers to the message from the system error list which corresponds to the system error which occurred before the call to *ERROR* (see *intro(2)* for a more detailed description of these messages). The exit code descriptions also refer to the "warning code". This is described in the *Exit Interface* section.

The exit code **NO_ERRS** has the value 0. This causes the warning code to be set to 0, and no message is printed.

Exit codes from 1 to 120 are not special to *ERROR* and should be used to give useful information to the user via the exit code. These codes cause the warning code to be set to the value of the exit code and an error message to be printed. The error messages produced for these codes are in the format:

program : *user_message* (*tag*)

The exit code **NO_CMD** is used by the program *sh(1sh)* to signal that the given command could not be executed.

The exit code **NP_WARN** causes the warning code to be set to the value of **NP_WARN** and a message to be printed in the format:

program : *user_message* (*tag*)

The exit code **NP_ERR** causes the program to exit with the value of **NP_WARN** after printing a message in the format:

program : *user_message* (*tag*)

The exit code **P_WARN** is used to print a warning message when a system call such as *open(2)* fails. This code causes the warning code to be set to the value of **P_WARN** and a message to be printed in the

format:

program : *user_message* : *system_message* (*tag*)

The exit code **P_ERR** is used to print an error message when a system call such as *open(2)* fails. This code causes the program to exit with the value of **P_ERR** after printing a message in the format:

program : *user_message* : *system_message* (*tag*)

The exit code **USAGE** is used to print a synopsis of the command line syntax of the program. This code causes the program to exit with the value of **USAGE** after printing a message in the format:

program : **usage** : *program user_message*

The exit code **INTERNAL** is used to print error messages when an error occurs that should never occur. This code causes the program to dump core and exit after printing a message in the format:

program : **INTERNAL ERROR** : *user_message* (*tag*)
At line *line_number* **in Source file** *source_file*

Tags
The second argument to *ERROR* is a special 'tag' which is printed in parentheses after the error message. This tag is an index into the verbose error message utilities, about which very little is currently known.

If the *exit_code* argument is either **P_WARN** or **P_ERR** and the *tag* argument has a value of NULL or "" (the null string), the tag printed will be '(sys#)', where '#' is the system error number (errno). Otherwise, a null tag will cause no tag to be printed.

If the *exit_code* argument is **USAGE,** the *tag* argument is ignored.

If the environment variable NOTAGS is set, no tags will be printed.

Output Codes
The output code argument tells *ERROR* whether or not to print a message, and where to print it. The output codes are described in the include file *errdefs.h*. These codes are macros which send the output code number, the source code file name and the line number to *ERROR,* and only these macros should be used. The defined output codes and corresponding actions are described in the next three paragraphs.

The output code **O_ERR** tells *ERROR* to print the error message on standard error.

The output code **O_ERROUT** tells *ERROR* to print the error message on the standard output.

The output code **NO_OUT** tells *ERROR* not to print any error message.

User Error Messages (format and **args)**
The user may specify the error message to be printed by providing a format and arguments in the same way as with *printf(3s).* Unless the output code is **NO_OUT,** a format must be specified, even if it is null.

Debugging with ERROR

ERROR has a special feature which can be useful while debugging programs. When the environment variable PRLINE is set, each error message is followed by a line of the format:

At line *line_number* **in Source file** *source_file*

The source file is the file containing the call to *ERROR* and the line number is the line on which *ERROR* is called. The exit code **INTERNAL** always causes this message to be printed.

Exit Interface

When *ERROR* is called with an exit code that does not cause an immediate exit, such as **NP_WARN,** an internal warning code is set. When the program exits by calling the subroutine *exit* with a value of **NO_ERRS,** or via the C statement *return* with a value of **NO_ERRS,** the exit code is replaced by the value of the internal warning code. This way, programs do not need to keep track of warnings and programs that print warning messages do not always exit with a value of 0.

EXAMPLES

The following example shows a use of *ERROR* with the **P_WARN** exit code. Assume that the program is called "example".

```
char *file;
FILE *fp;
...
        if ((fp = fopen(file, "r")) == NULL) {
                ERROR (P_WARN, "open1", O_ERR, "%s", file);
        }
...
        exit (NO_ERRS);
}
```

In this case, if the program tried to open the file "foo" for reading and the file did not exist, the message:

example : foo : No such file or directory

would be printed. When the program exits, the exit code would be the value of **P_WARN.**

VARIABLES

PRLINE	Causes the source file name and line number to be printed after any message.
NOTAGS	Suppresses the printing of tags.

CAVEATS

If the format or its arguments are invalid, errors will occur. No attempt is made to check the validity of these arguments. For example, the call:

```
ERROR (P_ERR, "open1", O_ERR, file);
```

may cause problems if the string *file* contains any '%' characters.

If a null format is given to *ERROR* with the exit codes **P_WARN** or **P_ERR,** the error message will contain two colons separated by a space. For example, the code fragment :

```
if ((fp = fopen ("file", "r")) = = NULL) {
        ERROR (P_WARN, "open1", O_ERR, "");
}
```

Will print the error message :

```
program : : No such file or directory
```

SEE ALSO

msghlp(1), sh(1sh), intro(2), abort(3c), crt0(3c), exit(3c), fopen(3s), printf(3s), errtag(5).

NAME

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

SYNOPSIS

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

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

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

execvp(filename, argv)
char *filename, *argv[];

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

exect(filename, argv, envp)
char *filename, *argv[], *envp[];

extern char **environ;

DESCRIPTION

These routines provide various interfaces to the **execve** 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 *filename* 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 filename (or its last component). A **0** argument must end the argument list. **Execle** is like **execl** but uses the addition argument **envp**; see below.

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

The **exect** version is used when the executed file is to be manipulated with *ptrace(2)*. The program is forced to single step a single instruction giving the parent an opportunity to manipulate its state. See *ptrace(2)*.

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

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

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

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

Envp is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an equal sign (=), and a null-terminated value. The array of pointers is terminated by a null pointer. The shell *sh(1sh)* 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.

Even for the superuser, at least one of the execute-permission bits must be set for a file to be executed.

FILES

/bin/sh	Shell, invoked if command file found by **execlp** or **execvp**.

DIAGNOSTICS

See *execve(2)*.

RETURN VALUE

If **exec** returns to the calling process, an error has occurred. The return value will be -1.

CAVEATS

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

SEE ALSO

csh(1csh), sh(1sh), execve(2), fork(2), ptrace(2), a.out(5), environ(7).

NAME

exit — terminate a process

SYNOPSIS

exit(status)
int status;
extern int _last_err

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; see *wait(2)*. The low–order eight bits of **status** are available to the parent process.

If the status given to **exit** is 0 or the status is not given, the status is replaced by the value of the last warning code given to the subroutine *ERROR(3c)*. The value of this warning code is stored in the variable *_last_err,* and may be set by user programs in cases where **ERROR** cannot be used.

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

RETURN VALUE

This call never returns.

CAVEATS

Programs that "fall off the end" (for example, do not explicitly call **exit** and do not explicitly return with a value) do not exit with any useful value. In this cases, any exit code may result.

Calling **exit** with no parameters causes the same action as calling **exit** with a value of 0.

SEE ALSO

fork(2), wait(2), exit(2), ERROR(3c).

NAME

frexp, ldexp, modf — split into mantissa and exponent

SYNOPSIS

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

double ldexp(value, exp)
double value;

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

DESCRIPTION

Frexp returns the mantissa of a double *value* as a double quantity, x, of magnitude less than 1 and stores an integer n such that $value = x*2^n$ indirectly through **eptr**.

Ldexp returns the quantity $value*2^{exp}$.

Modf returns the fractional part of *value* and stores the integer part indirectly through **iptr**.

DIAGNOSTICS

If the result of **ldexp** could cause an overflow or underflow then **errno** is set to ERANGE. HUGE or –HUGE is returned on overflow depending on the sign of *value* and 0 is returned on underflow.

SEE ALSO

intro(3f).

NAME

getarhdr, fgetarhdr — read archive header

SYNOPSIS

```
#include <ar.h>
int getarhdr(fd, header)
int fd;
struct ar_hdr *header;
#include <stdio.h>
#include <ar.h>
int fgetarhdr(fp, header);
FILE *fp;
struct ar_hdr *header;
```

DESCRIPTION

The subroutine **getarhdr** reads the archive header from the file whose descriptor is *fd* and places this information in the structure pointed to by *header*. The value returned is the length of the name field in the archive file, including slashes and padding, except when there is a problem or end–of–file is reached (see RETURN VALUE). The subroutine **fgetarhdr** is similar, except that it takes a FILE pointer instead of a file descriptor.

The name field is terminated by a null. This means that by using **fgetarhdr** or **getarhdr**, filenames with imbedded spaces can be handled correctly.

The subroutine is needed because the archive format supports long filenames by surrounding them by slashes instead of making the name field longer. This means that it is no longer possible to read the entire header into a structure with a single read. See *ar(1)* and *ar(5)* for more information on the archive format.

EXAMPLES

A typical program that reads archive files would read and check the magic number and use **getarhdr** to get the header for each member of the archive file.

DIAGNOSTICS

Getarhdr and **fgetarhdr** return 0 on end–of–file and –1 when the archive file is malformed.

CAVEATS

Files opened via *fopen(3s)* should use **fgetarhdr**, especially if seeks are performed.

The file is expected to be an archive file, but it can be either a long or short format file (see *ar(5)*).

SEE ALSO

ar(1), open(2), fopen(3s), ar(5), oldar(5).

NAME

getenv — value for environment name

SYNOPSIS

char *getenv(name)
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; otherwise **getenv** returns the value 0 (NULL).

SEE ALSO

environ(7), execve(2).

NAME

getfsent, getfsspec, getfsfile, setfsent, endfsent — get file system descriptor file entry

SYNOPSIS

#include <fstab.h>

struct fstab *getfsent()

struct fstab *getfsspec(name)
char *name;

struct fstab *getfsfile(name)
char *name;

int setfsend()

int endfsent()

DESCRIPTION

Getfsent, **getfsspec**, 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, */usr/include/fstab.h*.

```
#define FSNMLG 16

struct fstab{
char fs_spec[FSNMLG];
char fs_file[FSNMLG];
char fs_type[3];
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 filename or file system filename is found, or until EOF is encountered.

FILES

/etc/fstab

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

CAVEATS

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

SEE ALSO

fstab(5).

NAME

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

SYNOPSIS

#include <grp.h>

struct group *getgrent()

struct group *getgrgid(gid)
int gid;

struct group *getgrnam(name)
char *name;

setgrent()

endgrent()

DESCRIPTION

Getgrent, **getgrgid**, and **getgrnam** each return pointers to an object with the following structure containing the broken–out fields of a line in the group file:

```
/*
 * grp.h
 *
 * $Header: grp.h,v 1.3 84/05/11 16:02:45 dce Stable $
 * $Locker:  $
 *
 * Modifications from 4.2bsd
 * Copyright (c) 1984, Tektronix Inc.
 * All Rights Reserved
 */
struct    group { /* see getgrent(3) */
          char      *gr_name;
          char      *gr_passwd;
          int       gr_gid;
          char      **gr_mem;
};

struct group *getgrent(), *getgrgid(), *getgrnam();
```

The members of this structure are:

gr_name
The name of the group.

gr_passwd
The encrypted password of the group.

gr_gid
The numerical group ID.

gr_mem

Null–terminated vector of pointers to the individual member names.

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

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

FILES

/etc/group

DIAGNOSTICS

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

CAVEATS

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

SEE ALSO

getlogin(3c), getpwent(3c), group(5).

NAME

getlogin — get loginname

SYNOPSIS

char *getlogin()

DESCRIPTION

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

If **getlogin** is called within a process that is not attached to a typewriter, it returns NULL. The correct procedure for determining the loginname is to first call **getlogin** and if it fails, to call *getpw(getuid())*.

FILES

/etc/utmp

DIAGNOSTICS

Returns NULL (0) if name not found.

CAVEATS

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

SEE ALSO

getgrent(3c), getpw(3c), getpwent(3c), utmp(5).

NAME

getopt — get option letter from argument vector

SYNOPSIS

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

extern char *optarg;
extern int optind;
extern int opterr;
extern int optopt;

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 (for example, up to the first nonoption argument), **getopt** returns **EOF**. The special option —— (dash, dash) may be used to delimit the end of the options; **EOF** will be returned, and —— (dash, dash) will be skipped.

The variable *opterr* determines whether or not **getopt** will print error messages itself. If set to 0, no messages are printed. Otherwise, **getopt** will print an error message for any unknown option or missing option argument.

The variable *optopt* is set to the current option letter, which is the same value that **getopt** returns.

EXAMPLES

The following code fragment shows how you 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':
                ifilename = optarg;
                break;
        case 'o':
                ofilename = optarg;
                bufsiza = 512;
                break;
        case '?':
                errflg++;
        }
if (errflg) {
        fprintf (stderr, ``usage: . . . ``);
        exit (2);
}
for ( ; optind < argc; optind++) {
        if (access (argv[optind], 4)) {
:
:
}
```

DIAGNOSTICS

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

CAVEATS

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

SEE ALSO

getopt(1).

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 eight characters.

Upon receiving an interrupt, the terminal state is restored, and an interrupt is sent to the process for further handling. The return value in this case will be a pointer to a null string.

FILES

/dev/tty	The user's terminal.

CAVEATS

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

SEE ALSO

crypt(3c).

NAME

getpw — get name from uid

SYNOPSIS

getpw(uid, buf)
char *buf;

DESCRIPTION

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

FILES

/etc/passwd

DIAGNOSTICS

Nonzero return on error.

SEE ALSO

getpwent(3c), passwd(5).

NAME

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

SYNOPSIS

#include <pwd.h>

struct passwd *getpwent()

struct passwd *getpwuid(uid)
int uid;

struct passwd *getpwnam(name)
char *name;

int setpwent()

int endpwent()

DESCRIPTION

Getpwent, **getpwuid**, and **getpwnam** each return a pointer to an object with the following structure containing the broken–out fields of a line in the password file.

```
/*
 * pwd.h
 *
 * $Header: pwd.h,v 1.3 84/05/11 16:05:03 dce Stable $
 * $Locker:  $
 *
 * Modifications from 4.2bsd
 * Copyright (c) 1984, Tektronix Inc.
 * All Rights Reserved
 */
struct    passwd { /* see getpwent(3) */
          char     *pw_name;
          char     *pw_passwd;
          int      pw_uid;
          int      pw_gid;
          int      pw_quota;
          char     *pw_comment;
          char     *pw_gecos;
          char     *pw_dir;
          char     *pw_shell;
};

struct passwd *getpwent(), *getpwuid(), *getpwnam();
```

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

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

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

FILES

/etc/passwd

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

CAVEATS

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

SEE ALSO

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

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.

If the current working directory is on another host, the current working hostname is prepended to the name preceded by two slashes (*//*). For example, if the local host is *home* and the current working directory is */bin* on the machine *remote*, the current working directory is *//remote/bin*.

DIAGNOSTICS

Getwd returns 0 and places a message in *pathname* if an error occurs.

CAVEATS

Getwd may fail to return to the current directory if an error occurs.

Maximum pathname length is MAXPATHLEN characters.

SEE ALSO

pwd(1), pwd(1sh), getwroot(3c).

NAME

getwd — get current working root directory pathname

SYNOPSIS

#include <sys/max.h>
char *getwroot(pathname, print_local)
char *pathname;
int print_local;

DESCRIPTION

Getwroot places the full name of the root directory of the current working directory in *pathname.* If the current working directory is on the remote machine *host*, the name will be *//host.*

If *print_local* is 0 and the current working directory is on the local host, the name will be a slash (*/*). If *print_local* is nonzero, the name will always contain the local hostname.

Pathname should be a pointer to at least MAXHOSTNAMESIZE + 3 characters. MAXHOSTNAMESIZE is defined in */usr/include/sys/max.h.*

DIAGNOSTICS

Getwroot returns –1 on error, and the value of **errno** will be set to indicate the cause.

SEE ALSO

getwd(3c).

NAME

hsearch, hcreate, hdestroy — manage hash search tables

SYNOPSIS

```
#include <search.h>

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

int hcreate (nel)
unsigned nel;

void hdestroy ( )
```

DESCRIPTION

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

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

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

DIAGNOSTICS

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

Hcreate returns 0 if it cannot allocate sufficient space for the table.

CAVEATS

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

Hsearch uses *open addressing* with a *multiplicative* hash function.

SEE ALSO

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

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

DIAGNOSTICS

Initgroups returns —1 if it was not invoked by the superuser.

CAVEATS

Initgroups uses the routines based on *getgrent(3c)*. 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.

SEE ALSO

setgroups(2).

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 be in the form of "struct qelem". **Insque** inserts *elem* in a queue immediately after *pred*; **remque** removes an entry *elem* from a queue. No values are returned.

SEE ALSO

bsearch(3c), hsearch(3c), lsearch(3c), tsearch(3c).

NAME

knlist — get entries from kernel CVT table

SYNOPSIS

#include ⟨nlist.h⟩
knlist(filename, nl)
char *filename;
struct nlist nl[];

DESCRIPTION

Knlist examines a table of kernel symbols in kernel memory (the CVT table) and selectively extracts a list of values. See *cvt(4)* for a description of the CVT table. **Knlist** performs the same function as *nlist(3c),* but runs faster and returns values that are guaranteed to correspond with the currently running kernel.

The list of names to be looked up is passed in *nl. Nl* should be terminated with a null name. *Filename* is the name of the special file containing the CVT table (usually */dev/cvt*).

If the name of an entry in *nl* is found in the CVT table, the value from the table is copied into that entry's value field in *nl.* If the name is not found, the value is set to 0. See *⟨nlist.h⟩* for the **nlist** structure declaration.

If *filename* names an *a.out* file rather than the CVT table, **knlist** will call **nlist**, passing its arguments.

FILES

/dev/cvt	Standard name for CVT table.

RETURN VALUE

Knlist returns 0 if everything worked as expected,

Knlist returns —1 upon error.

SEE ALSO

nlist(3c), cvt(4).

NAME

l3tol, ltol3 — convert between three–byte integers and long integers

SYNOPSIS

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

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

DESCRIPTION

L3tol converts a list of *n* three–byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

Ltol3 performs the reverse conversion from long integers (*lp*) to three–byte integers (*cp*).

These functions are useful for file system maintenance where the *i*–numbers are three bytes long.

SEE ALSO

intro(3f).

NAME

lsearch — linear search and update

SYNOPSIS

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table.

Key points to the datum to be sought in the table.

Base points to the first element in the table.

Nelp points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table.

Compar is the name of the comparison function which you must supply (**strcmp**, for example). It is called with two arguments that point to the elements being compared. The function must return 0 if the elements are equal and nonzero otherwise.

CAVEATS

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

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

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

Undefined results can occur if there is not enough room in the table to add a new item.

SEE ALSO

bsearch(3c), hsearch(3c), intro(3), tsearch(3c).

NAME

malloc, free, realloc, calloc — main memory allocator

SYNOPSIS

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

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION

Malloc and **free** provide a simple general–purpose memory allocation package. **Malloc** returns a pointer to a block of at least *size* bytes beginning on a word boundary.

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

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

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls **sbrk** (see *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.

Realloc also works if *ptr* points to a block freed since the last call of **malloc**,**realloc**, or **calloc**; thus sequences of **free**,**malloc**, and **realloc** can force the search strategy of **malloc** to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS

Malloc,**realloc**, and **calloc** return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. **Malloc** may be recompiled to check the arena very stringently on every transaction; those sites with a source code license may check the source code to see how this can be done.

CAVEATS

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

SEE ALSO

brk(2).

NAME

mktemp — make a unique filename

SYNOPSIS

char *mktemp(template)
char *template;

DESCRIPTION

Mktemp replaces **template** by a unique filename, and returns the address of the template. The template should look like a filename with six trailing X's, which will be replaced with the current process ID and possibly a unique letter.

The uniqueness of the filename is determined by checking to see if the resulting file exists. Therefore, successive calls to **mktemp** without creating the named file will result in the same name.

The unique letter is only supplied if it is required to make a unique filename, and is in the range a–z.

If no unique name can be built, a pointer to the string / is returned.

SEE ALSO

getpid(2).

NAME

monitor, monstartup — prepare execution profile

SYNOPSIS

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

DESCRIPTION

There are two different forms of monitoring available. An executable program created by:

```
cc -p . . .
```

automatically includes calls for **monstartup** with default parameters; **monitor** need not be called explicitly except to gain fine control over profiling. An executable program created by:

```
cc -pg . . .
```

obtains a different monitor.

Monstartup is a high level interface to *profil(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((int) 2, etext);
```

Etext lies just above all the program text, see *end(3c)*.

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

```
monitor(0)
```

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

Monitor is a low level interface to *profil(2)*. *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. At most *nfunc* calls, counts can be kept. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled. **Monitor** divides the buffer into space to record the histogram of program counter samples over the range *lowpc* to *highpc*, and spaces to record call counts of functions compiled with the **—p** option of *cc(1)*.

To profile the entire program, it is sufficient to use

```
extern etext();
monitor((int) 2, etext, buf, bufsize, nfunc);
```

FILES

mon.out

SEE ALSO

cc(1), prof(1), gprof(1), profil(2), sbrk(2).

NAME

nargs — returns the number of arguments

SYNOPSIS

int nargs()

DESCRIPTION

Nargs returns the number of arguments passed to the calling subroutine.

FILES

/lib/libc.a

CAVEATS

Nargs actually returns the number of words pushed on the stack which may not give an accurate account for structures.

SEE ALSO

varargs(3).

NAME

nice — set program priority

SYNOPSIS

int 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 super-user. The priority is limited to the range —20 (most urgent) to 20 (least).

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

RETURN VALUE

Nice returns –1 on failure, leaving the global variable *errno* set to indicate the error, and 0 on success.

SEE ALSO

nice(1), fork(2), setpriority(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, **nl**, in the given, executable 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.

DIAGNOSTICS

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

CAVEATS

On other versions of UNIX you must include ⟨*a.out.h*⟩ rather than ⟨*nlist.h*⟩; this is unfortunate, but ⟨*a.out.h*⟩ cannot be used on a 4.2bsd-based system because it has a *union* , which cannot be initialized.

SEE ALSO

a.out(5).

NAME

notmagic — interface to magic number file

SYNOPSIS

int notmagic (lowbyte, highbyte)
char lowbyte, highbyte;

DESCRIPTION

The file */usr/lib/magic* contains a list of magic numbers which describe various special files such as object files, compacted data files, and archives. The subroutine **notmagic** gets the magic numbers for files that are definitely not text files, and compares them against the combination of *lowbyte* and *highbyte* to see if the file is not a text file.

The normal way to use **notmagic** is to read the first two bytes of the file and give them to **notmagic** in the proper order for your system's byte ordering.

The first time **notmagic** is called, a table is built from the magic number file. This table is saved for further calls to **notmagic**.

See RETURN VALUE for the meaning of the return values.

FILES

/usr/lib/magic	The file containing magic number information.

RETURN VALUE

The possible return values from **notmagic** are:

0 If the characters given match a magic number.

1 If the characters given do not match a magic number.

—1 If the magic number file cannot be opened. In this case, the subroutine *ERROR(3c)* should be called with an exit code of P_ERR or P_WARN.

—2 If the magic number descriptions contain obvious errors. The command *file(1)* should be used to diagnose the problem.

CAVEATS

The only information taken from the magic number file is from lines whose magic numbers are not of type *string*, and only those lines which begin with a 0.

The characters given to **notmagic** are expected to be the first two characters in the file.

Due to some special restrictions, the programs in the **ex** editor family (which includes **vi**) use a special version of **notmagic** which can only handle up to 300 magic numbers of type long and/or short.

SEE ALSO

ex(1), file(1), ERROR(3c), magic(5).

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, see *setitimer*(2). Upon termination of a signal handler started during a **pause**, the **pause** call will return.

RETURN VALUE

Always returns —1.

ERRORS

Pause sets *errno* to:

[EINTR] The call was interrupted.

SEE ALSO

kill(2), select(2), sigpause(2).

NAME

perror, sys_errlist, sys_nerr — system error messages

SYNOPSIS

```
perror(s)
char *s;

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

DESCRIPTION

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

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; **errno** can be used as an index in this table to get the message string without the newline. *Sys_nerr* is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2), psignal(3c).

NAME

psignal, sys_siglist — system signal messages

SYNOPSIS

psignal(sig, s)
unsigned sig;
char *s;

char *sys_signlist[];

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 newline. 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 defined 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

perror(3c).

NAME

qsort — quicker sort

SYNOPSIS

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

DESCRIPTION

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

SEE ALSO

sort(1).

NAME

rand, srand — random number generator

SYNOPSIS

srand(seed)
int seed;

rand()

DESCRIPTION

Note: The newer *random(3c)* should be used in the new applications; **rand** remains for compatibilty.

Rand uses a multiplicative congruential random number generator with period 232 to return successive pseudo–random numbers in the range from 0 to 231—1.

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

SEE ALSO

random(3c).

NAME

random, srandom, initstate, setstate — better random number generator and 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 nonlinear 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(3c)* 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(3c),* 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 eight 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; its argument state array and 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.

DIAGNOSTICS

If **initstate** is called with less than eight bytes of state information, or if **setstate** detects that the state information has been garbled, error messages are printed on the standard error output.

CAVEATS

About one and a half times the speed of *rand(3c)*.

SEE ALSO

rand(3c).

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.

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 \)

CAVEATS

The string is considered to match the regular expression if a portion of the string matches. Therefore, if the entire string must match, it must be enclosed by an caret (^) and **$**.

SEE ALSO

ed(1), ex(1), egrep(1), fgrep(1), grep(1).

NAME

scandir — scan a directory

SYNOPSIS

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

scandir(dirname, namelist, select, compar)
char *dirname;
struct direct *(*namelist[]);
int (*select)();
int (*compar)();

alphasort(d1, d2)
struct direct **d1, **d2;
```

DESCRIPTION

Scandir reads the directory *dirname* and builds an array of pointers to directory entries using *malloc(3c)*. The third parameter is a pointer to a routine which is called with a pointer to a directory entry and should return a nonzero value if the directory entry should be included in the array. If this pointer is null, then all the directory entries will be included. The last argument is a pointer to a routine which is passed to *qsort(3c)* to sort the completed array. If this pointer is null, the array is not sorted. **Alphasort** is a routine which will sort the array alphabetically.

Scandir returns the number of entries in the array and a pointer to the array through the parameter namelist.

DIAGNOSTICS

Returns —1 if the directory cannot be opened for reading or if *malloc(3c)* cannot allocate enough memory to hold all the data structures.

SEE ALSO

directory(3c), malloc(3c), qsort(3c).

NAME

setjmp, longjmp — nonlocal goto

SYNOPSIS

#include ⟨setjmp.h⟩

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* for later use by **longjmp**. It returns value 0.

Longjmp restores the environment saved by the last call of **setjmp**. It then returns in such a way that execution continues as if the call of **setjmp** had just returned the value *val* to the function that invoked **setjmp**, which must not itself have returned in the interim. All accessible data have values as of the time **longjmp** was called, except for objects of storage class auto or register whose values have been changed between the **setjmp** and **longjmp** calls. These values are undefined.

SEE ALSO

goto(1csh), signal(3c).

NAME

setuid, seteuid, setruid, setgid, setegid, setrgid — set user and group ID

SYNOPSIS

```
setuid(uid)
int uid;
seteuid(euid)
int euid;
setruid(ruid)
int ruid;
setgid(gid)
int gid;
setegid(egid)
int egid;
setrgid(rgid)
int rgid;
```

DESCRIPTION

Setuid (**setgid**) sets both the real and effective user ID (group ID) of the current process to the given ID.

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.

Only the superuser may change the real user or group ID of a process. Unprivileged users may change the effective user or group ID to the real user or group ID, but to no other.

DIAGNOSTICS

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

SEE ALSO

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

NAME

signal — simplified software signal facilities

SYNOPSIS

#include ⟨signal.h⟩

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

DESCRIPTION

Signal is a simplified interface to the more general *sigvec(2)* facility.

A signal is generated by some abnormal event, initiated by a user at a terminal (**quit**, **interrupt**, **stop**), by a program error (*bus error*, and so forth), 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*⟩. The sig parameters are listed in the first column.

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction (not reset when caught)
SIGTRAP	5*	trace trap (not reset when caught)
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal from kill
SIGURG	16●	urgent condition present on socket, exception condition present on device
SIGSTOP	17†	stop (cannot be caught or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19●	continue after stop (cannot be blocked)
SIGCHLD	20●	to parent on child stop or exit
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)*)

The following signals are not yet implemented, but are planned for a later release of the system:

SIGUSR1	28	user–defined signal 1
SIGUSR2	29	user–defined signal 2
SIGCLD	30	death of a child
SIGPWR	31	power fail

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 occurences 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.

> **Note:** 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.

DIAGNOSTICS

Signal will fail and no action will take place if one of the following occurs:

[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).

RETURN VALUE

The previous action is returned on a successful call. Otherwise, —1 is returned and **errno** is set to indicate the error.

SEE ALSO

kill(1), kill(2), ptrace(2), sigblock(2), sigpause(2), sigsetmask(2), sigstack(2), sigvec(2), setjmp(3c), tty(4).

NAME

sleep — suspend execution for interval

SYNOPSIS

sleep(seconds)
unsigned seconds;

DESCRIPTION

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

CAVEATS

An interface with finer resolution is needed.

SEE ALSO

sigpause(2), setitimer(2).

NAME

strcat, strncat, strcatn, strcmp, strncmp, strcmpn, strcpy, strncpy, strcpyn, strlen, index, strchr, rindex, strrchr, strpbrk, strspn, strcspn, strtok, strtrn, strntrn — string operations

SYNOPSIS

```
#include <strings.h>

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

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

char *strcatn(s1, s2, n)
char *s1, *s2;
int n;

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

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

strcmpn(s1, s2, n)
char *s1, *s2;
int n;

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

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

char *strcpyn(s1, s2, n)
char *s1, *s2;
int n;

int strlen(s)
char *s;

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

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

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

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

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

int strspn(s1, s2)
char *s1, *s2;

int strcspn(s1, s2)
char *s1, *s2;

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

char *strtrn(s, from, to)
char *s, *from, *to;

char *strntrn(s, n, from, to)
char *s, *from, *to;
int n;
```

DESCRIPTION

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

Strcat appends a copy of string *s2* to the end of string *s1*.
Strncat and **strcatn** copy at most *n* characters. Both return a pointer to the null–terminated result.

Strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, depending on if *s1* is lexicographically greater than, equal to, or less than *s2*.
Strncmp and **strcmpn** make the same comparison but looks at at most *n* characters.

Strcpy copies string *s2* to *s1*, stopping after the null character has been moved.
Strncpy and **strcpyn** copy exactly *n* characters, truncating or null–padding *s2;* the target may not be null–terminated if the length of *s2* is *n* or more. All three return *s1*.

Strlen returns the number of nonnull characters in *s*.

Index (**rindex**) returns a pointer to the first (last) occurrence of character *c* in string *s*, or 0 if *c* does not occur in the string. The null character terminating the string is considered to be part of the string.

Strchr is a synonym for **index**.
Strrchr is a synonym for **rindex**.

Strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or **NULL** if no character from *s2* exists in *s1*.

Strspn (**strcspn**) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

Strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first

character of the first token, and will have written a **NULL** character into *s1* immediately following the returned token. If there are no separators in the string, the entire string is returned. As long as the string pointed at by s1 remains unchanged, subsequent calls with 0 for the first argument will work through string *s1* in this way until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1,* a **NULL** is returned.

Strtrn and **strntrn** implement the function of the utility *tr(1)* in a subroutine. The string given is translated using a given key. The string *s* is a pointer to the string to be translated. The integer *n* is the maximum number of characters to be translated in the string. *From* and *to* are pointers to strings containing the translation key. Each character position in each string has a corresponding character in the other. If the key strings are not of equal length, the shorter is padded using the last character in the string (not including the null terminator). As with *tr(1),* the key strings may contain range. The range **a–e** is expanded as **abcde**. A reversed range, like **e–a** is not expanded, but is interpreted as the three characters **e,—**, and **a**. If the *from* string pointer is **NULL,** the last set of key strings given are used again. In this way, a translation can be repeated a number of times without having the key strings expanded each time.

EXAMPLES

For the following examples, constant strings are used, but pointers to strings work the same way.

The following example uses **strpbrk**:

```
strpbrk ("abcde", "ce")
```

The pointer returned points to the string *cde.*

This example uses **strspn**:

```
strspn ("abcde", "cda")
```

The number returned is 1.

The following example uses **strcpy** and **strtok**:

```
char *s, *t;
(void) strcpy (s, "field1:field2 field3");
t = strtok (s, ": ");
printf ("%s ", t);
t = strtok (0, ": ");
```

```
printf ("%s ", t);
t = strtok (0, ": d");
printf ("%s\n", t);
```

This sequence of calls, when executed, cause the text

```
field1 field2 field3
```

to be displayed on the standard output.

The following example uses **strncpy** and **strtrn**. In this example, each uppercase character in each string is converted to lowercase:

```
char *s, *t;
(void) strncpy (s, "First STRING.");
t = strtrn (s, "A-Z", "a-z");
printf ("%s\n", t);
t = strtrn ("Second STRING.", 0);
printf ("%s\n", t);
```

When executed, the above code would display the following two lines on the standard output :

```
first string.
second string.
```

CAVEATS

Note that comparison of characters outside of the seven-bit ASCII character set (for example, those with octal values greater than 0177) may be unpredictable except that all such characters will be considered either all greater than or all less than the standard ASCII character set.

All string movement is performed character by character starting at the left (the first character in the string). Thus, overlapping moves toward the left will work as expected, but overlapping moves to the right may yield surprises.

Since the character **NULL** is the string terminator, the translation routines are not able to translate null characters. The call

```
strtrn (string, chars, "")
```

produces undefined results and should *not* be used.

SEE ALSO

bstring(3c), printf(3s).

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 PDP11's and other machines. *Nbytes* should be even.

SEE ALSO

bstring(3c).

NAME

syslog, openlog, closelog — control system log

SYNOPSIS

#include <syslog.h>

openlog(ident, logstat)
char *ident;

syslog(priority, message, parameters ...)
char *message;

closelog()

DESCRIPTION

Syslog arranges to write the *message* onto the system log maintained by *syslog(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 *syslog(8)* and output to the system console or files as appropriate.

If special processing is needed, **openlog** can be called to initialize the log file. Parameters are *ident* which is prepended to every message, and *logstat* which is a bit field indicating special status; current values are:

LOG_PID log the process ID with each message: useful for identifying instances of daemons.

Openlog returns zero (0) on success. If it cannot open the file */dev/log*, it writes on */dev/console* instead and returns —1.

Closelog can be used to close the log file.

EXAMPLES

```
syslog(LOG_SALERT, "who: internal error 23");

openlog("serverftp", LOG_PID);
syslog(LOG_INFO, "Connection from host %d", CallingHost);
```

CAVEATS

If you elect to have your messages sent to the standard system log as specified in the configuration file (see *syslog(8)*) and the daemon, */etc/syslog* is not running, your program will be sent a SIGPIPE (see *sigvec(2)*) on the second call to **syslog**.

SEE ALSO

syslog(8).

NAME

time, ftime — get date and time

SYNOPSIS

```
long time(0)

long time(tloc)
long *tloc;

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

DESCRIPTION

Note: These interfaces are obsoleted by *gettimeofday(2)*.

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

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

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

```
/*      timeb.h   6.1(Berkeley)83/07/29*/
/*
 * 4.2 BSD Unix – include file
 *
 * Modifications from Berkeley 4.2 BSD
 * Copyright (c) 1983, Tektronix Inc.
 * All Rights Reserved
 *
 */
/*
 * Structure returned by ftime system call
 */
struct timeb
{
        time_t    time;
        unsigned short millitm;
        short     timezone;
        short     dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local 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), settimeofday(2), ctime(3c).

NAME

times — get process times

SYNOPSIS

#include ⟨sys/types.h⟩
#include ⟨sys/times.h⟩

times(buffer)
struct tms *buffer;

DESCRIPTION

This interface is obsoleted by getrusage(2).

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

This is the structure returned by *times*:

```
/usr/include/sys/times.h
```

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

SEE ALSO

time(1), getrusage(2), wait3(2), time(3).

NAME

tsearch, tdelete, twalk — manage binary search trees

SYNOPSIS

#include ⟨search.h⟩

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );

DESCRIPTION

Tsearch is a binary tree search routine generalized from Knuth (6.2.2) Algorithm T. It returns a pointer into a tree indicating where a datum may be found. If the datum does not occur, it is added at an appropriate point in the tree.

Key points to the datum to be sought in the tree.

Rootp points to a variable that points to the root of the tree. A NULL pointer value for the variable denotes an empty tree; in this case, the variable will be set to point to the datum at the root of the new tree.

Compar is the name of the comparison function. It is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

Tdelete deletes a node from a binary search tree. It is generalized from Knuth (6.2.2) algorithm D. The arguments are the same as for **tsearch**. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. **Tdelete** returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

Twalk traverses a binary search tree.

Root is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.)

Action is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments.

The first argument is the address of the node being visited.

> **NOTE:** This is not the address of the data in the tree; it is a pointer to that address.

The second argument is a value from an enumeration data type:

```
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

It is defined in the ⟨*search.h*⟩ header file, depending on whether this is the first, second, or third time that the node has been visited (during a depth–first, left–to–right traversal of the tree), or whether the node is a leaf.

The third argument is the level of the node in the tree, with the root being level 0.

DIAGNOSTICS

A NULL pointer is returned by **tsearch** if there is not enough space available to create a new node.

A NULL pointer is returned by **tsearch** and **tdelete** if *rootp* is NULL on entry.

CAVEATS

The pointers to the key and the root of the tree should be of type pointer–to–element, and cast to type pointer–to–character.

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

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

> **Warning:** The *root* argument to **twalk** is one level of indirection less than the *rootp* arguments to **tsearch** and **tdelete**.*nentry*.

Awful things can happen if the calling function alters the pointer to the root.

SEE ALSO

bsearch(3c), hsearch(3c), intro(3), lsearch(3c).

NAME

ttyname, isatty, ttyslot — find name of a terminal

SYNOPSIS

char *ttyname(filedes)

isatty(filedes)

ttyslot()

DESCRIPTION

Ttyname returns a pointer to the null–terminated pathname 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; it is 0 otherwise.

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

FILES

*/dev/**

/etc/ttys

DIAGNOSTICS

Ttyname returns a null pointer (0) if *filedes* does not describe a terminal device in directory */dev*.

Ttyslot returns 0 if */etc/ttys* is inaccessible or if it cannot determine the control terminal.

CAVEATS

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

SEE ALSO

ioctl(2), ttys(5).

NAME

ulimit — get and set user limits

SYNOPSIS

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

1 Get the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the process's file size limit to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of superuser may increase the limit. **Ulimit** will fail and the limit will be unchanged if a process with an effective user ID other than superuser attempts to increase its file size limit. [EPERM]

3 Get the maximum possible break value (in bytes). See *brk(2)*.

RETURN VALUE

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

SEE ALSO

limit(1csh), ulimit(1sh), unlimit(1csh), brk(2), getrlimit(2), sbrk(2), setrlimit(2), write(2).

NAME

vlimit — control maximum system resource consumption

SYNOPSIS

#include ⟨sys/vlimit.h⟩

vlimit(resource, value)

DESCRIPTION

This facility is superseded by getrlimit(2).

Vlimit 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; therefore, *sh(1sh)* and *csh(1csh)* have builtin commands to set limits, called *ulimit(1sh)* and *limit(1csh),* respectively.

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.

CAVEATS

If LIM_NORAISE is set, then no grace should be given when the cpu time limit is exceeded.

SEE ALSO

csh(1csh), fork(2), getrlimit(2), sbrk(2).

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 512 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), time(2), wait3(2).

NAME

dbminit, dbmclose, fetch, store, delete, firstkey, nextkey — database subroutines

SYNOPSIS

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

dbminit(filename)
char *filename;

dbmclose()

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

DESCRIPTION

These functions maintain key/content pairs in a database. 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 database 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 *filename.dir* and *filename.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))
```

Dbmclose may be called to close the current database files.

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**.

CAVEATS

The database is not locked so concurrent access by reading and writing processes is dangerous.

Only one database may be opened at a time, though multiple databases may be handled by closing one and opening another.

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

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

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 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.

SEE ALSO

ar(1), cp(1), tar(1), tp(1).

NAME

intro — introduction to FORTRAN library functions

DESCRIPTION

This section describes those functions that are in the FORTRAN run-time library. The functions listed here provide an interface from **f77** programs to the system in the same manner as the C library does for C programs. They are automatically loaded as needed by the FORTRAN compiler *f77(1)*.

Most of these functions are in *libU77.a*. Some are in *libF77.a* or *libI77.a*.

For efficiency, the RCS ID strings are not normally included in the *a.out* file. To include them, simply declare

```
external f77lid
```

in any **f77** module.

FILES

/usr/lib/libF77.a	**f77** intrinsic function (math) and startup library.
/usr/lib/libI77.a	**f77** I/O library.
/usr/lib/libU77.a	**f77** UTek system interface library.

SEE ALSO

intro(3).

NAME

abort — terminate abruptly with core image

SYNOPSIS

subroutine abort (string)
character*(*) string

DESCRIPTION

Abort cleans up the I/O buffers and then aborts producing a *core* file in the current directory. If *string* is given, it is written to logical unit 0 preceded by **abort:**.

EXAMPLES

The following section of FORTRAN code calls the **abort** routine in cases where the value of **i** is out of range. The code could be used for debugging a FORTRAN program where **i** should never be out of the range from 1 to 3.

```
          .
          .

        GOTO (10,20,30) i
        CALL abort ("Computed GOTO out of range 1-3")

10      [FORTRAN statements associated with 1st
         statement label]
          .
          .

20      [FORTRAN statements associated with 2nd
         statement label]
          .
          .

30      [FORTRAN statements associated with 3rd
         statement label]
          .
          .
```

FILES

/usr/lib/libF77.a **f77** intrinsic function (math) and startup library.

SEE ALSO

abort(3c).

NAME

abs, iabs, dabs, cabs, zabs — FORTRAN absolute value

SYNOPSIS

integer i1, i2
real r1, r2
double precision dp1, dp2
complex cx1, cx2
double complex dx1, dx2
r2 = abs(r1)
i2 = iabs(i1)
i2 = abs(i1)
dp2 = dabs(dp1)
dp2 = abs(dp1)
cx2 = cabs(cx1)
cx2 = abs(cx1)
dx2 = zabs(dx1)
dx2 = abs(dx1)

DESCRIPTION

Abs is the family of absolute value functions. **Iabs** returns the integer absolute value of its integer argument. **Dabs** returns the double–precision absolute value of its double–precision argument. **Cabs** returns the complex absolute value of its complex argument. **Zabs** returns the double–complex absolute value of its double–complex argument. The generic form **abs** returns the type of its argument.

SEE ALSO

floor(3m).

NAME

access — determine accessability of a file

SYNOPSIS

integer function access (filename, mode)
character*(*) filename, mode

DESCRIPTION

Access checks the *filename*, for accessability with respect to the caller according to *mode*. *Mode* may include, in any order and in any combination, one or more of the following:

r	Test for read permission
w	Test for write permission
x	Test for execute permission
(blank)	Test for existence

An error code is returned if either argument is illegal, or if the file cannot be accessed in all of the specified modes. 0 is returned if the specified access would be successful.

EXAMPLES

The following FORTRAN subroutine **Openfn** illustrates how the **access** function might be used to open a file for writing. The **access** function is called to check if *filename* already exists. If the file does not exist, unit number 7 is opened for writing and the subroutine is exitted with **errno** as nonzero; otherwise an error message is printed to standard error and the routine is exitted with **errno** set to zero.

```
      SUBROUTINE Openfn (filename, errno)

        CHARACTER*(*)  filename
        INTEGER  errno, access

        errno = access (filename " ")
        IF (errno .NE. 0) THEN
                OPEN (7, FILE=filename, STATUS='NEW')
        ELSE
                WRITE (0, 900)  'File ''', filename, ''' already exists.'
        ENDIF

 900    FORMAT (A6, A, A17)

        RETURN
        END
```

Note that the same result can be obtained with the error specifier (**ERR**=) and the input/output status specifier (**IOSTAT**=) within the **open** statement.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

CAVEATS

Pathnames can be no longer than MAXPATHLEN as defined in */usr/include/max.h* .

SEE ALSO

access(2), " *perror(3f).*"

NAME

acos, dacos — FORTRAN arccosine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
r2 = acos(r1)
dp2 = dacos(dp1)
dp2 = acos(dp1)

DESCRIPTION

Acos returns the real arccosine of its real argument. **Dacos** returns the double–precision arccosine of its double–precision argument. The generic form **acos** may be used with impunity as its argument will determine the type of the returned value.

SEE ALSO

sin(3m).

NAME

aimag, dimag — FORTRAN imaginary part of complex argument

SYNOPSIS

```
real r
complex cxr
double precision dp
double complex cxd
r = aimag(cxr)
dp = dimag(cxd)
```

DESCRIPTION

Aimag returns the imaginary part of its single–precision complex argument. **Dimag** returns the double–precision imaginary part of its double–complex argument.

SEE ALSO

intro(3f).

NAME

aint, dint — FORTRAN integer part intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2 = aint(r1)
dp2 = dint(dp1)
dp2 = aint(dp1)
```

DESCRIPTION

Aint returns the truncated value of its real argument in a real. **Dint** returns the truncated value of its double-precision argument as a double-precision value. **Aint** may be used as a generic function name, returning either a real or double-precision value depending on the type of its argument.

SEE ALSO

intro(3f).

NAME

alarm — execute a subroutine after a specified time

SYNOPSIS

integer function alarm (time, proc)
integer time
external proc

DESCRIPTION

This routine arranges for subroutine *proc* to be called after *time* seconds. If *time* is 0, the alarm is turned off and no routine will be called. The returned value will be the time remaining on the last alarm.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

CAVEATS

Alarm and **sleep** interact. If **sleep** is called after **alarm**, the **alarm** process will never be called. SIGALRM will occur at the lesser of the remaining **alarm** time or the **sleep** time.

SEE ALSO

alarm(3c), sleep(3f), signal(3f).

NAME

asin, dasin — FORTRAN arcsine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
r2 = asin(r1)
dp2 = dasin(dp1)
dp2 = asin(dp1)

DESCRIPTION

Asin returns the real arcsine of its real argument. **Dasin** returns the double-precision arcsine of its double-precision argument. The generic form **asin** may be used with impunity as it derives its type from that of its argument.

SEE ALSO

sin(3m).

NAME

atan, datan — FORTRAN arctangent intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
r2 = atan(r1)
dp2 = datan(dp1)
dp2 = atan(dp1)

DESCRIPTION

Atan returns the real arctangent of its real argument. **Datan** returns the double–precision arctangent of its double–precision argument. The generic form **atan** may be used with a double–precision argument returning a double–precision value.

SEE ALSO

sin(3m).

NAME

atan2, datan2 — FORTRAN arctangent intrinsic function

SYNOPSIS

real r1, r2, r3
double precision dp1, dp2, dp3
r3 = atan2(r1, r2)
dp3 = datan2(dp1, dp2)
dp3 = atan2(dp1, dp2)

DESCRIPTION

Atan2 returns the arctangent of *arg1/arg2* as a real value. **Datan2** returns the double–precision arctangent of its double–precision arguments. The generic form **atan2** may be used with impunity with double–precision arguments.

SEE ALSO

sin(3m).

NAME

bessel functions — of two kinds for integer orders

SYNOPSIS

function besj0 (x)

function besj1 (x)

function besjn (n, x)

function besy0 (x)

function besy1 (x)

function besyn (n, x)

double precision function dbesj0 (x)
double precision x

double precision function dbesj1 (x)
double precision x

double precision function dbesjn (n, x)
double precision x

double precision function dbesy0 (x)
double precision x

double precision function dbesy1 (x)
double precision x

double precision function dbesyn (n, x)
double precision x

DESCRIPTION

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

FILES

/usr/lib/libF77.a **f77** intrinsic function (math) and startup library.

DIAGNOSTICS

Negative arguments cause **besy0**, **besy1**, and **besyn** to return a huge negative value. The system error code will be set to EDOM (33).

SEE ALSO

j0(3m), perror(3f).

NAME

bit — and, or, xor, not, rshift, lshift bitwise functions

SYNOPSIS

(intrinsic) function and (word1, word2)

(intrinsic) function or (word1, word2)

(intrinsic) function xor (word1, word2)

(intrinsic) function not (word)

(intrinsic) function rshift (word, nbits)

(intrinsic) function lshift (word, nbits)

DESCRIPTION

These bitwise functions are built into the FORTRAN compiler and return the data type of their argument(s). It is recommended that their arguments be *integer* values; inappropriate manipulation of *real* objects may cause unexpected results.

The bitwise combinatorial functions return the bitwise "and" (**and**), "or" (**or**), or "exclusive or" (**xor**) of two operands. **Not** returns the bitwise complement of its operand.

Lshift, or **rshift** with a negative **nbits**, is a logical left shift with no end around carry. **Rshift** with a positive **nbits**, is a logical right shift with no sign extension. **Lshift** with a negative **nbits**, is an logical right shift with no sign extension. No test is made for a reasonable value of **nbits**.

FILES

These functions are generated in-line by the **f77** compiler.

SEE ALSO

intro(3f).

NAME

bcopy, bcmp, bzero — byte string operations

SYNOPSIS

subroutine bcopy(s1, s2, length)
character *(*) s1, s2
integer length

integer function bcmp(s1, s2, length)
character *(*) s1, s2
integer length

subroutine bzero(b1, length)
character *(*) b1
integer length

DESCRIPTION

The routines **bcopy**, **bcmp**, and **bzero** operate on variable length strings of bytes.

Bcopy copies *length* bytes from string *s1* to the string *s2*.

Bcmp compares byte string *s1* against byte string *s2*, returning zero if they are identical, and nonzero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* 0 bytes in the string *s1*.

Although these subroutines are declared to operate on character strings they operate equally well on common blocks and arrays, as long as the addresses are properly passed and *length* is the number of bytes to be manipulated.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

SEE ALSO

bstring(3c).

NAME

chdir — change default directory

SYNOPSIS

integer function chdir (dirname)
character*(*) dirname

DESCRIPTION

The default directory for creating and locating files is changed to *dirname*. Zero is returned as the value of the function if successful; an error code otherwise.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

RETURN VALUE

Zero is returned as the value of the function if successful; an error code otherwise.

CAVEATS

Pathnames can be no longer than MAXPATHLEN as defined in */usr/include/max.h* .

Use of this function may cause the FORTRAN **inquire** statement by unit number to fail.

SEE ALSO

chdir(2), cd(1), perror(3f).

NAME

chmod — change mode of a file

SYNOPSIS

integer function chmod (filename, mode)
character*(*) filename, mode

DESCRIPTION

This function changes the file system *mode* of *filename*. *Mode* can be any specification recognized by *chmod(1)*. *Filename* must be a single pathname.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.
/bin/chmod	**Exec**'ed to change the mode.

RETURN VALUE

The normal returned value is 0. Any other value will be a system error number.

CAVEATS

Pathnames can be no longer than MAXPATHLEN as defined in */usr/include/max.h* .

SEE ALSO

chmod(1).

NAME

conjg, dconjg — FORTRAN complex conjugate intrinsic function

SYNOPSIS

complex cx1, cx2
double complex dx1, dx2
cx2 = conjg(cx1)
dx2 = dconjg(dx1)

DESCRIPTION

Conjg returns the complex conjugate of its complex argument. **Dconjg** returns the double-complex conjugate of its double-complex argument.

SEE ALSO

intro(3f).

NAME

cos, dcos, ccos — FORTRAN cosine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
complex cx1, cx2
r2 = cos(r1)
dp2 = dcos(dp1)
dp2 = cos(dp1)
cx2 = ccos(cx1)
cx2 = cos(cx1)

DESCRIPTION

Cos returns the real cosine of its real argument. **Dcos** returns the double–precision cosine of its double–precision argument. **Ccos** returns the complex cosine of its complex argument. The generic form **cos** may be used with impunity as its returned type is determined by that of its argument.

SEE ALSO

sin(3m).

NAME

cosh, dcosh — FORTRAN hyperbolic cosine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
r2 = cosh(r1)
dp2 = dcosh(dp1)
dp2 = cosh(dp1)

DESCRIPTION

Cosh returns the real hyperbolic cosine of its real argument. **Dcosh** returns the double-precision hyperbolic cosine of its double-precision argument. The generic form **cosh** may be used to return the hyperbolic cosine in the type of its argument.

SEE ALSO

sinh(3m).

NAME

etime, dtime — return elapsed execution time

SYNOPSIS

call etime (tarray)
function etime (tarray)
real tarray

call dtime (tarray)
function dtime (tarray)
real tarray

DESCRIPTION

These two routines return elapsed runtime in seconds for the calling process.

Dtime returns the elapsed time since the last call to **dtime**, or the start of execution on the first call. On return from this routine the two–element time array *(tarray)* receives the user time and system elapsed time since the last call to **dtime**, or since the start of execution. The user time is returned in the first element and system time in the second element.

Etime returns the total elapsed execution time in seconds for the calling process. The two–element time array, *tarray,* receives the user time and system elapsed time since the start of execution.

The time array *tarray* must always be given. When called as a function, **dtime** or **etime** returns the sum of user and system times.

The resolution of all timing is 1/Hz second where Hz is currently 60.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

SEE ALSO

intro(3f).

NAME

exit — terminate process with status

SYNOPSIS

subroutine exit (status)
integer status

DESCRIPTION

Exit flushes and closes all the process's files, and notifies the parent process if it is executing a **wait**. The low–order 8 bits of **status** are available to the parent process. (Therefore, **status** should be in the range 0 — 255.)

This call will never return.

The C function **exit** may cause cleanup actions before the final "sys exit".

FILES

/usr/lib/libF77.a	**f77** intrinsic function (math) and startup library.

SEE ALSO

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

NAME

exp, dexp, cexp — FORTRAN exponential intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
r2 = exp(r1)
dp2 = dexp(dp1)
dp2 = exp(dp1)
cx2 = clog(cx1)
cx2 = exp(cx1)
```

DESCRIPTION

Exp returns the real exponential function e^x of its real argument. **Dexp** returns the double-precision exponential function of its double-precision argument. **Cexp** returns the complex exponential function of its complex argument. The generic function **exp** becomes a call to **dexp** or **cexp** as required, depending on the type of its argument.

SEE ALSO

exp(3m).

NAME

fdate — return date and time in an ASCII string

SYNOPSIS

subroutine fdate (string)
character*(*) string

character*(*) function fdate()

DESCRIPTION

Fdate returns the current date and time as a 24 character string in the format described under *ctime(3c)*. Neither newline nor NULL will be included.

Fdate can be called either as a function or as a subroutine.

EXAMPLES

To print the current date and time either a function or subroutine call can be used. The following three lines illustrate the use of a subroutine call.

```
CHARACTER  string*3

CALL fdate (string)
WRITE (*,*)  string
```

If called as a function, the calling routine must define **fdate**'s type and length as shown below.

```
CHARACTER  fdate*24
EXTERNAL   fdate

WRITE (*,*)  fdate()
```

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

SEE ALSO

ctime(3c), time(3f), itime(3f), idate(3f).

NAME

fileno — Map FORTRAN logical unit to UTek file number

SYNOPSIS

integer function fileno(lunit)
integer lunit

DESCRIPTION

Fileno returns the integer file descriptor associated with the FORTRAN logical unit number, *lunit*.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

CAVEATS

The logical unit must be connected to a file or **fileno** will not return a meaningful file number.

SEE ALSO

fileno(3s).

NAME

flmin, flmax, ffrac, dflmin, dflmax, dffrac, inmax — return extreme values

SYNOPSIS

function flmin()

function flmax()

function ffrac()

double precision function dflmin()

double precision function dflmax()

double precision function dffrac()

function inmax()

DESCRIPTION

Functions **flmin** and **flmax** return the minimum and maximum positive floating point values respectively. Functions **dflmin** and **dflmax** return the minimum and maximum positive double precision floating point values. Function **inmax** returns the maximum positive integer value.

The functions **ffrac** and **dffrac** return the fractional accuracy of single and double precision floating point numbers respectively. These are the smallest numbers that can be added to 1.0 without being lost.

These functions can be used by programs that must scale algorithms to the numerical range of the processor.

FILES

/usr/lib/libF77.a **f77** intrinsic function (math) and startup library.

SEE ALSO

intro(3f).

NAME

flush — flush output to a logical unit

SYNOPSIS

subroutine flush (lunit)
integer lunit

DESCRIPTION

Flush causes the contents of the buffer for logical unit *lunit* to be flushed to the associated file. This is most useful for logical units 0 and 6 when they are both associated with the control terminal.

FILES

/usr/lib/libI77.a **f77** I/O library.

SEE ALSO

fclose(3s).

NAME

fork — create a copy of this process

SYNOPSIS

integer function fork()

DESCRIPTION

Fork creates a copy of the calling process. The only distinction between the two processes is that the value returned to one of them (referred to as the *parent* process) will be the process ID of the copy. The copy is usually referred to as the *child* process. The value returned to the child process will be zero.

All logical units open for writing are flushed before the fork to avoid duplication of the contents of I/O buffers in the external file(s).

If the returned value is negative, it indicates an error and will be the negation of the system error code. See *perror(3f)*.

A corresponding **exec** routine has not been provided because there is no satisfactory way to retain open logical units across the **exec**. However, the usual function of **fork/exec** can be performed using *system(3f)*.

A pipe can be opened to another process using the f77 **open** statement with

filename='*process***', status='pipe', access='read'**

or

filename='*process***', status='pipe', access='write'**

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

DIAGNOSTICS

If the returned value is negative, it indicates an error and will be the negation of the system error code. See *perror(3f)*.

SEE ALSO

fork(2), wait(3f), kill(3f), system(3f), perror(3f).

NAME

fseek, ftell — reposition a file on a logical unit

SYNOPSIS

integer function fseek (lunit, offset, from)
integer lunit, offset, from

integer function ftell (lunit)
integer lunit

DESCRIPTION

Fseek repositions a file accociated with a logical unit. *Lunit* must refer to an open logical unit. *Offset* is an offset in bytes relative to the position specified by *from*. Valid values for *from* are:

0 meaning *beginning of the file*
1 meaning *the current position*
2 meaning *the end of the file*

The value returned by **fseek** will be 0 if successful, and is a system error code otherwise. (See *perror(3f)*.)

Ftell returns the current position of the file associated with the specified logical unit, *lunit*. The value is an offset, in bytes, from the beginning of the file. If the value returned is negative, it indicates an error and will be the negation of the system error code. (See *perror(3f)*.)

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

SEE ALSO

fseek(3s), perror(3f).

NAME

int, ifix, idint, real, float, sngl, dble, cmplx, dcmplx, ichar, char — explicit FORTRAN type conversion

SYNOPSIS

```
integer i, j
real r, s
double precision dp, dq
complex cx
double complex dcx
character*1 ch

i = int(r)
i = int(dp)
i = int(cx)
i = int(dcx)
i = ifix(r)
i = idint(dp)

r = real(i)
r = real(dp)
r = real(cx)
r = real(dcx)
r = float(i)
r = sngl(dp)

dp = dble(i)
dp = dble(r)
dp = dble(cx)
dp = dble(dcx)

cx = cmplx(i)
cx = cmplx(i, j)
cx = cmplx(r)
cx = cmplx(r, s)
cx = cmplx(dp)
cx = cmplx(dp, dq)
cx = cmplx(dcx)

dcx = dcmplx(i)
dcx = dcmplx(i, j)
dcx = dcmplx(r)
dcx = dcmplx(r, s)
dcx = dcmplx(dp)
dcx = dcmplx(dp, dq)
dcx = dcmplx(cx)

i = ichar(ch)
ch = char(i)
```

DESCRIPTION

These functions perform conversion from one data type to another. **Int** converts to *integer* from its *real*, *double precision*, *complex*, or *double*

complex argument. If the argument is *real* or *double precision*, **int** returns the integer whose magnitude is the largest integer that does not exceed the magnitude of the argument and whose sign is the same as the sign of the argument (for example, truncation).

For complex types, the above rule is applied to the real part. **Ifix** and **idint** convert only *real* and *double precision* arguments respectively. **Real** converts to *real* from an *integer*, *double precision*, *complex*, or *double complex* argument. If the argument is *double precision* or *double complex*, as much precision is kept as is possible. If the argument is one of the complex types, the real part is returned. **Float** and **sngl** convert only *integer* and *double precision* arguments respectively. **Dble** converts any *integer*, *real*, *complex*, or *double complex* argument to *double precision* form.

If the argument is of a complex type, the real part is returned. **Cmplx** converts its *integer*, *real*, *double precision*, or *double complex* argument(s) to *complex* form. **Dcmplx** converts to *double complex* form its *integer*, *real*, *double precision*, or *complex* argument(s). Either one or two arguments may be supplied to **cmplx** and **dcmplx**. If there is only one argument, it is taken as the real part of the complex type and a imaginary part of zero is supplied. If two arguments are supplied, the first is taken as the real part and the second as the imaginary part.

Ichar converts from a character to an integer depending on the character's position in the collating sequence. **Char** returns the character in the *i*th position in the processor collating sequence where *i* is the supplied argument. For a processor capable of representing *n* characters:

ichar(char(i)) = i for 0 ⟨ = i ⟨ *n*

char(ichar(ch)) = ch for any representable character *ch*.

SEE ALSO

intro(3f).

NAME

getarg, iargc — return command line arguments

SYNOPSIS

subroutine getarg (argno, arg)
integer argno
character*(*) arg

function iargc ()

DESCRIPTION

These routines permit FORTRAN programs to access the command arguments. A call to subroutine **getarg** will return the **argno***th* command line argument in character string *arg*. The 0*th* argument is the command name. The string is truncated or padded with blanks, in accordance with the rules of FORTRAN character assignment.

Iargc returns the index of the last command line argument.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

SEE ALSO

getenv(3f), execve(2).

NAME

getc, fgetc — get a character from a logical unit

SYNOPSIS

integer function getc (char)
character char

integer function fgetc (lunit, char)
integer lunit
character char

DESCRIPTION

These routines return the next character from a file associated with a FORTRAN logical unit, bypassing normal FORTRAN I/O. **Getc** reads from logical unit 5, normally connected to the control terminal input. **Fgetc** reads from logical unit *lunit,* which must be opened for input.

The value of each function is a system status code. 0 indicates no error occured on the read; A return value of —1 indicates end-of-file was detected. A positive value will be either a UTek system error code or an **f77** I/O error code. See *perror(3f).*

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

The value of each function is a system status code.

[0] Indicates no error occured on the read.

[—1] Indicates end-of-file was detected.

A positive value will be either a UTek system error code or an **f77** I/O error code. See *perror(3f).*

SEE ALSO

getc(3s), intro(2), perror(3f).

NAME

getcwd — get pathname of current working directory

SYNOPSIS

integer function getcwd (dirname)
character*(*) dirname

DESCRIPTION

The pathname of the default directory for creating and locating files will be returned in *dirname*. The value of the function will be zero if successful; it is an error code otherwise.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

RETURN VALUE

The value of the function will be zero if successful; it is an error code otherwise.

CAVEATS

Pathnames can be no longer than MAXPATHLEN as defined in */usr/include/max.h* .

SEE ALSO

chdir(3f), perror(3f).

NAME

getenv — get value of environment variables

SYNOPSIS

subroutine getenv (ename, evalue)
character*(*) ename, evalue

DESCRIPTION

Getenv searches the environment list (see *environ(7))* for a string of the form *ename=value.* If such a string is present, **getenv** returns *value* in *evalue;* otherwise *evalue* is filled with blanks.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

SEE ALSO

environ(7), execve(2).

NAME

getlog — get user's loginname

SYNOPSIS

subroutine getlog (loginname)
character*(*) loginname

character*(*) function getlog ()

DESCRIPTION

Getlog will return the user's loginname or all blanks if the process is running detached from a terminal.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

SEE ALSO

getlogin(3c).

NAME

getpid — get process id of current process

SYNOPSIS

integer function getpid ()

DESCRIPTION

Getpid returns the process ID number of the current process.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

SEE ALSO

getpid(2).

NAME

getuid, getgid — get user or group ID of the caller

SYNOPSIS

integer function getuid ()

integer function getgid ()

DESCRIPTION

The functions **getuid** and **getgid** return the real user or group ID of the user of the current process, respectively.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

SEE ALSO

getuid(2).

NAME

hostnm — get name of current host

SYNOPSIS

integer function hostnm (name)
character*(*) name

DESCRIPTION

This function puts the name of the current host machine into character string *name*. The return value should be 0; any other value indicates an error.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

The return value should be 0; any other value indicates an error.

SEE ALSO

gethostname(2).

NAME

idate, itime — return date or time in numerical form

SYNOPSIS

subroutine idate (iarray)
integer iarray

subroutine itime
integer iarray

DESCRIPTION

Idate returns the current date in *iarray*. The order is: day, month, year. Day will be in the range 1–31. Month will be in the range 1–12. Year will be ≥ 1969.

Itime returns the current time in *iarray*. The order is: hour, minute, second.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

SEE ALSO

ctime(3f), fdate(3f).

NAME

index, rindex, lnblnk, len — tell about character objects

SYNOPSIS

(intrinsic) function index (string, substr)
character*(*) string, substr

integer function rindex (string, substr)
character*(*) string, substr

function lnblnk (string)
character*(*) string

(intrinsic) function len (string)
character*(*) string

DESCRIPTION

Index (**rindex**) returns the index of the first (last) occurrence of the substring *substr* in *string,* or zero if it does not occur. **Index** is an **f77** built–in intrinsic function; **rindex** is a library routine.

Lnblnk returns the index of the last nonblank character in *string*. This is useful since all **f77** character objects are fixed length, and blank padded.

Intrinsic function **len** returns the size of the character object argument, *string*.

FILES

Index and **len** are reinerated in–line by the **f77** compiler.

/usr/lib/libF77.a **f77** intrinsic function (math) and startup library.

SEE ALSO

intro(3f).

NAME

ioinit — change f77 I/O initialization

SYNOPSIS

logical function ioinit (cctl, bzro, apnd, prefix, vrbose)
logical cctl, bzro, apnd, vrbose
character*(*) prefix

DESCRIPTION

This routine will initialize several global parameters in the **f77** I/O system, and attaches externally defined files to logical units at run time. The effect of the flag arguments applies only to logical units opened after **ioinit** is called. The exception is the preassigned units, 5 and 6, to which *cctl* and *bzro* apply at any time. **Ioinit** is written in Fortran–77.

By default, carriage control is not recognized on any logical unit. If *cctl* is **.true.** then carriage control will be recognized on formatted output to all logical units except unit 0 (**stderr**), the diagnostic channel. Otherwise the default will be restored.

By default, trailing and embedded blanks in input data fields are ignored. If *bzro* is **.true.** then such blanks will be treated as zeros. Otherwise, the default will be restored.

By default, all files opened for sequential access are positioned at their beginning. It is sometimes necessary or convenient to open at the end–of–file so that a **write** will append to the existing data. If *apnd* is **.true.** then files opened subsequently on any logical unit will be positioned at their end upon opening. A value of **.false.** will restore the default behavior.

Many systems provide an automatic association of global names with FORTRAN logical units when a program is run. There is no such automatic association in **f77**. However, if the argument *prefix* is a nonblank string, then names of the form *prefixNN* will be sought in the program environment. The value associated with each such name found will be used to open logical unit *NN* for formatted sequential access. See the first example below.

If the argument *vrbose* is **.true.** then **ioinit** will report on its activity.

The internal flags are stored in a labeled common block with the following definition:

```
INTEGER*2 ieof, ictl, ibzr
COMMON /ioiflg/ ieof, ictl, ibzr
```

EXAMPLES

Proper usage of the *prefix* parameter is shown in the following subprogram call within the **f77** program **myprogram**:

```
CALL ioinit (.true., .false., .false., 'FORT', .false.)
```

Executing the following sequence

```
% setenv FORT01 mydata
% setenv FORT12 myresults
% myprogram
```

would result in logical unit 1, opened to file *mydata* and logical unit 12, opened to file *myresults*. Any formatted output would have column 1 removed and interpreted as carriage control as indicated by the first parameter *cctl* set to **.true.**. Embedded and trailing blanks would be ignored on input (second parameter *bzro* set to **.false.**). Both files would be positioned at their beginning (third parameter *apnd* set to **.false.**).

The effect of

```
CALL ioinit (.true., .true., .false., '', .false.)
```

can be achieved without the actual call by including **—ll66** on the **f77** command line. This gives carriage control on all logical units except 0, causes files to be opened at their beginning, and causes blanks to be interpreted as zeros.

FILES

/usr/lib/libF77.a	**f77** intrinsic and startup library.
/usr/lib/libI66.a	Sets older FORTRAN I/O modes.

RETURN VALUE

The value of **ioinit** will be **.true.** unless some error occurred.

CAVEATS

Prefix can be no longer than 30 characters. A pathname associated with an environment name can be no longer than 255 characters.

The + carriage control character does not work.

SEE ALSO

getarg(3f), getenv(3f).

NAME

kill — send a signal to a process

SYNOPSIS

function kill (pid, signum)
integer pid, signum

DESCRIPTION

Kill sends a signal to a user process. *Pid* must be the process ID of one of the user's processes. *Signum* must be a valid signal number (see *sigvec(2)*). The returned value will be 0 if successful; it is an error code otherwise.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

The returned value will be 0 if successful; it is an error code otherwise.

SEE ALSO

kill(2), sigvec(2), signal(3f), fork(3f), perror(3f).

NAME

link — make a link to an existing file

SYNOPSIS

function link (filename1, filename2)
character*(*) filename1, filename2

integer function symlnk (filename1, filename2)
character*(*) filename1, filename2

DESCRIPTION

Link makes a hard link to an existing *filename1*; the link itself has the name *filename2*.

Filename1 must be the pathname of an existing file. *Filename2* is a pathname to be linked to *filename1*. *Filename2* must not already exist. With hard links, both *filename1* and *filename2* must be in the same file system. Unless the caller is the superuser, *filename1* must not be a directory. Both all old links and the new **link** share equal access and rights to the underlying object (*filename1*).

Symlnk creates a symbolic link to *filename1*; the link is *filename2*. Either name may be an arbitrary pathname; the files need not be on the same file system.

The returned value will be 0 if successful; a system error code otherwise.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

The returned value will be 0 if successful; a system error code otherwise.

CAVEATS

Pathnames can be no longer than MAXPATHLEN as defined in */usr/include/max.h* .

SEE ALSO

link(2), symlink(2), perror(3f), unlink(3f).

NAME

loc — return the address of an object

SYNOPSIS

integer function loc (arg)

DESCRIPTION

The value returned from **loc** is the address of *arg*.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

The returned value is the address of *arg*.

SEE ALSO

intro(3f).

NAME

log, alog, dlog, clog — FORTRAN natural logarithm intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
complex cx1, cx2
r2 = alog(r1)
r2 = log(r1)
dp2 = dlog(dp1)
dp2 = log(dp1)
cx2 = clog(cx1)
cx2 = log(cx1)

DESCRIPTION

Alog returns the real natural logarithm of its real argument. **Dlog** returns the double-precision natural logarithm of its double-precision argument. **Clog** returns the complex logarithm of its complex argument. The generic function **log** becomes a call to **alog**, **dlog**, or **clog** depending on the type of its argument.

SEE ALSO

exp(3m).

NAME

log10, alog10, dlog10 — FORTRAN common logarithm intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
r2 = alog10(r1)
r2 = log10(r1)
dp2 = dlog10(dp1)
dp2 = log10(dp1)

DESCRIPTION

Alog10 returns the real common logarithm of its real argument. **Dlog** returns the double-precision common logarithm of its double-precision argument. The generic function **log** becomes a call to **alog** or **dlog** depending on the type of its argument.

SEE ALSO

exp(3m).

NAME

long, short — integer object conversion

SYNOPSIS

integer*4 function long (int2)
integer*2 int2

integer*2 function short (int4)
integer*4 int4

DESCRIPTION

The functions **long** and **short** provide conversion between short and long integer objects. **Long** is useful when constants are used in calls to library routines and the code is to be compiled with **—i2**. **Short** is useful in similar context when an otherwise long object must be passed as a short integer.

FILES

/usr/lib/libF77.a	**f77** intrinsic function (math) and startup library.

SEE ALSO

intro(3f).

NAME

max, max0, amax0, max1, amax1, dmax1 — FORTRAN maximum-value functions

SYNOPSIS

```
integer i, j, k, l
real a, b, c, d
double precision dp1, dp2, dp3
l = max(i, j, k)
c = max(a, b)
dp = max(a, b, c)
k = max0(i, j)
a = amax0(i, j, k)
i = max1(a, b)
d = amax1(a, b, c)
dp3 = dmax1(dp1, dp2)
```

DESCRIPTION

The maximum-value functions return the largest of their arguments (of which there may be any number). **Max** is the generic form which can be used for all data types and takes its return type from that of its arguments (which must all be of the same type). **Max0** returns the integer form of the maximum value of its integer arguments. **Amax0** is the real form of its integer arguments. **Max1** is the integer form of its real arguments. **Amax1** is the real form of its real arguments. And **dmax1** is the double-precision form of its double-precision arguments.

SEE ALSO

min(3f).

NAME

mclock — returns FORTRAN time accounting

SYNOPSIS

integer i i = mclock( )

DESCRIPTION

Mclock returns time accounting information about the current process and its child processes. The value returned is the sum of the current process's user time and the user and system times of all child processes.

SEE ALSO

time(3c).

NAME

min, min0, amin0, min1, amin1, dmin1 — FORTRAN minimum-value functions

SYNOPSIS

```
integer i, j, k, l
real a, b, c, d
double precision dp1, dp2, dp3
l = min(i, j, k)
c = min(a, b)
dp = min(a, b, c)
k = min0(i, j)
a = amin0(i, j, k)
i = min1(a, b)
d = amin1(a, b, c)
dp3 = dmin1(dp1, dp2)
```

DESCRIPTION

The minimum-value functions return the minimum of their arguments (of which there may be any number). **Min** is the generic form which can be used for all data types and takes its return type from that of its arguments (which must all be of the same type). **Min0** returns the integer form of the minimum value of its integer arguments. **Amin0** is the real form of its integer arguments. **Min1** is the integer form of its real arguments. **Amin1** is the real form of its real arguments. And **dmin1** is the double-precision form of its double-precision arguments.

SEE ALSO

max(3f).

NAME

mod, amod, dmod — FORTRAN remaindering intrinsic functions

SYNOPSIS

```
integer i, j, k
real r1, r2, r3
double precision dp1, dp2, dp3
k = mod(i, j)
r3 = amod(r1, r2)
r3 = mod(r1, r2)
dp3 = dmod(dp1, dp2)
dp3 = mod(dp1, dp2)
```

DESCRIPTION

Mod returns the integer remainder of its first argument divided by its second argument. **Amod** and **dmod** return, respectively, the real and double–precision whole number remainder of the integer division of their two arguments. The generic version **mod** will return the data type of its arguments.

SEE ALSO

intro(3f).

NAME

nargs — returns the number of arguments

SYNOPSIS

integer function nargs()

DESCRIPTION

Nargs returns the number of arguments passed to the calling subroutine.

FILES

/usr/lib/libU77.a

CAVEATS

Nargs actually returns the number of words pushed on the stack which may not give an accurate count of arguments. Extra arguments are passed for complex and character functions and character or procedure arguments.

SEE ALSO

intro(3f), nargs(3c).

NAME

perror, gerror, ierrno — get system error messages

SYNOPSIS

subroutine perror (string)
character*(*) string

subroutine gerror (string)
character*(*) string

character*(*) function gerror ()

integer function ierrno ()

DESCRIPTION

Perror writes a message to FORTRAN logical unit 0 *(stderr)* appropriate to the last detected system error. The user specified *string* will be written preceding the standard error message.

Gerror returns the system error message in character variable *string*. **Gerror** may be called either as a subroutine or as a function.

Ierrno will return the error number of the last detected system error. This number is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call; that value is a more reliable indicator of what caused the error condition.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

DIAGNOSTICS

UTek system error codes are described in *intro*(2). The **f77** I/O error codes and their meanings are:

100	error in format
101	illegal unit number
102	formatted I/O not allowed
103	unformatted I/O not allowed
104	direct I/O not allowed
105	sequential I/O not allowed
106	cannot backspace file
107	off beginning of record
108	cannot stat file
109	no * after repeat count
110	off end of record
111	truncation failed
112	incomprehensible list input
113	out of free space
114	unit not connected
115	read unexpected character
116	blank logical input field

117	*new* file exists
118	cannot find *old* file
119	unknown system error
120	requires seek ability
121	illegal argument
122	negative repeat count
123	illegal operation for unit

CAVEATS

String in the call to **perror** can be no longer than 127 characters.

The length of the string returned by **gerror** is determined by the calling program.

SEE ALSO

intro(2), perror(3c).

NAME

putc, fputc — write a character to a FORTRAN logical unit

SYNOPSIS

integer function putc (char)
character char

integer function fputc (lunit, char)
integer lunit
character char

DESCRIPTION

These funtions write a character to the file associated with a FORTRAN logical unit, bypassing normal FORTRAN I/O. **Putc** writes to logical unit 6, normally connected to the control terminal output. **Fputc** writes to logical unit *lunit*, which must be opened for output.

The value of each function will be zero unless some error occurred; it is a system error code otherwise. The value 0 indicates no error occurred on the write. See *perror(3f)*.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

The value of each function will be 0 unless some error occurred; a system error code otherwise. The value0 indicates no error occurred on the write. See *perror(3f)*.

SEE ALSO

putc(3s), intro(2), perror(3f).

NAME

qsort — quick sort

SYNOPSIS

subroutine qsort (array, len, isize, compar)
integer len, isize
external compar
integer*2 compar

DESCRIPTION

Qsort is an implementation of the quicker—sort algorithm. One dimensional *array* contains the elements to be sorted. *Len* is the number of elements in the array. *Isize* is the size (or width) of an element in bytes, typically:

4	For *integer* and *real.*
8	For *double precision* or *complex.*
16	For *double complex.*
length of character object	For *character* arrays.

Compar is the name of a user supplied integer*2 comparison function that will determine the sorting order. This function will be called with two arguments that will be elements of *array*. The function must return:

negative	If *arg1* is considered to percede *arg2.*
zero	If *arg1* is equivalent to *arg2.*
positive	If *arg1* is considered to follow *arg2.*

On return, the elements of *array* will be sorted.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

On return, the elements of *array* will be sorted.

SEE ALSO

qsort(3c).

NAME

rand, drand, irand — return random values

SYNOPSIS

integer function irand (iflag)

real function rand (iflag)

double precision function drand (iflag)

DESCRIPTION

These functions use *rand(3c)* to generate sequences of random numbers. If *iflag* is 1, the generator is restarted and the first random value is returned. If *iflag* is otherwise nonzero, it is used as a new seed for the random number generator, and the first new random value is returned.

Irand returns positive integers in the range 0 through 2147483647 $(2^{31} - 1)$. **Rand** and **drand** return values in the range 0. through 1.0.

FILES

/usr/lib/libF77.a **f77** intrinsic function (math) and startup library.

RETURN VALUE

Irand returns positive integers in the range 0 through 2147483647 $(2^{31} - 1)$. **Rand** and **drand** return values in the range 0. through 1.0 .

CAVEATS

The algorithm returns a 31 bit quantity.

SEE ALSO

rand(3c).

NAME

rename — rename a file

SYNOPSIS

integer function rename (from, to)
character*(*) from, to

DESCRIPTION

From must be the pathname of an existing file. *To* will become the new pathname for the file. If *to* exists, then both *from* and *to* must be the same type of file, and must reside on the same file system. Furthermore, if *to* exists, it will be removed first.

The returned value will be 0 if successful; it is a system error code otherwise.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

The value returned by **rename** will be 0 if successful; a system error code otherwise.

CAVEATS

Pathnames can be no longer than MAXPATHLEN as defined in */usr/include/max.h* .

SEE ALSO

rename(2), perror(3f).

NAME

rndmode — set rounding method for floating point instructions

SYNOPSIS

subroutine rndmode (mode)
int mode

DESCRIPTION

Rndmode allows the user to chose which rounding method should be used when a floating point instruction produces a result that can not be exactly represented. The default method is to round to the nearest value.

The *mode* has the following meaning:

VALUE	MEANING
0	Round to the nearest value.
1	Round toward zero.
2	Round toward positive infinity.
3	Round toward negative infinity.

FILES

/usr/lib/libF77.a **f77** intrinsic function (math) and startup library.

RETURN VALUE

[0] **Rndmode** was not able to set rounding mode.

[1] **Rndmode** set rounding mode.

SEE ALSO

traper(3f).

NAME

anint, dnint, nint, idnint — FORTRAN nearest integer functions

SYNOPSIS

```
integer i
real r1, r2
double precision dp1, dp2
r2 = anint(r1)
i = nint(r1)
dp2 = anint(dp1)
dp2 = dnint(dp1)
i = nint(dp1)
i = idnint(dp1)
```

DESCRIPTION

Anint returns the nearest whole real number to its real argument (for example, **int(a+0.5)** if a $\geq$ 0, **int(a—0.5)** otherwise). **Dnint** does the same for its double–precision argment. **Nint** returns the nearest integer to its real argument. **Idnint** is the double–precision version. **Anint** is the generic form of **anint** and **dnint**, performing the same operation and returning the data type of its argument. **Nint** is also the generic form of **idnint**.

SEE ALSO

intro(3f).

NAME

setjmp, longjmp — nonlocal goto

SYNOPSIS

integer function setjmp(ienv) integer ienv(18)

integer function longjmp(ienv, ival) integer ienv(18) integer ival

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 *ienv* 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 *ival* 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, except for objects of storage class auto or register whose values have been changed between the **setjmp** and **longjmp** calls. These values are undefined.

SEE ALSO

setjmp(3c), signal(3c).

NAME

sign, isign, dsign — FORTRAN transfer-of-sign intrinsic function

SYNOPSIS

```
integer i, j, k
real r1, r2, r3
double precision dp1, dp2, dp3
k = isign(i, j)
k = sign(i, j)
r3 = sign(r1, r2)
dp3 = dsign(dp1, dp2)
dp3 = sign(dp1, dp2)
```

DESCRIPTION

Isign returns the magnitude of its first argument with the sign of its second argument. **Sign** and **dsign** are its real and double-precision counterparts, respectively. The generic version is **sign** and will devolve to the appropriate type depending on its arguments. When using **sign**, it is required that the both arguments be of the same type.

SEE ALSO

intro(3f).

NAME

signal — change the action for a signal

SYNOPSIS

integer function signal (signum, proc, flag)
integer signum, flag
external proc

DESCRIPTION

When a process incurs a signal (see *signal(3c)*) the default action is usually to clean up and abort. The user may choose to write an alternative signal handling routine. A call to **signal** is the way this alternate action is specified to the system.

Signum is the signal number (see *signal(3c)*). If *flag* is negative, then *proc* must be the name of the user signal handling routine. If *flag* is 0 or positive, then *proc* is ignored and the value of *flag* is passed to the system as the signal action definition. In particular, this is how previously saved signal actions can be restored. Two possible values for *flag* have specific meanings: 0 means *use the default action.* 1 means *ignore this signal.*

A positive returned value is the previous action definition. A value greater than 1 is the address of a routine that was to have been called on occurrence of the given signal. The returned value can be used in subsequent calls to **signal** in order to restore a previous action definition. A negative returned value is the negation of a system error code. (See *perror(3f).*)

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

CAVEATS

F77 arranges to trap certain signals when a process is started. The only way to restore the default **f77** action is to save the returned value from the first call to **signal**.

If the user signal handler is called, it will be passed the signal number as an integer argument.

SEE ALSO

kill(1), kill(3f), signal(3c).

NAME

sin, dsin, csin — FORTRAN sine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
complex cx1, cx2
r2 = sin(r1)
dp2 = dsin(dp1)
dp2 = sin(dp1)
cx2 = csin(cx1)
cx2 = sin(cx1)

DESCRIPTION

Sin returns the real sine of its real argument. **Dsin** returns the double-precision sine of its double-precision argument. **Csin** returns the complex sine of its complex arguemnt. The generic **sin** function becomes **dsin** or **csin** as required by argument type.

SEE ALSO

sin(3m).

NAME

sinh, dsinh — FORTRAN hyperbolic sine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2 = sinh(r1)
dp2 = dsinh(dp1)
dp2 = sinh(dp1)
```

DESCRIPTION

Sinh returns the real hyperbolic sine of its real argument. **Dsinh** returns the double–precision hyperbolic sine of its double–precision argument. The generic form **sinh** may be used to return a double–precision value given a double–precision argument.

SEE ALSO

sinh(3m).

NAME

sleep — suspend execution for an interval

SYNOPSIS

subroutine sleep (itime)
integer itime

DESCRIPTION

Sleep causes the calling process to be suspended for *itime* seconds. The actual time can be up to one second less than *itime* due to granularity in system timekeeping.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

SEE ALSO

sleep(3c).

NAME

sqrt, dsqrt, csqrt — FORTRAN square root intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
r2 = sqrt(r1)
dp2 = dsqrt(dp1)
dp2 = sqrt(dp1)
cx2 = csqrt(cx1)
cx2 = sqrt(cx1)
```

DESCRIPTION

Sqrt returns the real square root of its real argument. **Dsqrt** returns the double–precision square root of its double–precision arguement. **Csqrt** returns the complex square root of its complex argument. **Sqrt**, the generic form, will become **dsqrt** or **csqrt** as required by its argument type.

SEE ALSO

exp(3m).

NAME

stat, lstat, fstat — get file status

SYNOPSIS

integer function stat (filename, statb)
character*(*) filename
integer statb(12)

integer function lstat (filename, statb)
character*(*) filename
integer statb(12)

integer function fstat (lunit, statb)
integer statb(12), lunit

DESCRIPTION

These routines return detailed information about a file. **Stat** and **lstat** return information about *filename*; **fstat** returns information about the file associated with FORTRRAN logical unit *lunit*.

The order and meaning of the information returned in array *statb* is as described for the structure **stat** under *stat(2)*. The spare values are not included.

The value of either function will be 0 if successful; an error code is returned otherwise.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

The value of either function will be zero if successful; an error code is returned otherwise.

CAVEATS

Pathnames can be no longer than MAXPATHLEN as defined in */usr/include/max.h* .

SEE ALSO

stat(2), access(3f), perror(3f), time(3f).

NAME

system — execute a UTek command

SYNOPSIS

integer function system (string)
character*(*) string

DESCRIPTION

System causes *string* to be given to your shell as input as if the string had been typed as a command. If environment variable **SHELL** is found, its value will be used as the command interpreter (shell); otherwise *sh(1sh)* is used.

The current process waits until the command terminates. The returned value will be the exit status of the shell. See *wait(2)* for an explanation of this value.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

The returned value will be the exit status of the shell. See *wait(2)* for an explanation of this value.

CAVEATS

String cannot be longer than NCARGS—50 characters, as defined in ⟨*sys/param.h*⟩.

SEE ALSO

exec(2), wait(2), system(3s).

NAME

tan, dtan — Fortran tangent intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2 = tan(r1)
dp2 = dtan(dp1)
dp2 = tan(dp1)
```

DESCRIPTION

Tan returns the real tangent of its real argument. **Dtan** returns the double–precision tangent of its double–precision argument. The generic **tan** function becomes **dtan** as required with a double–precision argument.

SEE ALSO

sin(3m).

NAME

tanh, dtanh — FORTRAN hyperbolic tangent intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
r2 = tanh(r1)
dp2 = dtanh(dp1)
dp2 = tanh(dp1)

DESCRIPTION

Tanh returns the real hyperbolic tangent of its real argument. **Dtanh** returns the double–precision hyperbolic tangent of its double precision argument. The generic form **tanh** may be used to return a double–precision value given a double–precision argument.

SEE ALSO

sinh(3m).

NAME

time, ctime, ltime, gmtime — return system time

SYNOPSIS

integer function time ()

character*(*) function ctime (stime)
integer stime

subroutine ltime (stime, tarray)
integer stime, tarray(9)

subroutine gmtime (stime, tarray)
integer stime, tarray(9)

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. This is the value of the UTek system clock.

Ctime converts a system time to a 24 character ASCII string. The format is described under *ctime(3c)*. No newline or NULL is included.

Ltime and **gmtime** disect a UTek time into month, day, and so forth, either for the local time zone or as GMT. The order and meaning of each element returned in *tarray* is described under *ctime(3c)*.

FILES

/usr/lib/libU77.a	**f77** UTek system interface library.

SEE ALSO

ctime(3c), itime(3f), idate(3f), fdate(3f).

NAME

traper — trap arithmetic errors

SYNOPSIS

integer function traper (mask)

DESCRIPTION

Floating point roundoff errors and floating point underflow are not normally trapped during execution. This routine enables these traps by setting status bits in the floating point status register. These bits remain set throughout the execution of a program, and the previous state is restored only upon exit of the program or through another call to **traper**. If the condition occurs and trapping is enabled, signal SIGFPE is sent to the process. (See *signal(3c)*.)

The mask has the following meaning:

VALUE	MEANING
0	Do not trap either condition.
1	Trap inexact results (roundoff) only.
2	Trap floating underflow only.
3	Trap both the above.

The previous value of these bits is returned.

FILES

/usr/lib/libF77.a **f77** intrinsic function (math) and startup library.

RETURN VALUE

The previous value of the *mask* bits is returned.

SEE ALSO

signal(3c), signal(3f).

NAME

trapov — trap and repair floating point overflow

SYNOPSIS

subroutine trapov (numesg, rtnval)
double precision rtnval

DESCRIPTION

NOTE: This routine is outdated by **trpfpe**. See *trpfpe(3f)* for the newer error handler.

This subroutine sets up signal handlers to trap arithmetic exceptions. Trapping arithmetic exceptions allows the user's program to proceed from instances of floating point overflow or divide by zero. The result of such operations will be replaced by *rtnval*.

The first *numesg* occurrences of a floating point arithmetic error will cause a message to be written to the standard error file. *Rtnval* must be a double precision value. For example, **0d0** or **dflmax()**.

FILES

/usr/lib/libF77.a	**f77** intrinsic function (math) and startup library.

CAVEATS

Other arithmetic exceptions can be trapped but not repaired.

SEE ALSO

trpfpe(3f), signal(3f), flmin(3f).

NAME

trpfpe, fpecnt — trap and repair floating point faults

SYNOPSIS

subroutine trpfpe (numesg, rtnval)
double precision rtnval

integer function fpecnt ()

common /fpeflt/ fperr
logical fperr

DESCRIPTION

Trpfpe sets up a signal handler to trap arithmetic exceptions. If the exception is due to a floating point arithmetic fault, the result of the operation is replaced with the *rtnval* specified. *Rtnval* must be a double precision value. For example, **0d0** or **dflmax()**.

The first *numesg* occurrences of a floating point arithmetic error will cause a message to be written to the standard error file *(stderr)*. Any exception that can not be repaired will result in the default action, typically an abort with core image.

Fpecnt returns the number of faults since the last call to **trpfpe**.

The logical value in the common block labeled **fpeflt** will be set to **.true.** each time a fault occurs.

FILES

/usr/lib/libF77.a **f77** intrinsic function (math) and startup library.

CAVEATS

There is not a problem with the fixing an operation generated by the **f77** compiler, but there may be problems in fixing an operation in an assembly language routine.

The CMP and DEI opcodes are not dealt with.

SEE ALSO

signal(3f), flmin(3f).

NAME

ttynam, isatty — find name of a terminal port

SYNOPSIS

character*(*) function ttynam (lunit)
integer lunit

logical function isatty (lunit)
integer lunit

DESCRIPTION

Ttynam returns a blank padded pathname of the terminal device associated with logical unit *lunit*.

Isatty returns **.true.** if *lunit* is associated with a terminal device; it returns **.false.** otherwise.

FILES

*/dev/**

/usr/lib/libU77.a **f77** UTek system interface library.

DIAGNOSTICS

Ttynam returns an empty string (all blanks) if *lunit* is not associated with a terminal device in directory */dev*.

SEE ALSO

intro(3f), isatty(3c), ttyname(3c).

NAME

unlink — remove a directory entry

SYNOPSIS

integer function unlink (dirname)
character*(*) dirname

DESCRIPTION

Unlink causes the directory entry specified by pathname *dirname* to be removed. If this was the last link to the file, the contents of the file are lost. The returned value will be 0 if successful; it is a system error code otherwise.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

The value returned by **unlink** will be 0 if successful; it is a system error code otherwise.

CAVEATS

Pathnames can be no longer than MAXPATHLEN as defined in */usr/include/max.h* .

SEE ALSO

unlink(2), link(3f), filsys(5), perror(3f).

NAME

wait — wait for a process to terminate

SYNOPSIS

integer function wait (status)
integer status

DESCRIPTION

Wait causes its caller to be suspended until a signal is received or one of its child processes terminates. If any child has terminated since the last **wait**, return is immediate; if there are no children, return is immediate with an error code.

If the returned value is positive, it is the process ID of the child, and *status* is its termination status (see *wait(2)*). If the returned value is negative, it is the negation of a system error code.

FILES

/usr/lib/libU77.a **f77** UTek system interface library.

RETURN VALUE

If the returned value is positive, it is the process ID of the child and *status* is its termination status (see *wait(2)*). If the returned value is negative, it is the negation of a system error code.

SEE ALSO

wait(2), signal(3f), kill(3f), perror(3f).

NAME

intro — introduction to mathematical library functions

DESCRIPTION

These functions constitute the math library, *libm.a*. They are automatically loaded as needed by the FORTRAN compiler *f77(1)*. The link editor searches this library under the **—lm** option. Declarations for these functions may be obtained from the include file RI ⟨ math.h ⟩.

DESCRIPTION OF FUNCTIONS

Name	*Appears on Page*	*Description*
acos	sin.3m	trigonometric functions
asin	sin.3m	trigonometric functions
atan	sin.3m	trigonometric functions
atan2	sin.3m	trigonometric functions
cabs	hypot.3m	Euclidean distance
ceil	floor.3m	floor, ceiling, remainder, absolute value functions
cos	sin.3m	trigonometric functions
cosh	sinh.3m	hyperbolic functions
exp	exp.3m	exponential, logarithm, power, square root
fabs	floor.3m	floor, ceiling, remainder, absolute value functions
floor	floor.3m	floor, ceiling, remainder, absolute value functions
fmod	floor.3m	floor, ceiling, remainder, absolute value functions
gamma	gamma.3m	log gamma function
hypot	hypot.3m	Euclidean distance
intro	intro.3m	introduction to mathemetical library functions
j0	j0.3m	bessel functions
j1	j0.3m	bessel functions
jn	j0.3m	bessel functions
log	exp.3m	exponential, logarithm, power, square root
log10	exp.3m	exponential, logarithm, power, square root
pow	exp.3m	exponential, logarithm, power, square root
sin	sin.3m	trigonometric functions
sinh	sinh.3m	hyperbolic functions
sqrt	exp.3m	exponential, logarithm, power, square root
tan	sin.3m	trigonometric functions
tanh	sinh.3m	hyperbolic functions
y0	j0.3m	bessel functions
y1	j0.3m	bessel functions
yn	j0.3m	bessel functions

FILES

/usr/lib/libm.a	Mathematical library functions .
/usr/lib/libF77.a	F77 intrinsic function (math) and startup library.

SEE ALSO

intro(3).

NAME

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

SYNOPSIS

#include ⟨math.h⟩

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

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

double sqrt (x)
double x;

DESCRIPTION

Exp returns the exponential function of *x*.

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

Pow returns *x* to the power of *y*.

Sqrt returns the square root of *x*.

FILES

/usr/lib/libm.a	Mathematical library functions.

DIAGNOSTICS

Exp and **pow** return a huge value when the correct value would overflow; **errno** is set to ERANGE. **Pow** returns 0 and sets **errno** to EDOM when the first argument is negative or zero and the second argument is negative and non–integral, for example, pow(–1.0,–0.5). The value of pow (0.0,0.0) is defined to be 1.0

Log returns –HUGE when *x* is zero or negative; **errno** is set to EDOM.

Sqrt returns 0 when *x* is negative; **errno** is set to EDOM.

SEE ALSO

hypot(3m), sinh(3m), intro(3m).

NAME

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

SYNOPSIS

#include ⟨math.h⟩

double floor(x)
double x;

double ceil(x)
double x;

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

double fabs(x)
double x;

DESCRIPTION

Floor returns the largest integer not greater than x.

Ceil returns the smallest integer not less than x.

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

Fabs returns the absolute value of $|x|$.

EXAMPLES

```
floor (5.4)  =  5.0
floor (-5.4) = -6.0

ceil (5.4)  =  6.0
ceil (-5.4) = -5.0

fmod (-10.6, 0.0) = -10.6
fmod (10.6, 4.0)  =  2.6
fmod (-10.6, 4.0) = -2.6
```

FILES

/usr/lib/libm.a	Mathematical library functions.
/lib/libc.a	Standard C library functions (**fabs** part of this library).

SEE ALSO

abs(3c).

NAME

gamma — log gamma function

SYNOPSIS

#include <math.h>

double gamma (x)
double x;

DESCRIPTION

Gamma returns $\ln \left| \Gamma(\left| x \right|) \right|$. The sign of $\Gamma(\left| x \right|)$ is returned in the external integer *signgam*.

EXAMPLES

The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.0)
        error();
y = exp(y);
if(signgam)
        y = -y;
```

FILES

/usr/lib/libm.a	Mathematical library functions.

DIAGNOSTICS

A huge value is returned for negative integer arguments.

CAVEATS

There should be a positive indication of error.

NAME

hypot, cabs — Euclidean distance

SYNOPSIS

#include ⟨**math.h**⟩

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

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

DESCRIPTION

Hypot and **cabs** return

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

taking precautions against unwarranted overflows.

The *x* and *y* arguments to the complex absolute value function **cabs** consist of the real and imaginary parts, respectively.

FILES

/usr/lib/libm.a	Mathematical library functions.
/usr/lib/libF77.a	F77 intrinsic function (math) and startup library.

SEE ALSO

exp(3m).

NAME

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

SYNOPSIS

#include ⟨math.h⟩

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
double x;

DESCRIPTION

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

FILES

/usr/lib/libm.a	Mathematical library functions.

DIAGNOSTICS

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

SEE ALSO

intro(2), perror(3f).

NAME

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

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double tan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x, y)
double x, y;
```

DESCRIPTION

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

Asin returns the arc sin of x in the range $-\pi/2$ to $\pi/2$.

Acos returns the arc cosine of x in the range 0 to π.

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

Atan2 returns the arc tangent of x/y in the range $-\pi$ to π.

FILES

/usr/lib/libm.a	Mathematical library functions.

DIAGNOSTICS

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

SEE ALSO

intro(3m).

NAME

sinh, cosh, tanh — hyperbolic functions

SYNOPSIS

#include ⟨math.h⟩

double sinh (x)

double cosh (x)
double x;

double tanh (x)
double x;

DESCRIPTION

The functions **sinh**, **cosh**, and **tanh** compute the designated hyperbolic functions for real arguments.

FILES

/usr/lib/libm.a	Mathematical library functions.

DIAGNOSTICS

Sinh and **cosh** return a huge value of appropriate sign when the correct value would overflow.

SEE ALSO

intro(3m).

NAME

itom, mcmp, move — multiple precision integer assignment and comparison

SYNOPSIS

cc ... -lmp

#include <mp.h>

MINT *itom(n)
int n;

int mcmp(a, b)
MINT *a, *b;

move (a, b)
MINT *a, *b;

DESCRIPTION

These routines perform various functions on integers of arbitrary length. The integers are stored using the defined type MINT, which is found in */usr/include/mp.h*.

The function **itom** is used to initialize a multiple precision integer. The value of the parameter *n* is stored in a newly–allocated structure. The return value is a pointer to this structure.

The function **mcmp** is used to compare two multiple precision integers. The return value is 0 if the two are equal, greater than 0 if the first argument is greater than the second, and less than 0 otherwise.

The subroutine **move** is the assignment operation. The value of the first argument is copied to the second argument.

CAVEATS

The argument given to **itom** is an integer, but the value must fit into a short integer (between –32768 and 32767). Other values will result in strange behavior. This is only true for initialization. Other routines work correctly for any size value.

SEE ALSO

cc(1), intro(3), madd(3mp), mout(3mp).

NAME

madd, msub, mult, mdiv, pow, rpow, gcd, invert, msqrt, sdiv — multiple precision integer arithmetic

SYNOPSIS

cc ... -lmp

#include ⟨mp.h⟩

madd(a, b, c)
msub(a, b, c)
mult(a, b, c)
mdiv(a, b, q, r)
rpow(a, b, c)
pow(a, b, m, c)
gcd(a, b, c)
invert(a, b, c)
msqrt(a, b, r)
MINT *a, *b, *c, *m, *q,

sdiv (a, n, q, r)
MINT *a, *q;
int n;
short *r;

DESCRIPTION

These routines perform arithmetic on integers of arbitrary length. The integers are stored using the defined type MINT, which is found in */usr/include/mp.h.*

The subroutine **madd** calculates the sum of its first two arguments and places the result in the third.

The subroutine **msub** calculates the difference of its first two arguments and places the result in the third.

The subroutine **mult** calculates the product of its first two arguments and places the result in the third.

The subroutine **mdiv** divides its first two arguments and places the quotient in the third and the remainder in the fourth.

The subroutine **rpow** calulates the first argument raised to the power of the second argument and places the result in the third argument.

The subroutine **pow** calulates the first argument raised to the power of the second argument modulo the third argument, and places the result in the fourth argument.

The subroutine **gcd** calculates the greatest common divisor of the first two arguments and places the result in the third.

The subroutine **invert** calculates the modular inverse of the first argument modulus the second argument and places it in the third.

The subroutine **msqrt** calculates the closest perfect square that is less than or equal to the first argument and places the square root in the second argument. The difference between the first argument and the perfect square found is placed in the third argument. Thus, ('arg2' * 'arg2') + 'arg3' = arg1.

The subroutine **sdiv** is the same as **mdiv,** except that the divisor is an integer instead of a multiple precision integer.

CAVEATS

All arguments should be initialized by using the subroutine *itom(3mp).*

SEE ALSO

cc(1), intro(3), itom(3mp), mout(3mp).

NAME

mout, min, omout, omin, fmout, fmin — multiple precision integer input/output

SYNOPSIS

cc ... -lmp

#include <stdio.h>
#include <mp.h>

mout(a)
min(a)
omout(a)
omin(a)
fmout(a, fp)
fmin(a, fp)
MINT *a;
FILE *fp;

DESCRIPTION

These routines are used to input and output on integers of arbitrary length. The integers are stored using the defined type MINT, which is found in */usr/include/mp.h.*

The subroutine **mout** prints the value of the given multiple precision integer on the standard output, followed by a newline.

The subroutine **min** reads an integer from the standard input and places the value in the argument. Spaces, tabs and backslashes (\) are ignored in the input, and any '–' character negates the number. Thus, the string "–12 5–" is interpreted as 125. If a character other than 0–9, space, tab, newline, backslash, or '–' is encountered, the character is placed back in the input via *ungetc(3s),* and is ignored. Upon end of file, the value EOF is returned.

The subroutines **omout** and **omin** work the same as **mout** and **min,** except that the conversions are done in octal. **Omin** does not check to see if the input contains digits greater than 7.

The subroutines **fmout** and **fmin** work the same as **mout** and **min,** except that the given FILE pointer is used for input and output.

SEE ALSO

cc(1), intro(3), itom(3mp), madd(3mp).

NAME

intro — introduction to network library functions

DESCRIPTION

The functions in this library are applicable to the Internet domain.

SEE ALSO

intro(3).

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 SUN, 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 *gethostent(3n)* and *getservent(3n)*.

CAVEATS

The VAX handles bytes backwards from almost everyone else in the world. This is not expected to be fixed in the near future.

SEE ALSO

gethostent(3n), getservent(3n).

NAME

gethostent, gethostbyaddr, gethostbyname, sethostent, endhostent — get network host entry

SYNOPSIS

#include <netdb.h>

struct hostent *gethostent()

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;

sethostent(stayopen)
int stayopen;

endhostent()

sethostsock(stayopen)
int stayopen;

endhostsock()

DESCRIPTION

Gethostent, **gethostbyname**, and **gethostbyaddr** each return a pointer to an object with the following structure.

```
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;        /* address */
};
```

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 A pointer to the network address for the host. Host addresses are returned in network byte order.

Gethostent reads the next line of the file, */etc/hosts*, opening the file if necessary. The file remains open upon completion.

Gethostbyname and **gethostbyaddr** open a UTek domain socket (*/tmp/nameserver*) to the *nameserver(8n)*, if necessary, then make a request and get an answer. The socket is closed upon completion. Host addresses are supplied in network order.

Sethostent opens and rewinds the file. If the **stayopen** flag is nonzero, the host database will not be closed by subsequent calls to **endhostent**.

Endhostent closes the file.

Sethostsock opens the socket to the *nameserver(8n)*. If the **stayopen** flag is nonzero the socket will not be closed until **endhostsock** is called.

Endhostsock closes the socket.

FILES

/etc/hosts

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

CAVEATS

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.

SEE ALSO

hosts(5n).

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,addrtype)
long net;
int addrtype;

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 database, */etc/networks*.

```
struct  netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net address type */
        int     n_net;          /* network # */
};
```

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 host byte order.

Getnetent reads the next line of the file, opening the file if necessary. The file remains open upon completion.

Getnetbyname and **getnetbyaddr** open the file, if necessary, then sequentially search from the beginning of the file until a matching net name or net address and address family is found, or until EOF is encountered. The file is closed upon completion. Network numbers are supplied in host byte order.

Setnetent opens and rewinds the file. If the **stayopen** flag is nonzero, the net database will not be closed by subsequent calls to **endnetent** (either directly, or indirectly through one of the other **getnet** calls).

Endnetent closes the file.

FILES

/etc/networks

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

CAVEATS

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.

SEE ALSO

networks(5n).

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 database, */etc/protocols*.

```
struct  protoent {
        char    *p_name;        /* official protocol name */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol # */
};
```

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. The file remains open upon completion.

Getprotobyname and **getprotobynumber** open the file, if necessary, then sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered. The file is closed upon completion.

Setprotoent opens and rewinds the file. If the **stayopen** flag is nonzero, the protocol database will not be closed by subsequent calls to **endprotoent** (either directly, or indirectly through one of the other **getproto** calls).

Endprotoent closes the file.

FILES

/etc/protocols

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

CAVEATS

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.

SEE ALSO

protocols(5n).

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 service name */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port # */
        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. The file remains open upon completion.

Getservbyname and **getservbyport** open the file, if necessary, then 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. The file is closed upon completion. Port numbers are supplied in network byte order.

Setservent opens and rewinds the file. If the **stayopen** flag is nonzero, the service database will not be closed by subsequent calls to **endservent** (either directly, or indirectly through one of the other **getserv** calls).

Endservent closes the file.

FILES

/etc/services

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

CAVEATS

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

SEE ALSO

getprotoent(3n), services(5n).

NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof — internet address manipulation routines

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

u_long inet_addr(cp)
char *cp;

u_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 dot (.) 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 dot (.) 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 dot (.) notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32–bit integer quantity on

the VAX, the bytes referred to above appear as `d.c.b.a`. That is, VAX bytes are ordered from right to left.

When a three part address is specified, the last part is interpreted as a 16–bit quantity and placed in the right most two 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 right most three 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 dot (.) notation may be decimal, octal, or hexadecimal, as specified in the C language (for example, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The value —1 is returned by **inet_addr** and **inet_network** for malformed requests.

CAVEATS

The string returned by **inet_ntoa** resides in a static memory area so it must be copied if it is to be saved.

SEE ALSO

gethostent(3n), getnetent(3n), hosts(5n), networks(5n).

NAME

rcmd, rresvport, ruserok — routines for returning a stream to a remote command

SYNOPSIS

```
rem = rcmd(ahost, inport, locuser, remuser, cmd, fd2p);
char **ahost;
int 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 superuser 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 reserved port space. **Ruserok** is a routine used by servers to authenticate clients requesting service with **rcmd**. All three functions are used by the *rshd(8n)* server (among others).

Rcmd looks up the host **ahost* returning —1 if the host does not exist, or if some error occurred during or after setting up the connection. 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 of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is nonzero, 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 UTek 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; note that you may be able to get its attention by using out–of–band data.

The protocol is described in detail in *rshd(8n)*.

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 addresses consist of a port in the range 0 to 1023. Only the superuser is allowed to bind an address of this sort to a socket.

Ruserok takes a remote host's name, as returned by a *gethostent(3n)* routine, two user names and a flag indicating if the local user's name is the superuser. It then checks the files */etc/hosts.equiv* and, possibly, *.rhosts* in the current working directory (normally 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.

CAVEATS

There is no way to specify options to the **socket** call which **rcmd** makes.

SEE ALSO

rlogin(1n), rsh(1n), rexec(3n), hosts.equiv(5n), rlogind(8n), rshd(8n).

NAME

rexec — return stream to a remote command

SYNOPSIS

rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
int inport;
char *user, *passwd, *cmd;
int *fd2p;

DESCRIPTION

Rexec looks up the host **ahost* 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 or her 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(8n).*

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 nonzero, 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 UTek 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.

CAVEATS

There is no way to specify options to the **socket** call which **rexec** makes.

SEE ALSO

rcmd(3n), rexecd(8n), .rhosts(5n), hosts.equiv(5n).

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 in–line macros **getc** and *putc(3s)* handle characters quickly. The higher level routines **gets**, **fgets**, **scanf**, **fscanf**, **fread**, **puts**, **fputs**, **printf**, **fprintf**, and **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**, and **fileno**.

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.

CAVEATS

The standard buffered functions do not interact well with certain other library and system functions, especially **vfork** and **abort**.

SEE ALSO

open(2), close(2), read(2), write(2), fread(3s), fseek(3s).

NAME

fclose, fflush — close or flush a stream

SYNOPSIS

#include ⟨stdio.h⟩

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;

DESCRIPTION

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

Fclose is performed automatically upon calling *exit(3c)*.

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

DIAGNOSTICS

These routines return **0** on successful completion and **EOF** if buffered data cannot be transferred to the file.

SEE ALSO

close(2), exit(3c), fopen(3s), setbuf(3s).

NAME

feof, ferror, clearerr, fileno — stream status inquiries

SYNOPSIS

#include <stdio.h>

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream

clearerr(stream)
FILE *stream

fileno(stream)
FILE *stream;

DESCRIPTION

Feof returns nonzero when end–of–file is read on the named input *stream*; otherwise, it is zero.

Ferror returns nonzero when an error has occurred reading or writing the named *stream*; otherwise, it is zero. Unless cleared by **clearerr**, the error indication lasts until the stream is closed.

Clrerr resets the error indication on the named *stream.*

Fileno returns the integer file descriptor associated with the *stream*; see *open(2).*

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

SEE ALSO

fopen(3s), open(2).

NAME

fclose, fflush — close or flush a stream

SYNOPSIS

#include ⟨stdio.h⟩

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;

DESCRIPTION

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

Fclose is performed automatically upon calling *exit(3c)*.

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

DIAGNOSTICS

These routines return **0** on successful completion and **EOF** if buffered data cannot be transferred to the file.

SEE ALSO

close(2), exit(3c), fopen(3s), setbuf(3s).

NAME

fopen, freopen, fdopen — open a stream

SYNOPSIS

#include <stdio.h>

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

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

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

DESCRIPTION

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

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

r means *open for reading*

w means *create for writing*

a means *append: open for writing at end of file, or create for writing*

In addition, each *type* may be followed by a + to have the file opened for reading and writing. **r+** positions the stream at the beginning of the file, **w+** creates or truncates it, and **a+** positions it at the end. Both **reads** and **writes** may be used on **read/write** streams, with the limitation that an **fseek**, **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**, and **stderr**, to specified files.

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

DIAGNOSTICS

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

CAVEATS

Fdopen is not portable to systems other than UTek.

The **read/write** *types* do not exist on all systems. Those systems without **read/write** modes will probably treat the *type* as if the + was not present. These are unreliable in any event.

SEE ALSO

open(2), fclose(3s).

NAME

fread, fwrite — buffered binary input/output

SYNOPSIS

#include <stdio.h>

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

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

DESCRIPTION

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

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

DIAGNOSTICS

Fread and **fwrite** return 0 upon end–of–file or error.

SEE ALSO

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

NAME

fseek, ftell, rewind — reposition a stream

SYNOPSIS

#include <stdio.h>

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

long ftell(stream)
FILE *stream;

rewind(stream)

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, deturmined by whether *ptrname* has the value 0, 1, or 2.

Fseek undoes any effects of *ungetc(3s)*.

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

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

DIAGNOSTICS

Fseek returns —1 for improper seeks.

SEE ALSO

lseek(2), fopen(3s).

NAME

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

SYNOPSIS

#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;

DESCRIPTION

Getc returns the next character from the named input *stream*.

Getchar() is identical to **getc(stdin)**.

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

Getw returns the next word (in a 32–bit integer) 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.

DIAGNOSTICS

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

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

CAVEATS

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

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

SEE ALSO

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

NAME

gets, fgets — get a string from a stream

SYNOPSIS

#include <stdio.h>

char *gets(s)
char *s;

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

DESCRIPTION

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

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

DIAGNOSTICS

Gets and **fgets** return the constant pointer **NULL** upon end–of–file or error.

CAVEATS

Gets deletes a newline, and **fgets** keeps it, all in the name of backward compatibility.

SEE ALSO

puts(3s), getc(3f), scanf(3s), fread(3s), ferror(3s).

NAME

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

SYNOPSIS

#include <stdio.h>

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

pclose(stream)
FILE *stream;

DESCRIPTION

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

A stream opened by **popen** should be closed by **pclose**, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter, and a type **w** as an output filter.

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 **popen**ed command.

CAVEATS

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(3S)*.

Popen alwyas calls **sh**; it never calls **csh**.

SEE ALSO

pipe(2), fopen(3S), fclose(3S), system(3S), wait(2), sh(1sh).

NAME

printf, fprintf, sprintf — formatted output conversion

SYNOPSIS

#include ⟨stdio.h⟩

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

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

char *sprintf(s, format [, arg] ...)
char *s, format;

DESCRIPTION

Printf places output on the standard output stream **stdout**. The return value is 0 unless an error occurred, in which case EOF is returned.

Fprintf places output on the named output *stream*. The return value is 0 unless an error occurred, in which case EOF is returned.

Sprintf places *output* in the string *s*, followed by the character **\0**. The return value is a pointer to the string *s*.

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

Each conversion specification is introduced by the character **%**. Following the **%**, there may be, in this order

— An optional minus sign (—) which specifies *left adjustment* of the converted value in the indicated field.

— An optional digit string specifying a *field width;* if the converted value has fewer characters than the field width, it will be blank–padded on the left (or right, if the left–adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero–padding will be done instead of blank–padding.

— An optional dot (.) which serves to separate the field width from the next digit string.

— An optional digit string specifying a *precision* which specifies the number of digits to appear after the decimal point, for e– and f–conversion, or the maximum number of characters to be printed from a string

— An optional # character specifying that the value should be converted to an *alternate form*. For **c**, **d**, **s**, and **u** conversions, this option has no effect. 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 as they would otherwise be.

— The character **l** specifying that a following **d**, **o**, **x**, or **u** corresponds to a long integer *arg*.

— A character which indicates the type of conversion to be applied.

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

The conversion characters and their meanings are:

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

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

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

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

c The character *arg* is printed.

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

u The unsigned integer *arg* is converted to decimal and printed (the result will be in the range 0 through MAXUINT, where MAXUINT equals 4294967295 on a VAX-11 and 65535 on a PDP-11).

% Print 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. Characters generated by **printf** are printed by *putc(3s)*.

EXAMPLES

To print a date and time in the form *Sunday, July 3, 10:02*, where *weekday* and *month* are pointers to null-terminated strings:

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

To print π to five decimals:

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

CAVEATS

It is up the the programmer to ensure that the parameters passed to printf match the format string. Programming errors which cause a type mismatch may induce fatal runtime errors. Also, very wide specifier fields (>128 characters) fail.

SEE ALSO

putc(3s), scanf(3s), ecvt(3c).

NAME

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

SYNOPSIS

#include ⟨stdio.h⟩

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

putchar(c)

fputc(c, stream)
FILE *stream;

putw(w, stream)
FILE *stream;

DESCRIPTION

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

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

Fputc behaves like **putc**, but is a genuine function rather than a macro.

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.

DIAGNOSTICS

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

CAVEATS

Because it is implemented as a macro, **putc** treats a *stream* argument with side effects improperly. In particular

```
putc(c, *f++);
```

doesn't work sensibly.

Errors can occur long after the call to **putc**.

SEE ALSO

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

NAME

puts, fputs — put a string on a stream

SYNOPSIS

#include ⟨stdio.h⟩

puts(s)
char *s;

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

DESCRIPTION

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

Fputs copies the null–terminated string *s* to the named output *stream*.

Neither routine copies the terminal null character.

CAVEATS

Puts appends a newline, and **fputs** does not, all in the name of backward compatibility.

SEE ALSO

fopen(3s), gets(3s), putc(3s), printf(3s), ferror(3s) fread(3s).

NAME

scanf, fscanf, sscanf — formatted input conversion

SYNOPSIS

#include <stdio.h>

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

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

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

DESCRIPTION

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

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

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

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

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

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

d A decimal integer is expected; the corresponding argument should be an integer pointer.

e A floating point number is expected; the next field is converted
f accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e** followed by an optionally signed integer.

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

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

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

% A single **%** is expected in the input at this point; no assignment is done.

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

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

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

EXAMPLES

The following call with the input line

```
25   54.32E-1  thompson
```

will assign to *i* the value 25, *x* the value 5.432, and *name* will contain **'thompson\0'**:

```
int i; float x; char name[50];
scanf("%d%f%s", &i, &x, name);
```

In the next example, the input

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip 0123, and place the string 56\0 in *name*.

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

After this, the next call to **getchar** will return *a*.

DIAGNOSTICS

The **scanf** functions return **EOF** on end of input, and a short count for missing or illegal data items.

CAVEATS

The success of literal matches and suppressed assignments is not directly determinable.

SEE ALSO

atof(3C), getc(3S), printf(3S).

NAME

setbuf, setbuffer, setlinebuf — assign buffering to a stream

SYNOPSIS

#include <stdio.h>

setbuf(stream, buf)
FILE *stream;
char *buf;

setbuffer(stream, buf, size)
FILE *stream;
char *buf;
int size;

setlinebuf(stream)
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 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(3C)* 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 instead of an automatically allocated buffer. If *buf* is the constant pointer **NULL**, input/output will be completely unbuffered. A manifest constant **BUFSIZ** tells how big an array is needed. For example:

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

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 file descriptor 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**.

CAVEATS

The standard error stream should be line buffered by default.

The **setbuffer** and **setlinebuf** functions are not portable to non 4.2 BSD versions of UNIX.

SEE ALSO

fopen(3S), getc(3S), putc(3S), malloc(3C), fclose(3S), puts(3S), printf(3S), fread(3S).

NAME

stdio — standard buffered input/output package

SYNOPSIS

#include ⟨stdio.h⟩

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

DESCRIPTION

The functions described in Sections 3S constitute a user-level buffering scheme. The in-line macros **getc** and *putc(3s)* handle characters quickly. The higher level routines **gets**, **fgets**, **scanf**, **fscanf**, **fread**, **puts**, **fputs**, **printf**, **fprintf**, and **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(3c)* 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**.

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 *flush(3f)* the standard output before going off and computing so that the output will appear.

CAVEATS

The standard buffered functions do not interact well with certain other library and system functions, especially *vfork* and *abort.*

SEE ALSO

open(2), close(2), read(2), write(2), fread(3s), fseek(3s).

NAME

system — issue a shell command

SYNOPSIS

system(string)
char *string;

DESCRIPTION

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

DIAGNOSTICS

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

CAVEATS

It is very important to remember that **system** executes **sh**. This means that shell metacharacters such as '*' and '¦' are interpreted, even when present in filenames. This can cause undesirable results when **system** is used to execute commands with arguments provided by the user.

Since **system** executes **sh,** the overhead of executing **sh,** such as copying the environment into shell variables, causes **system** to be slower than simply using **fork** and **exec.**

For these reasons, it is best to use **system** in cases where the command to be executed contains shell variables and/or redirection and/or pipes, which are difficult to build in to a program.

SEE ALSO

execve(2), fork(2), wait(2), popen(3s).

NAME

ungetc — push character back into input stream

SYNOPSIS

#include ⟨stdio.h⟩

ungetc(c, stream)
FILE *stream;

DESCRIPTION

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

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

Fseek(3s) erases all memory of pushed back characters.

DIAGNOSTICS

Ungetc returns **EOF** if it cannot push a character back.

SEE ALSO

getc(3s), setbuf(3s), fseek(3s).

NAME

curses — screen functions with "optimal" cursor motion

SYNOPSIS

#include <curses.h>
#include <termcap.h>

cc [flags] filenames —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.

FUNCTIONS:

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

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

CAVEATS

The size of the termcap entry buffer in this system is required to be TCAPSIZ (defined in **termcap.h**). If this is not true, the program may get a memory fault.

SEE ALSO

getenv(3c), getenv(3f), termcap(5t).

NAME

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

SYNOPSIS

```
#include <termcap.h>
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(5t)*. These are low level routines; see *curses(3t)* 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 TCAPSIZ (defined in **termcap.h**) 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 pathname 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 cannot **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; it returns 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(5t),* 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**, ⟨CTRL–D⟩, or ⟨CTRL–@⟩ 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 ⟨CTRL–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 *ioctl(2).* See the manual page for *tty(4)* for information on the output speed. The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null (⟨CTRL–@⟩) is inappropriate.

FILES

/usr/lib/libtermcap.a	—ltermcap library.
/etc/termcap	Database.

CAVEATS

In order to be able to work with longer **termcap** entries, TCAPSIZ is 2048. Programs which do not use this size may get memory faults.

SEE ALSO

ex(1), curses(3t), tty(4), termcap(5t).

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*, *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 *config(8)* program. The DIAGNOSTICS section lists messages which may appear on the console and in the system error log */usr/adm/messages* due to errors in device operation.

PROTOCOLS

The system currently supports only the DARPA Internet protocols fully. Raw socket interfaces are provided to IP protocol layer of the DARPA Internet. 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:

```
#define   AF_UNIX          1      /* local to host (pipes, streams) */
#define   AF_INET          2      /* internetwork: UDP, TCP, etc. */
```

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) */
```

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, each network interface autoconfigured installs a routing table entry when it wishes to have packets sent through it. Normally the interface specifies the route through it is 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 may be requested to address the packet to an entity different from the eventual recipient (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 resources associated with it will not be reclaimed until further references to it are released.

The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existant 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. This value is used to select among multiple routes to the same destination. When multiple routes to the same destination exist, the least used route is selected.

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(4n)*, do not.

At boot time each interface which has underlying hardware support makes itself known to the system during the autoconfiguration process. Once the interface has acquired its address it is expected to install a routing table entry so that messages may be routed through it. Most interfaces require some part of their address specified with an SIOCSIFADDR ioctl before they will allow traffic to flow through them. On interfaces where the network-link layer address mapping is static, only the network number is taken from the ioctl; the remainder is found in a hardware specific manner. On interfaces which provide dynamic network-link layer address mapping facilities (e.g. 10Mb/s Ethernets), the entire address specified in the ioctl is used.

The following *ioctl* calls may be used to manipulate network interfaces. 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. "ln
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
```

```
                short   ifru_flags;
        } 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_flags       ifr_ifru.ifru_flags     /* flags */
};
```

SIOCSIFADDR
: Set interface address. If the family is AF_INET then the Internet address is set; family AF_UNSPEC sets the hardware address (Ethernet). Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR
: Get interface address.

SIOCSIFDSTADDR
: Set point to point address for interface.

SIOCGIFDSTADDR
: Get point to point address for interface.

SIOCSIFFLAGS
: Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified.

SIOCGIFFLAGS
: Get interface flags.

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), ioctl(2), config(8), lna(4N), routed(8N).

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 all the 10Mb/s Ethernet interface drivers.

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 messages are 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 enable 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         2       /* completed entry (arp_ha valid) */
#define ATF_PERM        4       /* permanent entry */
#define ATF_PUBL        8       /* publish (respond for other 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*and*ATF_PUBL*. *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 4 (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 Sun to act as an " **ARP** server" which may be useful in convincing an **ARP**–only machine to talk to a non–**ARP** machine.

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.

CAVEATS

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.

SEE ALSO

inet(4n), arp(8n), ifconfig(8n).

NAME

cvt — table of kernel symbols

DESCRIPTION

Cvt is a special file in kernel memory containing a table of names and values of kernel symbols. The table is accessed using the *knlist(3c)* subroutine and provides a fast way to obtain current values for kernel symbols.

The table format consists of a magic number identifying the table (0401), a value giving the size of the table, and a value and name field for each symbol included in the table. The magic number, the size and the symbol values are of type "long"; names are null–padded strings aligned on four–byte boundaries. The table is terminated with a null name (four bytes of 0's where a name field would normally be).

The table is built from a description file containing a list of kernel symbols to be included.

FILES

/dev/cvt	standard name of cvt table

SEE ALSO

knlist(3c).

NAME

df — standard floppy disk driver for 6130 System workstations

DESCRIPTION

The **df** device provides access to the standard on–board flexible disk drive. It uses 5 1/4 inch, double–sided, double density soft–sectored pre–formatted flexible disks. Formatted flexible disks contain 48 tracks per inch, 9 sectors per track and 512 bytes per sector. The disks are compatible with IBM PC disks.

Device naming conventions specify a two–letter description of the device, followed by the slot number, followed by the drive number. The slot number and drive number for integral devices are 0. The standard name for the flexible disk is **df00.** The flexible disk may also be accessed in character (raw) mode. The character device name is the block name with an "r" prefixed. Standard device names **df** and **rdf** are linked to df00 and rdf00, respectively, for ease of use.

It is possible to make make a file system on a floppy, using *mkfs*, and mount the file system, using *mount*. Care must be taken if this is done because the removal of a floppy diskette while it is *mount*ed may cause a loss of data on any diskette subsequently inserted in the floppy. It could even cause UTEK to panic.

I/O requests must begin on a sector boundary and must avoid going beyond the end of the disk.

FILES

/dev/df00, /dev/df	block files
/dev/rdf00, /dev/rdf	raw files

DIAGNOSTICS

The following errors may be returned.

[EBUSY]
Drive not ready (on a read or write).

[ENXIO]
Nonexistent drive (on open); or (on a read or write) offset is too large or not on a sector boundary.

[EIO]
A physical error other than "not ready".

SEE ALSO

open(2), mknod(2), diskpart(5), mkfs(8), mount(8).

NAME

ds — SCSI Winchester disk

DESCRIPTION

The files **/dev/ds***xyz* refer to the SCSI Winchester disk interface. The 61TC01 option 14 disk provides 40 Mb of storage in the same cabinet as the 61TC01 tape drive. The partition tables and overall operation of the SCSI disk are identical to those for the built–in disk.

The raw disk (**/dev/rds***xyz*) is formatted with 512 byte blocks. All *reads*, *writes* and *seeks* should be multiples of 512 bytes.

DIAGNOSTICS

ds*xyz*: no disk
: An attempt was made to access a nonexistant disk drive.

ds*xyz*: blown format
: The disk must be reformatted. See format(8).

ds*xyz*: hard error, sector *nnnnn*
: An unrecoverable disk error occurred. If the error is repeatable, the drive should be reformatted and the given address added to the defect list. This message is printed on the system console.

RETURN VALUE

If an error occurs, the call returns –1 and one of the following values is left in *errno*:

[ENODEV] You have attempted to open a nonexistant or unformatted drive.

[ENXIO] You have attempted to read or write beyond the end of the partition.

[EIO] An unrecoverable I/O error has occurred.

CAVEATS

Turning off the disk drive while it is in use may damage the file system or crash the workstation.

SEE ALSO

dump(8), *restore*(8), *format(8), mkfs(8), fstab(5).*

NAME

dw — Winchester disk driver for 6130 System workstations

DESCRIPTION

The **dw** device provides access to the integral Winchester disk drive. Device naming conventions specify a two letter description of the device, followed by the slot number, followed by the drive number, followed by any partition information. The slot number and drive number for integral devices are 0. The disk is normally divided into 16 disk partitions. The desired partition is indicated by minor device number 0 through 15. The partitions are named 'a' through 'p'. A partition on the standard Winchester disk is therefore accessed as **dw00[a–p]**

In block mode, the system accesses the disk via the normal file buffering mechanism and allows reading and writing without regard to physical disk records. A raw interface exists which provides one I/O operation per read or write call. The names of the raw files are prefixed with 'r'.

The following disk drives are supported: Micropolis 1302, Micropolis 1304, and Maxtor XT–1105. The Micropolis 1302 has 830 cylinders with 512 bytes/sector, 16 sectors/track and 3 tracks/cylinder. The Micropolis 1304 has 830 cylinders with 512 bytes/sector, 16 sectors/track and 6 tracks/cylinder. The Maxtor XT–1105 has 918 cylinders with 512 bytes/sector, 16 sectors/track and 11 tracks/cylinder.

I/O requests must begin on a sector boundary and must not go beyond the end of the disk. The **dw00** *a* partition is normally used for the root file system; and the **dw00** *b* partition as a paging area. Partitions *c* through *h* are considered data partitions and are currently not defined. Partitions *i* through *k* are reserved for further use by UTek. Partition *l* covers partitions *a* through *h* for easier referencing; partition *m* is used by the UTek diagnostics; *n* is used as the defect partition (it contains the manufacture's defect data). *o* is the maintainence partition; and *p* references the entire disk. When accessing partitions *a* through *k,* a bad sector replacement scheme is used to present a "perfect" disk. For partitions *l* through *p,* no such scheme is used.

FILES

/dev/dw00[a-p]	block files
/dev/rdw00[a-p]	raw files

DIAGNOSTICS

The following errors may be returned.

[EBUSY]
Drive not ready (on a read or write).

[ENXIO]
Nonexistent drive (on open); or (on a read or write) offset is too large or not on a sector boundary.

[EIO]
A physical error other than "not ready".

NAME

gins — GPIB instrument controller

SYNOPSIS

#include ⟨**box/gpib.h**⟩
#include ⟨**sys/ioctl.h**⟩

DESCRIPTION

This section describes both the *instrument control* special files and the overall **GPIB** system organization.

System organization.

Each **GPIB** interface is associated with one *configuration* special file (*device*), one *interface* special file (*device*), and up to 15 *instrument controller* special files (*devices*). *Configuration* devices are used by the **GPIB**utilities *gpconf(1)*, *gpinit(1)*, and *gprm(1)* to set up the *instrument controller* devices. They are described in *gpid(4)*. *Interface* devices support all **GPIB**operations. They are also described in *gpib(4)*. *Instrument control* devices provide device–independent access to a single **GPIB**instrument. They are created by the *gpconf(1)* utility, and may be given whatever names the user desires. *Instrument control* devices are described below.

Applications which require the special features of the **GPIB**, such as the ability to initiate a transfer in which the controller does not particitate, should access the bus through the *interface* special files described in *gpib(4)*. Most applications do not need such features and should access individual instruments through the *instrument control* devices.

The *instrument control* devices provide a device–independent means of communicating with **GPIB**instruments. In many cases, standard commands such as *cat(1)* and *echo(1)* can be used to perform simple control tasks without further programming. The system call *ioctl(2)* calls configure the device and send interface messages. The system calls *read(2)* and *write(2)* transfer device–dependent data without interpretation. The **GPIB** is a message–oriented system; each **read** or **write** transfers a single message. Note that this precludes the use of *stdio(3s)* which assumes a stream–oriented device, although the function *sscanf(3s)* can be used to format messages in a local buffer.

Configuration.

Instrument control devices are initially configured by the *gpconf(1)* utility. Applications which need to change the device configuration "on the fly" may use the GIOCGCONF and GIOCSCONF *ioctl(2)* described in *gpib(4)*. Specific differences in these calls are listed here:

Addr specifies the *instrument's* **GPIB**address. The primary address is stored in the high byte. The secondary address is stored in the low byte. Setting the primary address to NOADRS effectively removes the device from the system.

The SCAS, STD1, VSTD1, TCSYNC, and PPST **flags** are ignored. TRDMA, EXCL, ASYNC, and NDELAY behave as described in *gpib(4)*. One new **flag** is added:

POLLME Enable polling of this instrument on service request. See the description of the SRQ interrupt, below.

Interface messages.

ioctl(fd, GIOCCMD, cmd)
char cmd;

Assert ATN and send **cmd** on the **GPIB**. **Cmd** must be one of the addressed commands defined in ⟨*box/gpib.h*⟩. The driver will add the appropriate device addresses just as it does for *read(2)* and *write(2)*.

ioctl(fd, GIOCSPOLL, status)
char *status;

Serial poll the instrument. If the POLLME **flag** is set and the instrument has requested service since the last GIOCSPOLL or GIOCGSTAT, the driver will return the saved **status** rather than polling the instrument again.

Device-dependent messages.

The system calls *read(2)* and *write(2)* transfer device–dependent messages. Each *read* transfers exactly one message. If the message is longer than the buffer, the remaining bytes will be discarded. The **term** field of the **gpibstat** structure (described below) records the termination condition for the last transfer.

The driver sends the appropriate talk and listen addresses before each transfer and UNT UNL afterwards.

Status and signals.

The GIOCGSTAT *ioctl(2)* returns the **gpibstat** structure described in *gpib(4)* . Specific differences in this structure are listed here:

Intr records only the following events:

SRQ Service request. If the POLLME **flag** is set, the driver will poll this instrument whenever it receives a bus SRQ message. If this instrument is requesting service, the driver will set the SRQ bit and update **status**.

LOST The instrument has requested service more than once. Only the most recent status byte is saved.

If the ASYNC **flag** and the corresponding bit in the **mask** are set, the driver will send a signal (SIGURG) to the associated **pgrp**. A service request will also wake up processes waiting for an exception in *select(2)*. As a special case, if the POLLME **flag** is cleared, a **select** on an

exception will enable polling until the device first requests service. At this time, further polling will be disabled, the select will return, and a GIOCGSTAT *ioctl* will report SRQ and the device status.

Status is the *instrument*'s last reported serial poll status.

Interaction with interface devices.

The driver arbitrates access to the interface hardware between the *interface* device and any *instrument control* devices present. The driver guarantees that there will be no interruptions during an *instrument control* device operation (address—transfer—unaddress); in most cases, this will be sufficient to prevent problems. You may use the GIOCASGN and GIOCRELSE *ioctls* to prevent other devices from interrupting sequences of operations.

SEE ALSO

gpconf(1), gpinit(1), gprm(1), gpstat(1), close(2), fcntl(2), ioctl(2), open(2), read(2), select(2), sigvec(2), write(2), signal(3c), gpib(4), gpid(4), gins(4), config(8).

NAME

gpib — GPIB interface driver

SYNOPSIS

#include ⟨box/gpib.h⟩
#include ⟨sys/ioctl.h⟩

DESCRIPTION

This section describes both the special files */dev/gpibn* and the overall **GPIB** system organization.

System organization.

Each **GPIB** interface is associated with one *configuration* special file (*device*), one *interface* special file (*device*), and up to 15 *instrument controller* special files (*devices*). *Configuration* devices are used by the **GPIB** utilities *gpconf(1)*, *gpinit(1)*, and *gprm(1)* to set up the *instrument controller* devices. They are described in *gpid(4)*. *Interface* devices support all **GPIB** operations. They are fully described below. *Instrument control* devices provide device–independent access to a single **GPIB** instrument. They are created by the *gpconf(1)* utility, and may be given whatever names the user desires. *Instrument control* devices are described in *gins(4)*.

Applications which require the special features of the **GPIB**, such as the ability to initiate a transfer in which the controller does not particitate, should access the bus through the *interface* devices described here. Most applications do not need such features and should access individual instruments through the *instrument control* devices.

The */dev/gpibn* devices allow direct control of the **GPIB** interface. *Ioctl(2)* calls configure the interface and send interface messages. *Read(2)* and *write(2)* transfer device–dependent data without interpretation. The **GPIB** is a message–oriented system; each **read** or **write** transfers a single message. Note that this precludes the use of *stdio(3s)* which assumes a stream–oriented device, although the function *sscanf(3s)* can be used to format messages in a local buffer.

Configuration.

The GIOCSCONF *ioctl(2)* sets or modifies the configuration of the interface. GIOCGCONF returns the current settings. GIOCSCONF and GIOCGCONF take the following structure (defined in ⟨*box/gpib.h*⟩) as argument:

```
struct gpibconf {
        short           gc_addr;   /* GPIB address */
        struct timeval  gc_htime;  /* handshake timeout */
        struct timeval  gc_ptime;  /* serial poll timeout */
        short           gc_eom;    /* end-of-message byte */
        long            gc_flags;  /* device-dependent control flags */
        short           gc_pgrp;   /* process or group to signal */
        short           gc_mask;   /* interrupt mask */
        };
```

Addr specifies the *interface's* **GPIB** address. The primary address is stored in the high byte. The secondary address (low byte) is forced to NOADRS. Setting the primary address to NOADRS effectively removes the interface from the system.

Htime and **ptime** specify the time limits for command/data transfers and serial poll response, respectively. These values are the time allowed per byte transferred. The actual timeout may be up to twice as long as specified in certain cases. Setting the time to zero disables the timeout.

Eom is recognized as end-of-message during *read(2)*. The driver recognizes end-of-message, but does not modify the data. In particular, the end-of-message byte is not removed or changed. **Eom** can be set to -1 to disable end-of-message recognition. EOI will always terminate input.

Flags encodes various options:

SCAS This interface is the **GPIB** system controller.

STD1 Reduce the interface T1 delay from 2.2 us to 1.2 us.

VSTD1 Reduce the interface T1 delay to 600 ns for the second and following bytes of any device-dependent message.

TCSYNC Take control synchronously. Asynchronous assertion of ATN (TCSYNC cleared) may cause data to be lost.

PPST Instrument status for parallel poll.

TRDMA Device-dependent data should be sent using DMA hardware, if present.

POLLME (Not used by the *interface* device.)

EXCL Block further *open(2)* calls on this device. This bit may also be set by the TIOCEXCL and cleared by the TIOCNXCL *ioctls*.

ASYNC Request asynchronous notification when the device status changes. See the discussion of status and signals below. ASYNC may also be set/cleared by the *fcntl(2)* system call or the FIOASYNC *ioctl*.

NDELAY Set the driver into non-blocking mode. *Read(2)*, *write(2)*, and *ioctl(2)* will return EWOULDBLOCK if the interface is busy. This bit may also be set/cleared by the FIONBIO *ioctl* or the *fcntl(2)* system call.

WEOI By default, this bit is turned on by the **GPIB** configuration utility *GPCONF(1)*. If a program turns this bit off and then writes data, EOI won't be asserted with the last byte of the message.

Pgrp is the process group to be signalled when the driver is ready for I/O (SIGIO) or flags an exception (SIGURG). Note that the ASYNC **flag** must be set to enable signalling. **Pgrp** defaults to the first process to open the device. It may also be set/tested with the TIOCSPGRP/TIOCGPGRP *ioctls*.

Mask may be used to disable signals from certain events. See the discussion of status and signals below.

Interface messages.

ioctl(fd, GIOCIFC, atv)
struct timeval *atv;

Send interface clear and take control. GIOCIFC will return an error if this is not the system controller. **Atv** may be NULL, in which case IFC is asserted for some default interval (0.5 second).

ioctl(fd, GIOCREN, aren)
int *aren;

Set the state of the interface REN line. GIOCREN will return an error if this is not the system controller.

ioctl(fd, GIOCCMD, cmd)
char *cmd;

Assert ATN and send *cmd* on the **GPIB** . Addressed, universal, and secondary commands are defined in ⟨*box/gpib.h*⟩. **Cmd** may point to a null character ('\0') to take control (assert ATN) without sending any command. The null will not be sent.

ioctl(fd, GIOCWEND, atv)
struct timeval *atv;

Drop ATN and wait for the data transfer to complete. There is no default time value. **Atv** specifies the maximum time to be allowed for the transfer.

ioctl(fd, GIOCSPOLL, status)
char *status;

Drop ATN and read one byte from the **GPIB**. This call is used to read the serial poll status from a previously addressed device. You must use GIOCSPOLL to read the serial poll status from a device; attempting to use *read(2)* may hang the **GPIB**.

ioctl(fd, GIOCPPOLL, status)
char *status;

Assert ATN with EOI (**GPIB** IDY message) and return the **GPIB** data.

Device-dependent messages.

Read(2) and *write(2)* transfer device–dependent messages. Each **read** transfers exactly one message. If the message is longer than the buffer, the remaining bytes will be discarded. The **term** field of the **gpibstat** structure (described below) records the termination condition for the last transfer.

Status and signals.

The GIOCGSTAT *ioctl(2)* returns the following structure (defined in ⟨*box/gpib.h*⟩):

```
struct gpibstat {
        short           gs_intr;        /* pending interrupts */
        char            gs_status;      /* device status byte */
        char            gs_term;        /* last I/O terminator */
        long            gs_state;       /* interface state */
};
```

Intr records the following events:

SRQ — Service request. This bit will remain set as long as the **GPIB** SRQ message is true.

LOST — (Not used by the *interface* device.)

RLC — Remote/local status change. The current status is part of the interface **state**.

TCT — Take control message received. Control is being passed to this interface. You must accept control (GIOCACPT *ioctl*) before attempting any I/O.

DCAS — Device clear active state. The driver interrupts any active I/O and sets this flag when the **GPIB** DCL or SDC message is received.

SPAS — Serial poll active state. The controller has polled this interface and the status may be changed.

GET — Device trigger message received.

IFC — Interface clear message received.

MTA — My talk address. The interface has entered the talker active state.

MLA — My listen address. The interface has entered the listener active state.

IWANT — I want control. This bit is provided for the benefit of certain shared–control applications. It is set by a **read** or **write** to an *instrument control* device if this interface is not the controller–in–charge.

When one of these events occurs, the status bit is set. SRQ, TCT, and IWANT stay active until the associated condition goes away. Other bits are cleared when read.

If the ASYNC **flag** and the corresponding bit in the **mask** are set, the driver will send a signal to the associated **pgrp**. MTA and MLA send SIGIO; all other interrupts send SIGURG. An interrupt will also wake up a process waiting in *select(2)*. MTA and MLA will wake up a process waiting for write or read, respectively. Other interrupts will wake up processes waiting for exceptions.

Status is this *interface*'s serial poll status, set by the GIOCSSTAT or GIOCRSV *ioctls*(2). It is not generally useful.

Term records the termination status for the last I/O operation:

END EOI was asserted with the last byte placed in the buffer.

EOM The last byte placed in the buffer was the end–of–message byte.

ERR There were no active listeners on the bus. (The *write(2)* returns EIO.)

TIME The transfer timed out. (The *read(2)* or *write(2)* returns EIO.)

INTR The transfer was interrupted by an interface message which changed the addressed state of the interface.

State records the current addressed state of the interface, along with some other values. See ⟨*gpib.h*⟩ for a complete list.

Interaction with instrument control devices.

The driver arbitrates access to the interface hardware between the *interface* device and any *instrument control* devices present. To prevent interference from other users, you must **assign** the **GPIB** during each sequence of operations.

ioctl(fd, GIOCASGN, was)
int *was;

If **was** is non–null, GIOCASGN returns the previous (assigned/not assigned) state. If the NDELAY flag is set and the interface is busy, GIOCASGN will return EWOULDBLOCK.

ioctl(fd, GIOCRELSE, NULL)

Release the **GPIB** for other use.

SEE ALSO

gpconf(1), gpinit(1), gprm(1), gpstat(1), close(2), fcntl(2), ioctl(2), open(2), read(2), select(2), sigvec(2), write(2), sigset(3J), gpib(4), gpid(4), gins(4), config(8).

NAME

gpid — GPIB configuration device

SYNOPSIS

#include ⟨**box/gpib.h**⟩
#include ⟨**sys/ioctl.h**⟩

DESCRIPTION

The **gpid** device is used by the **GPIB** utilities to create, modify, and remove the **GPIB** *instrument control* devices described in *gins(4)*. It is normally accessible only to the super–user. **Gpid** understands a modified form of the **GPIB** *ioctls GIOCGCONF* and *GIOCSCONF*; the low–order bits of the command (normally zero) are the internal unit number of the device to be configured.

SEE ALSO

gpconf(1), gpinit(1), gprm(1), gpstat(1), close(2), ioctl(2), open(2), gpib(4), gpid(4), gins(4), config(8).

NAME

hc — hard copy interface for 6130 System workstations

DESCRIPTION

Hc provides the interface to any Centronics type parallel interface devices such as line printers, hard copy units, plotters or other output devices. When the device is opened or closed, no page ejects are generated.

The unit number of the printer is specified by the minor device after removing the lowest byte which act as per–device parameters. Currently only the lowest three bits of the byte are interpreted. If bit 2 is set to 1, a carriage return is output before each newline character in the data buffer. If bit 1 is set to 0, all characters are "passed through" to the driver with no character editing. If bit 1 is set to 1, the device is treated as having a 64–character set, rather than a full 96–character set. In the resulting half–ASCII mode, the characters from columns 6 and 7 of the ASCII codes chart are directly translated to the corresponding row character in columns 4 and 5. The least significant bit of the minor device number is used to specify channel 0 (bit set to 0) or channel 1 (bit set to 1) of the interface.

An *ioctl* call is also available to change the device operating modes. The *request* value of the call is 0 to clear CRM and RCSM, 2 to set RCSM, 4 to set CRM, and 6 to set both modes. Each *ioctl* call will cause the modes to be cleared before the values of the *request* are set.

The driver does not interpret any control characters. It also does not make any assumptions about form width, form length, tab stop positions, etc.

FILES

/dev/hc

DIAGNOSTICS

ENXIO, EBUSY, EIO, ENOTTY

SEE ALSO

lpr(1), ioctl(2), mknod(8).

NAME

hs — high-speed serial interface to 4100 series Option 3C

SYNOPSIS

#include ⟨machine/hsio.h⟩
#include ⟨sys/ioctl.h⟩

DESCRIPTION

This device driver provides the capability to communicate with a Tektronix 4100 series terminal equipped with Option 3C or a standard Tektronix 4111 terminal. Used in conjunction with the standard RS-232 interface of the terminal, the High-speed Serial interface allows copying large volumes of pixel data and picture processor commands between workstation and terminal at rates up to one half megabit per second.

Part of the speed advantage over normal RS-232 communication is due to the data transmission facility itself. The **hs** device implements a simple block handshake protocol with CRC protection that insures reliable transmission over a serial data link using RS-422 electrical characteristics.

A further contributing factor in the long term transmission rate is the format of terminal data. Since the **hs** device is treated as a separate device by the terminal, the pixel and other display data does not have to conform to the 7-bit ASCII conventions (escape sequences) of normal RS-232 communications with a 4100 series terminal. Pixel data may be sent as a simple bit array. Vector and other drawing commands may be sent as "picture processor" instructions, eliminating the time consuming process of packing xy coordinates into 7-bit ASCII (and subsequent unpacking in the terminal).

The 4100 terminal treats the **hs** interface as a pseudo-device (DM:), from which data can be copied to the display as pixels (PX:) or picture processor commands (DS:) or to a particular segment (SG:). The copy operation is started by sending a "copy" command to the terminal. Until a copy command is sent to the terminal, the **hs** device will not be able to send or receive data. After a copy command is sent to the terminal, the terminal will not respond to RS-232 commands or data until the copy command terminates.

Reads and writes for the **hs** device should only follow a valid copy command. A "copy from DM:" at the terminal is associated with writing to the **hs** device. A "copy to DM:" is associated with reading the **hs** device. Once copying has started, **hs** device operation must be completed. Completion of a "copy from DM:" will normally result from the **hs** device sending an end-of-file indication. The **hs** device will send an end-of-file if the **hs** device is closed after writing or if the ioctl HSSIOCEOF is invoked. Completion of a "copy to DM:" will normally result from the terminal finding an end-of-file condition in the source device. The terminal will notify the **hs** device, which will return an end-of-file indication to the read system call. Once a "copy to DM:" has been initiated, the **hs** device should be read until an end-of-file is reached or

the terminal copy operation may not complete.

An alternative way to terminate a copy operation between the **hs** device and Option 3C is to "cancel" the operation. This can be done from the workstation by invoking the ioctl HSSIOCCANCEL. It can also be done from the terminal by pressing the *CANCEL* key. (Interface operations can not be reliably canceled from the terminal end. Under some circumstances, the terminal is unable to successfully cancel a copy command involving the `DM:` pseudo-device.)

The device may be opened for both read and write. However, it is not legal to change direction in mid-file. A write command may not follow a read unless the read returned end-of-file. Likewise, a read may not follow a write: it may follow an HSSIOCEOF ioctl.

For a more complete description of the factors involved in programming display operations involving the **hs** device and 4100 Option 3C, see the 4110/4120 Host Programming Manual. The manual includes the source listing of a demonstration program which copies pixel images between Utek files and the terminal screen.

Before using the **hs** device, disable the device as a login port (see the System Administration Manual). The **hs** device is an exclusive-use device, so it should be *assign*ed before using.

IOCTLS

HSSIOCEOF This ioctl sends an end of file to the terminal. HSSIOCEOF is not legal if the device is open for read.

HSSIOCCANCEL
This ioctl sends a cancel to the terminal. The effect is to shut down the current file transfer before the file has finished being written or read.

FILES

*/dev/hs, /dev/hs**
*/dev/tty**

DIAGNOSTICS

[EBUSY] Device already is use.

[EIO] A message was sent to the system console with the form:
igen*xxxxx***: hard I/O error, eioqual: 0x***n***
**The *xxxxx* will be either *read* or *write*. The value *n* will be one of the following:
1 hardware link down. (Check the cable.)
2 apparent protocol failure. (Probably hardware.)
5 retry count exceeded. (Bad connection.)

CAVEATS

If noise, bad connections, etc. are present, the link will slow down due to retries. (In some rare circumstances, a slower baud rate setting may deliver higher throughput.) If an exceptionally bad connection is used, link operations may terminate with an error due to retry exhaustion.

The link protocol used to connect the workstation and terminal delivers only correct bits. If you see things on the screen which are wrong, they are problems with the data being sent, not problems with the line or equipment.

SEE ALSO

4110/4120 SERIES HOST PROGRAMMING MANUAL
4110/4120 SERIES COMMAND REFERENCE MANUAL

NAME

ilan — intelligent local network interface

DESCRIPTION

The **ilan** interface provides access to a 10 Mb/s IEEE 802.3 (Ethernet) network on 6200 series workstations.

The host's Internet address is set from the on board NVRAM, but may be changed with an SIOCSIFADDR ioctl and family AF_INET. The host's IEEE 802.3 (Ethernet) address is also set from the NVRAM.

Use **netconfig**(8n) to change the Internet address. The Ethernet address should not be changed.

The **ilan** interface employs the address resolution protocol described in *arp*(4n) to dynamically map between Internet and Ethernet addresses on the local network.

A Time Domain Reflectometer measurement can be made with the SIOCTDR ioctl. The return value has the following form:

```
struct tdr {
     unsigned short tdr_ok:1,   /* link okay */
                    tdr_prb:1,  /* transceiver problem */
                    tdr_open:1, /* cable open */
                    tdr_srt:1,  /* cable shorted */
                    :1,
                    tdr_time:11;/* reflection time */

};
```

DIAGNOSTICS

ilan0: can't handle af%x. The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

CAVEATS

The current versions of the link controller and interface chip sometimes produce incorrect TDR results.

SEE ALSO

intro(4N), inet(4N), arp(4N), ioctl(2).

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.

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 address INADDR_ANY to effect "wildcard" matching on incoming messages.

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 not directly accessible.

CAVEATS

the Internet protocols develop. Users should not depend on details of the current implementation, but rather the services exported.

SEE ALSO

tcp(4N), udp(4N), ip(4n).

NAME

ip — Internet Protocol

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_RAW, 0);

DESCRIPTION

IP is the transport layer protocol used by the Internet protocol family. It may be accessed through a "raw socket" when developing new protocols, or special purpose applications. **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).

Outgoing packets automatically have an **IP** header prepended to them (based on the destination address and the protocol number the socket is created with). Likewise, incoming packets have their **IP** header stripped before being sent to the user.

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.

CAVEATS

One should be able to send and receive **IP** options.

The protocol should be settable after socket creation.

SEE ALSO

send(2), recv(2), inet(4n), intro(4n).

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.

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

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

lna — local network interface

DESCRIPTION

The **lna** interface provides access to a 10 Mb/s IEEE 802.3 (Ethernet) network on 6130 system workstations.

The host's Internet address is set from the on board NVRAM, but may be changed with an SIOCSIFADDR ioctl and family AF_INET. The host's IEEE 802.3 (Ethernet) address is also set from the NVRAM.

Use **netconfig**(8n) to change the Internet address. The Ethernet address should not be changed.

The **lna** interface employs the address resolution protocol described in *arp(4n)* to dynamically map between Internet and Ethernet addresses on the local network.

A Time Domain Reflectometer measurement can be made with the SIOCTDR ioctl. The return value has the following form:

```
struct tdr {
     unsigned short tdr_ok:1,   /* link okay */
                    tdr_prb:1,  /* transceiver problem */
                    tdr_open:1, /* cable open */
                    tdr_srt:1,  /* cable shorted */
                    :1,
                    tdr_time:11;/* reflection time */

};
```

DIAGNOSTICS

lna0: low on mbufs. A packet was dropped because the system was temporarily low on network buffers.

lna0: can't handle af%d. The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

CAVEATS

The current versions of the link controller and interface chip sometimes produce incorrect TDR results.

SEE ALSO

intro(4N), inet(4N), arp(4N), ioctl(2).

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. By default, the loopback interface is accessible at address 127.0.0.1 (non-standard); this address may be changed with the SIOCSIFADDR ioctl.

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.

CAVEATS

It should handle all address and protocol families. An approved network address should be reserved for this interface.

SEE ALSO

inet(4n), intro(4n).

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.

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

The file **kmem** is the same as **mem** except that kernel virtual memory rather than physical memory is accessed.

FILES

/dev/mem

/dev/kmem

SEE ALSO

brk(2).

NAME

mtio — magnetic tape interface for 6130 System workstations

SYNOPSIS

#include ⟨sys/types.h⟩
#include ⟨sys/mtio.h⟩

DESCRIPTION

A number of *ioctl* operations are available on raw magnetic tape. The following definitions are in ⟨*sys/mtio.h*⟩:

```
struct mtop {
        short   mt_op;      /* operations defined below */
        daddr_t mt_count;   /* how many of them */
};
```

Operations:

MTWEOF	write an end-of-file record
MTFSF	forward space file
MTBSF	backward space file
MTFSR	forward space record
MTBSR	backward space record
MTREW	rewind
MTOFFL	rewind and put the drive offline
MTNOP	no operation, sets status only
MTFSS	forward space sequential file marks
MTBSS	backward space sequential file marks
MTERA	erase tape
MTEND	move to end of media
MTTEN	re-tension tape

Not all operations are defined on all devices. The *type* field in the *mtget* structure below encodes some advisory information about drive capabilities. See the include file and the individual device manual pages for details.

Structure for **MTIOCGET** — mag tape get status command:

```
struct mtget {
        short   mt_type;    /* type of magtape device */
        short   mt_dsreg;   /* drive status (dev dependent)*/
        short   mt_erreg;   /* error (device dependent)*/
        short   mt_resid;   /* residual count */
        daddr_t mt_fileno;  /* reserved */
        daddr_t mt_blkno;   /* reserved */
};
```

The following values for *mt_type* are defined for 6130 System workstations.

MT_ISWANGTEK	Wangtek 5¼in cartridge tape drive
MT_ISXT	9–track tape

Mag tape IOCTL commands:

MTIOCTOP	do a mag tape operation
MTIOCGET	get tape status

FILES

/dev/*tc*	— cartridge tape interface
/dev/*mt*	— 9–track tape interface

CAVEATS

The status is not returned in a device independent format.

SEE ALSO

cpio(1), *dd*(1), *mt*(1), *tar*(1), *mt*(4), *tc*(4), *dump*(8), *restore*(8).

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

pty — pseudo terminal driver

SYNOPSIS

#include ⟨sys/ioctl.h⟩

DESCRIPTION

The *pty* driver provides support for a device-pair termed a *pseudo 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 device 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 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 *rlogind*(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*	master pseudo terminals
/dev/ttyp*	slave pseudo terminals

DIAGNOSTICS

None.

CAVEATS

It is not possible to send an EOT.

Pseudo terminals will be replaced by a more efficient mechanism in a future version of UTek.

NAME

rsa — on–board asynchronous interface

DESCRIPTION

The discussion of terminal I/O given in *tty(4)* applies to these devices.

Speed selections from B0 to B9600 as listed in *tty(4)* are available through the *ioctl(2)* system call. Speed selection B0 disables baud rate generation and drops data terminal ready (useful for forcing a dataset hang–up). Impossible speed changes are ignored.

In addition to the normal DC1/DC3 (CTRL–Q/CTRL–S) flow control, the RS–232–C lines *DTR* and *CTS* can be used. When *DTR* is unasserted (pin 20: –12v) by the terminal, then the driver will not transmit. conversely, when the driver's input buffer is near full, it will unassert *CTS* (pin 5) inhibiting the terminal from further transmission until the buffer is near empty and *CTS* is re–asserted. The use of these hardware lines can be individually enabled/disabled by the use of **tty** flags set/reset using the *ioctl(2)* system call and the commands **TIOC{GET,PUT,BIC,BIS}**. The appropriate flags are *DODTR* and *DOCTS*. These can be {re}set via *stty(1)* using the {-}*dtr* and {-}*cts* flags.

FILES

/dev/console
/dev/ttyo[0-1]

SEE ALSO

tty(4).

NAME

sc — raw SCSI interface

SYNOPSIS

#include ⟨sys/ioctl.h⟩
#include ⟨box/scsi.h⟩

DESCRIPTION

The files **/dev/sc***xy* refer to devices on the SCSI bus, where *x* specifies the interface slot number (1 through 6) and *y* specifies the SCSI device (0 through D). This provides for one or two drives (in SCSI parlance, *logical units*) on each of seven *controllers*. (The SCSI interface itself uses controller address 7.) For example, *sc57* refers to enhancement slot 5, controller 3, drive 1.

There are three *ioctl*s which send commands to the device. All three take the address of an *sccommand* structure (defined in *scsi.h*) as an argument. The structure provides for up to twelve command bytes and a pointer to any data to be moved. *SCIOCCMD* does not move any data; *cmdarg* and *cmdcount* are ignored. *SCIOCWCMD* writes data to the device. *SCIOCRCMD* reads data from the device. In each case, the driver will send the command to the device. If the device returns *BUSY* status, the driver will continue trying indefinitely. In such cases, it may be necessary to turn the SCSI device off or remove it from the bus to 'unhang' the driver. When the command completes, the driver will return 0 (successful completion) or −1 (error).

RETURN VALUE

If an error occurs, the call returns −1 and one of the following values is left in *errno*:

[ENODEV] You have attempted to access an interface or device which doesn't exist.

[EIO] The SCSI command returned with a *check* status. You should do a *request sense* command to find the reason for the error.

CAVEATS

There is no way to set the host interface address, share peripherals, or communicate between hosts.

There is no way to access logical units 2 through 7.

If a SCSI device hangs, it may be necessary to turn it off or unplug it from the SCSI bus to abort the command.

SEE ALSO

ds(4), *tc*(4).

NAME

tc — SCSI cartridge tape for 6130 System workstations

SYNOPSIS

#include ⟨sys/mtio.h⟩
#include ⟨box/tcreg.h⟩

DESCRIPTION

The files **/dev/tc***nn* refer to the SCSI cartridge tape interface. The tapes have nine tracks of data in QIC–24 format, providing up to 45Mb of storage. The driver normally rewinds the tape when closed; this may be suppressed (for example, if you intend to add data to a tape) by using the **/dev/ntc** device instead of the **/dev/tc** device.

Tapes are formatted with fixed length 512 byte blocks. All *reads* and *writes* should be multiples of 512 bytes. The streaming drive used in the tc device runs at 90 ips but takes several seconds to stop and restart. For this reason, large reads and writes (up to 256kb at a time) are preferred.

Reads and writes on tape are strictly sequential; seeks are ignored. The tape may be rewound or spaced forward with the operations described in *mtio*(4). Backward spacing is not allowed and attempts to backspace the tape may have peculiar results.

The *tc* device may not be opened for simultaneous reading and writing. Data may be written only at the beginning of the tape (erasing the old data) or at the end; existing data cannot be selectively erased or overwritten. The driver writes a file mark when closed after writing. Read returns a zero count when a file mark is read; the next read will fetch the first record of the next tape file.

DIAGNOSTICS

tc*nn*: can't update cartridge tape

An attempt was made to open the device for simultaneous read/write.

tc*nn*: write protected

An attempt was made to write on a write protected tape.

tc*nn*: no tape

An attempt was made to access a nonexistant tape drive or a drive without an installed tape cartridge.

tc*nn*: hard error, err = *value*⟨*bits*⟩

An unrecoverable tape error occurred. The error code is printed in hexadecimal with the bits symbolically decoded. This message is printed on the system console.

RETURN VALUE

Read(2) returns zero at a file mark.

If an error occurs, the call returns –1 and one of the following values is left in *errno*:

[ENODEV] The named device doesn't exist, or you have attempted to open a write–protected tape for writing.

[EBUSY] Some process has opened the device. No additional opens are allowed until the current user closes the device.

[ENXIO] You have attempted to write past the end of the tape.

[EIO] An unrecoverable read or write error has occurred.

CAVEATS

Programs which were written specifically for nine–track magtape may not work on cartridge tape because of the fixed block size. Other programs should be modified to buffer a reasonable amount of data with each *write*.

New tapes, tapes which have been in storage, or tapes which have been exposed to temperature or humidity changes should be retensioned twice before use. (The *mt* command provides an easy way to do this.) Failure to do this may result in unrecoverable read or write errors.

SEE ALSO

cpio(1), *dd(1)*, *mt(1)*, *tar(1)*, *mtio(4)*, *dump(8)*, *restore(8)*.

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.

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.

CAVEATS

It should be possible to send and receive **TCP** options. The system always tries to negotiate the maximum **TCP** segment size to be 1024 bytes. This can result in poor performance if an intervening network performs excessive fragmentation.

SEE ALSO

inet(4n), intro(4n).

NAME

tty — general terminal interface for 6130 System workstations

SYNOPSIS

#include <sgtty.h>

DESCRIPTION

NOTE: This **tty** interface might change in future releases.

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

Line disciplines.

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

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

new — A newer terminal driver, with features for job control; this must be used when using *csh(1csh)*.

net — A line discipline used for networking and loading data into the system over communications lines. It allows high speed input at very low overhead, and is described in *bk(4)*.

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

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

where LDISC is OTTYDISC for the standard **tty** driver, NTTYDISC for the new driver and NETLDISC for the networking discipline. 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 remainder of this section discusses the "old" and "new" disciplines.

The control terminal.

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

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

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

The association between a process and its control terminal can be broken by the TIOCNOTTY *ioctl(2)*. TIOCNOTTY does not close any existing file descriptors. Therefore, you can place the *ioctl* and still read and write from the file. But you no longer have a control terminal. Note that TIOCNOTTY has no effect if the **tty** you apply it to is not the control terminal.

A common use of TIOCNOTTY follows:

```
if((fd = open ("/dev/tty",O_WRONLY)) >= 0){
        ioctl (fd, TIOCNOTTY, NULL);
        close(fd);
}
```

TIOCNOTTY sets the process group to zero. The next time you open a terminal file, that file becomes the control terminal. This new association can be prevented by first setting the process group (see below) to some non-zero value. For example,

setpgrp(getpid());

Process groups.

Command processors such as *csh(1csh)* 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 terminals associated process group may be set using the TIOCSPGRP *ioctl(2)*:

ioctl(fildes, TIOCSPGRP, &pgrp)

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

Modes.

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

cooked — The normal mode. In this mode lines of input are collected and input editing is done. The edited line is made available when it is completed by a newline or when an EOT (control-D, hereafter ^D) is entered. A carriage return is usually made synonymous with newline in this mode, and replaced with a

newline whenever it is typed. All driver functions (input editing, interrupt generation, output processing such as delay generation and tab expansion, etc.) are available in this mode.

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

RAW This mode eliminates all input processing and makes all input characters available as they are typed; no output processing is done either.

The style of input processing can also be very different when the terminal is put in non-blocking i/o mode; see *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 UTek 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. When this limit is reached, no more input is accepted and the terminal bell is rung.

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 sections *Basic ioctl* and *Basic modes: sgtty* 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 TIOCSETN or TIOCSETP *ioctls* (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*

above and FIONREAD in *Basic ioctl* below.) No matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, line editing is normally done, with the character '^H' logically erasing the last character typed and the character '^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 '^V'; see below.

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 both the old and the new driver there is a literal–next character (default ^V) which can be typed in both cooked and CBREAK mode preceding **any** character to prevent its special meaning.

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.

Input echoing and redisplay

When a kill character is typed it is echoed followed by a new–line.

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

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

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

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

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

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

Output processing.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously–written characters have finished typing. (As noted above, input characters are normally echoed by putting them in the output queue as they arrive.) When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. Even parity is normally generated on output.

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

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 and output queues, TIOCFLUSH, is also available.

Hazeltine terminals

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

Flow control.

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

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(3c)* 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 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 (*a la* TIOCFLUSH). The characters shown here are the defaults; the field names in the structures (given below) are also shown. The characters may be changed, although this is not often done.

^C **t_intrc** (Delete) 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** (SUB) generates a SIGTSTP signal, which is used to suspend the current process group.

^Y **t_dsuspc** (EM) 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, is an *orphan process*, or is in the middle of process creation using *vfork(2)*), it is instead returned an end-of-file. (An *orphan process* is a process whose parent has exited and has been inherited by the *init(8)* process.) Under older UNIX systems these processes would typically have had their input files reset to */dev/null*, so this is a compatible change.

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

Summary of modes.

Unfortunately, due to the evolution of the terminal driver, there are 4 different structures which contain various portions of the driver data. The first of these (**sgttyb**) contains that part of the information largely common between version 6 and version 7 UNIX systems. The second contains additional control characters added in version 7. The third is a word of local state peculiar to the new terminal driver, and the fourth is another structure of special characters added for the new driver. In the future a single structure may be made available to programs which need to access all this information; most programs need not concern themselves with all this state.

Basic modes: sgtty.

The basic *ioctl*s use the structure defined in ⟨*sgtty.h*⟩:

```
struct sgttyb {
        char    sg_ispeed;/* input speed */
        char    sg_ospeed;/* output speed */
        char    sg_erase;/* erase character */
        char    sg_kill;/* kill character */
        short   sg_flags;/* mode flags */
};
```

The *sg_ispeed* and *sg_ospeed* fields describe the input and output speeds of the device according to the following table. NOTE: due to hardware limitations, *sg_ispeed*and*sg_ospeed* must be the same.

Symbolic values in the table are as defined in ⟨*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
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

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

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

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

ALLDELAY	0x0000FF00	Delay algorithm selection
BSDELAY	0x00008000	Select backspace delays (not implemented):
BS0	0x00000000	
BS1	0x00008000	
VTDELAY	0x00004000	Select form-feed and vertical-tab delays:
FF0	0x00000000	
FF1	0x00004000	
CRDELAY	0x00003000	Select carriage-return delays:
CR0	0x00000000	
CR1	0x00001000	
CR2	0x00002000	
CR3	0x00003000	
TBDELAY	0x00000C00	Select tab delays:
TAB0	0x00000000	
TAB1	0x00000400	
TAB2	0x00000800	
XTABS	0x00000C00	
NLDELAY	0x00000300	Select new-line delays:
NL0	0x00000000	
NL1	0x00000100	
NL2	0x00000200	

NL3	0x00000300	
EVENP	0x00000080	Even parity allowed on input (most terminals)
ODDP	0x00000040	Odd parity allowed on input
RAW	0x00000020	Raw mode: wake up on all characters, 8-bit interf
CRMOD	0x00000010	Map CR into LF; echo LF or CR as CR-LF
ECHO	0x00000008	Echo (full duplex)
LCASE	0x00000004	Map upper case to lower on input
CBREAK	0x00000002	Return each character as soon as typed
TANDEM	0x00000001	Automatic flow control
DODTR	0x02000000	Automatic flow control using DTR
DOCTS	0x08000000	Transmission conditional on CTS

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 0.08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about 0.16 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 (about 0.12 seconds) is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about 0.10 seconds. Type 3 is unimplemented and is 0.

Tab delay type 1 is about 0.1 seconds and and is tuned to the Teletype model 37. Type 2 is not implemented. 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 it is discarded; this applies to both new and old drivers.

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

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line; all processing is done except the input editing: character and word erase and line kill, input reprint, and the special treatment of \ or EOT are disabled.

TANDEM mode causes the system to produce a stop character (default is ^S) whenever the input queue is in danger of overflowing, and a start character (default is ^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.

DODTR and DOCTS provide a hardware flow control mechanism. DODTR works much like TANDEM in that when the buffer approaches being full, the DTR line is deasserted; when the buffer is emptied DTR is reasserted. DOCTS causes the USART to transmit or not depending on the state of the CTS pin.

Basic ioctls

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

#include ⟨sgtty.h⟩

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

The applicable codes are:

TIOCEXCL	Set "exclusive-use" mode: no further opens are permitted until the file has been closed.
TIOCGETP	Fetch the basic parameters associated with the terminal, and store in the pointed-to **sgttyb** structure.
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.
TIOCNXCL	Turn off "exclusive-use" mode.
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.
TIOCSTART	Simulates the use of ⟨CTRL-Q⟩.
TIOCSTOP	Simulates the use of ⟨CTRL-S⟩.
TIOCNOTTY	Breaks the association between a process and its control terminal.

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

TIOCSBRK the break bit is set in the terminal.

TIOCCBRK the break bit is cleared.

TIOCSDTR data terminal ready is set.

TIOCCDTR data terminal ready is cleared.

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

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

FIONREAD returns in the long integer whose address is arg the number of immediately readable characters from the argument unit. This works for files, pipes, terminals, and sockets.

The following call uses a different structure than do the previous calls:

#include ⟨sys/file.h⟩
ioctl(filedes, code, arg)
int *arg;

TIOCFLUSH
Flush all characters waiting in input and/or output queues, based on whether FREAD (input), FWRITE (output) or both have been set in the word pointed to by *arg*. If that word is 0, both input and output queues will be flushed. This last feature is provided for compatibility with 4.1c BSD.

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; except for the LNOHANG bit, this word is interpreted only when the new driver is in use. The bits of the local mode word are:

LCRTBS	000001	Backspace on erase rather than echoing erase
LPRTERA	000002	Printing terminal erase mode
LCRTERA	000004	Erase character echoes as BS–space–BS
LTILDE	000010	Convert ~ to ' on output (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
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
LDECCTQ	040000	Only ^Q restarts output after ^S, like DEC
LNOFLSH	0100000	Don't flush output on interrupt/suspend

The applicable *ioctl* functions are:

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

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

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

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

Local special chars

The final structure associated with each terminal is the *ltchars* structure, defined in ⟨*sys/ttychars.h*⟩, which defines interrupt 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 args is the address of a *ltchars* structure which defines the new local special characters.

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

FILES

/dev/tty

*/dev/tty**

/dev/console

CAVEATS

Half–duplex terminals are not supported.

SEE ALSO

csh(1csh), *stty*(1), *ioctl*(2), *sigvec*(2), *getty*(8), *init*(8).

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.

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

send(2), recv(2), inet(4n), intro(4n).

NAME

XD — XD Multibus adapter SMD disk driver

DESCRIPTION

Files with minor device numbers 0 through 15 refer to various portions of drive 0. The standard device names begin with **xdx** followed by the drive number and then a letter a–h for partitions 0–7 respectively. The origin and size (in sectors) of the default pseudo–disks on each drive are as follows:

CDC9715–340 partitions

disk	start	length	cyls
xdx0a	1	67584	1–88
xdx0b	89	67584	89–176
xdx0c	177	67584	177–264
xdx0d	265	67584	265–352
xdx0e	353	67584	353–440
xdx0f	441	67584	441–528
xdx0g	529	67584	529–616
xdx0h	617	71424	617–709

I/O requests must be for an integral multiple of 512 bytes,
start on a sector boundary
and not go off the end of the disk.

FILES

/dev/xdx0[a-h]	block files
/dev/rxdx0[a-h]	raw files

DIAGNOSTICS

The following error may be returned.

[ENXIO]
Nonexistent or not configured drive (on open, read or write).

SEE ALSO

xt(4).

NAME

XT — XT Multibus adapter 9–track tape driver

DESCRIPTION

The **xt** interface provides access to Multibus adapter 9–track tape drives.

FILES

/dev/xtx0	block files with rewind on close (minor device 0)
/dev/rxtx0	raw files with rewind on close (minor device 0)
/dev/nxtx0	block files with no rewind on close (minor device 8)
/dev/nrxtx0	raw files with no rewind on close (minor device 8)

DIAGNOSTICS

The following errors may be returned.

[EBUSY]
Drive not ready (on a read or write).

[ENXIO]
Nonexistent drive or not configured drive (on open, read or write) or attempt to read past end–of–file on block tape

[EIO]
Drive not online (open), no write ring (open for writing) or forward/backspace error (ioctl).

[EINVAL]
invalid mtio command (ioctl).

SEE ALSO

ioctl(2), *mtio*(4), *xd*(4).

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: name1, name2, name3, . .

The colon separates that which is aliased *(name)* from its aliases *(name1,name2,name3,...).* As shown above, comma–separated arguments can appear in the file. Lines beginning with spaces or tabs are continuation lines. Lines beginning with '#' are comments.

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.

Arguments take four forms: *loginname, loginname@host-id, /filename,* and **:include:** */filename. Loginname* is the login name of the recipient on the local machine. *Loginname@host-id* is the login name and the network name of the recipient's home machine. If the */filename* form is used, any mail sent to the name being aliased is also appended to the named file. If the file does not exist, it is created. If **:include:** */filename* is used, */filename* is the only argument allowed. Recursive definitions cause infinite recursion in *sendmail.* The name being aliased should never be the recipient of the file since it's a dummy name. Input is taken from the specified file until it ends. Processing of the current file continues.

This is an ASCII file used to modify the aliases database; the actual aliasing information is placed into a binary format in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag* using the command **newaliases.** These are *dbm(3d)* files.

Newaliases is automatically run on the first attempt to send mail after */usr/lib/aliases* is changed and then the change takes effect. To execute a **newaliases** command, enter:

In /usr/lib/sendmail /etc/newaliases

or

/usr/lib/sendmail –bi

The person who maintains the list of aliases is known as the owner. To establish an owner, enter:

owner-xxxx: yyyy

yyyy is the owner of the list and *xxxx* is the name of the list. If an error

occurs, the owner receives an error message. If there is no owner and an error occurs, the person sending the mail receives an error message.

EXAMPLES

An example of a simple alias is:

root: joe,sam,jane@central

Any mail addressed to root does not go to root, but rather to joe, sam, and jane.

The recipient can also be a file, for example,

bug-list: /usr/adm/bugsave

Mail is written to the file.

You can also read aliases in from a file, for example,

sys-list: :include:/usr/adm/systemusers

Lines in the file are similar to lines in */usr/lib/alias*. Including aliases in a file is done, for example, when the system administrator owns the mail list of aliases, but the group list is owned by someone else.

If an error occurs on sending mail to a specified list, only the owner of the list is notified of the error. In the example below, eric is the owner and vax-advice is the name of the list. Only eric receives the error message.

owner-vax-advice: eric
vax-advice: eric,jill,sam

Aliasing occurs only on local names. The following example is **not** valid.

john@ucbvax: bill

Duplicates cannot occur, since no messages are sent to any person more than once. For example, given the aliases

sys-issues: sam,robert
sys-bugs: sys-issues,sam,fred

"sam", "fred", and "robert" each receive one copy of mail for sys-bugs.

CAVEATS

Because of restrictions in *dbm*(3d) 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.

SEE ALSO

newaliases(1), dbm(3d), sendmail(8mh).

NAME

a.out — assembler and link editor output

SYNOPSIS

#include <a.out.h>

DESCRIPTION

A.out is the output file of the assembler *as(1)* and the link editor *ld(1)* . Both programs make **a.out** executable if there were no errors and no unresolved external references. Layout information as given in the include file for the VAX–11 is:

```
/*
 * Header prepended to each a.out file.
 */
struct exec {
        long    a_magic; /* magic number */
        unsigned a_text;  /* size of text segment */
        unsigned a_data;  /* size of initialized data */
        unsigned a_bss;   /* size of uninitialized data */
        unsigned a_syms;  /* size of symbol table */
        unsigned a_entry; /* entry point */
        unsigned a_trsize;/* size of text relocation */
        unsigned a_drsize;/* size of data relocation */
};

#define OMAGIC   0407     /* old impure format */
#define NMAGIC   0410     /* read-only text */
#define ZMAGIC   0413     /* demand load format */

/*
 * 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)!=NMAGIC && \
    ((x).a_magic)!=ZMAGIC)

#define N_TXTOFF(x) \
        ((x).a_magic==ZMAGIC ? 1024 : sizeof (struct exec))
#define N_SYMOFF(x) \
        (N_TXTOFF(x) + (x).a_text+(x).a_data + (x).a_trsize+(x).a_drs
#define N_STROFF(x) \
        (N_SYMOFF(x) + (x).a_syms)
```

The file has five sections: a header, the program text and data, relocation information, a symbol table and a string table (in that order). The last three may be omitted if the program was loaded with the '—s' option of **ld** or if the symbols and relocation have been removed by *strip(1)*.

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

When an **a.out** file is executed, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at 0 in the core image; the header is not loaded. If the magic number in the header is OMAGIC (0407), it indicates that the text segment is not to be write–protected and shared, so the data segment is immediately contiguous with the text segment. This is the oldest kind of executable program and is rarely used. If the magic number is NMAGIC (0410) or ZMAGIC (0413), the data segment begins at the first 0 mod 1024 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 1024 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 1024 bytes, and the pages of the file will be brought into the running image as needed, and not pre–loaded as with the other formats. This is especially suitable for very large programs and is the default format produced by *ld(1)* .

The stack will occupy the highest possible locations in the core image: growing downwards from 0x7ffff000. The stack is automatically extended as required. The data segment is only extended as requested by *brk(2)*.

After the header in the file follow the text, data, text relocation data relocation, symbol table and string table in that order. The text begins at the byte 1024 in the file for ZMAGIC format or just after the header for the other formats. The N_TXTOFF macro 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 layout of a symbol table entry and the principal flag values 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; see below */
```

```
        char            n_other;
        short           n_desc;   /* see <stab.h> */
        unsigned        n_value;  /* value of this symbol (or 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 se
 * These are given in <stab.h>
 */
#define N_STAB          0xe0      /* if any of these bits set, */
                                  /*   don't discard */

/*
 * Format for namelist values.
 */
#define N_FORMAT        "%08x"
```

In the **a.out** file a symbol's *n_un.n_strx* field gives an index into the string table. A *n_strx* value of 0 indicates that no name is associated with a particular symbol table entry. The field *n_un.n_name* can be used to refer to the symbol name only if the program sets this up using *n_strx* and appropriate data from the string table.

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

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

If relocation information is present, it amounts to eight bytes per relocatable datum as in the following structure:

```
/*
 * Format of a relocation datum.
 */
struct relocation_info {
        int       r_address;        /* address which is relocated */
        unsigned  r_symbolnum:24,   /* local symbol ordinal */
                  r_pcrel:1,        /* was relocated pc relative already */
                  r_length:2,       /* 0=byte, 1=word, 2=long */
                  r_extern:1,       /* does not include value of sym */
                                    /*   referenced */
                  :4;               /* nothing, yet */
};
```

There is no relocation information if *a_trsize* + *a_drsize* = = 0. If *r_extern* is 0, then *r_symbolnum* is actually a *n_type* for the relocation (i.e. *N_TEXT* meaning relative to segment text origin).

SEE ALSO

adb(1), as(1), ld(1), nm(1), strip(1), stab(5).

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:

```
/*
 * AR.H - ASCII archive header definitions.
 *
 * Modifications from Berkeley 4.2 BSD
 * Copyright (c) 1983, Tektronix Inc.
 * All Rights Reserved
 *
 */

/*
 * Note that the header format has changed. It is no longer fixed-form
 * See the manual page for ar(5) for
 * information on the new archive format.
 */

#include ⟨sys/max.h⟩
#define ARMAG     "!⟨arch⟩\n"/* Short format magic number.*/
#define LARMAG    "!⟨ARCH⟩\n"/* Long format magic number.*/
#define SARMAG    8

#define ARFMAG    "`\n"

struct ar_hdr {
        char      ar_name[MAXNAMLEN + 1];
        char      ar_date[12];
        char      ar_uid[6];
        char      ar_gid[6];
        char      ar_mode[8];
        char      ar_size[10];
        char      ar_fmag[2];
};

/*
 * sar_hdr is the short archive header format (name ⟨= 16 chars)
 * SARNAMLEN is the length of the name field for the short archive
 * header.
 */
```

```
#define SARNAMLEN 16

struct sar_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];
};
```

In this system, the name may be either a blank–padded string of up to 15 characters stored in a 16 character field, or a string of up to MAXNAMLEN characters surrounded by slashes. If the name has an odd number of characters, an extra slash is added in order to keep the size of the header even.

The command **ar** will keep the archive in short format (where the magic string is "!<arch>") unless the name of any one of the archive member names is longer than 15 characters. When this happens, the magic string is changed to "!<ARCH>".

In order to make it easier to modify programs which use the old archive format, the old format structure is provided.

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.

CAVEATS

File names with less than 15 characters lose trailing blanks.

Programs that work with this archive format will not work with the old archive files. See *oldar(5)* for a description of the old archive format, and use *arcv(1)* to convert old archive files to the new format.

Since the archive headers are of variable size, it is no longer possible to read in the archive header using *read(2)* or *fread(3s)*. The subroutines *fgetarhdr(3c)* and *getarhdr(3c)* are supplied for the purpose of reading archive headers.

Long format archives are not portable to other versions of UNIX. See the manual page for *ar(1)* for a way to convert long format archives to short format, which is portable.

SEE ALSO

ar(1), arcv(1), ld(1), make(1), nm(1), ranlib(1), fgetarhdr(3c), getarhdr(3c), oldar(5).

NAME

assign.classes — defined classes of devices for assignment

SYNOPSIS

/etc/assign.classes

DESCRIPTION

This file defines the grouping of devices for assignment by the *assign(1)* command. Devices are grouped into classes in order that all devices corresponding to an actual physical device, for example, raw and cooked versions, are assigned as a unit. Assignment changes the ownership of the devices.

Each line defines one class and is of the form

```
class ... = device ...
```

where classes are site defined names that correspond to the grouping of devices on the line, and devices are full path names, usually of devices in */dev*. The equal sign separates class names from device names. All names are separated by spaces or tabs. Class names that are listed in more than one line are termed generic class names, and will be checked by **assign** in the order listed in this file.

EXAMPLES

The following defines classes *mt3* and *mt4*, for mag tape devices 3 and 4, with a generic class name *tape*.

```
mt3 tape = /dev/rmt03 /dev/nrmt03 /dev/nmt03 /dev/mt03
mt4 tape = /dev/rmt04 /dev/nrmt04 /dev/nmt04 /dev/mt04
```

SEE ALSO

assign(1), sysadmin(8).

NAME

bom — Bill of Materials specification file format for use with comply

DESCRIPTION

Comply(8) specification file contains for each file to be checked for compliance the following information:

pathname (full or relative to *comply(8)* invocation)
owner
group
mode
size
hard link count
rcsid number
checksum
symlink target

This is an ASCII file. Each field within each specification entry is separated from the next by a tab. Each specification is separated from the next by a new-line. Any line beginning with a **%** will be considered a comment. If the first line in the file begins with a comment, that comment will be used as a verbose description of the comply specification file when *comply(8)* runs. All blank lines are ignored.

If any of the above described fields are empty (bracketed by tabs), then comply will not bother to check compliance to those fields. This is true for all fields but *pathname, mode,* and *symbolic link target* when the file type is symbolic link fields.

FIELDS

pathname
: This can be a absolute path (i.e. starts with /), or it can be a relative path from the current working directory of the *comply(8)* invocation.

owner
: Login name of the owner of the file/directory.

group
: Symbolic name of the group owner of the file/directory.

mode
: This field contains both the file type and the file mode. The format for this is as in *ls*(1). The first character specifies the file type. Directories, block special, character special, symbolically linked, and regular files are denoted by **d**, **b**, **c**, **l**, and **-**, respectively. The remaining nine characters specify the mode. The read (**r**), write (**w**), or execute (**x**) permissions are in the order for owner, group, and others. Instead of **x** for the owner (or group), **s** designates a setuid (setgid) program. Similarly, **t** instead of **x** in the other mode designates a program with the *sticky*(8) bit on.

size
: The size in bytes on the disk, this does **not** represent the size of the file/program in a running state in core. **Or** , if the file is a device, this field should be the major/minor device numbers comma or space separated.

hard link count
Number of hard links to this file.

rcsid **number** This is the RCS revision number associated with the file (e.g. 1.35).

checksum
Checksum of the file as provided by the *sum(1)* command.

symlink target
This is the target file/directory name if this file is a symbolic link. This has the same form as **pathname .**

EXAMPLES

An example of a **bom** file follows where '@' is used to signify a tab.

/etc/catman@sys@sys@-rwxr-xr-x@16384@1@1.4@65389@
/etc/chown@root@sys@-rwxr-xr-x@10360@1@1.17@31625@
/bin/ll@sys@sys@lrwxr-xr-x@16384@1@3.2@52346@/bin/ls

SEE ALSO

comply(8), ident(1RCS), ls(1), rcs(1RCS), sticky(8), sum(1).

NAME

bootsrv.conf — boot server configuration file

DESCRIPTION

The file */etc/bootsrv.conf* contains the host–specific and default download file specifications used by the boot server (see */etc/bootsrvd(8n)*). Comment lines begin with "#". The format of each non–comment file entry is:

Hostaddr *ID expr. load file #comment (optional)*

"*Hostaddr*" is the requesting node's Internet address, or "*" for a default file specification.
"*ID expr.*" is a regular expression (<20 characters) used to match the requesting node's ID.
"*load file*" is the full path name (<256 characters) of the download file.
All fields are separated by spaces or tabs.

A host–specific download file is associated with the Internet address given in the first field of the line. When a boot request is received from a remote station at that Internet address, the specified load file is then transferred to the remote station.

If the boot server cannot find a host–specific download file for a remote station, then the list of default file specifications is searched. For each default file specified, the ID regular expression is compared with the ID string in the boot request. When a match is found, the associated default file is transferred to the remote station.

A default download file is specified with "*" in the *hostaddr* field, and a valid regular expression in the *ID expr.* field. For default load files, the ID expression must not be "*", but must include the machine family (e.g., "*6100*", "*61.**"). Default file specifications should be ordered from most–specific to least–specific.

FILES

/usr/lib/bootsrv	This is the directory in which the boot server builds its database files. The "*boot_conf*" file in this directory is a working copy of the information contained in "*/etc/bootsrv.conf*".

EXAMPLES

The following entry will associate the download file "*/u/joeuser/myloadfile*" with the remote station at 201.123.234.255.

```
201.123.234.255 *       /u/joeuser/myloadfile
```

The follow default file specification will cause the file "*/diags/diags_os.lan*" to be transferred to 6130 System workstations.

```
*                       61.*    /diags/diags_os.lan
```

CAVEATS

If the configuration file is moved or deleted, the boot server will die. To change the information in the configuration file, edit a temporary copy of the information, then either copy or move this file into the configuration file.

SEE ALSO

bootsrvd(8n).

NAME

chfn — definitions for password file gecos field contents

SYNOPSIS

/usr/lib/chfn

DESCRIPTION

The **chfn** file is a collection of regular expressions that are used by the *chfn(1)* utility to check for legal input while updating the password file gecos field. They specify the order for the entries and restrict input for each section of the field.

The this file is used one line at a time by *chfn(1)* first to produce the prompt string for the entry and then to check the user's input. Line format is:

```
[!][prompt_string];[regular_expr [ ¦ regular_expr ]...]
```

The *prompt_string* is the prompt to be issued for this portion of the gecos field. Example entries should be included here. The current content of the sub-field is printed following this prompt string.

If the first character of the prompt string is "!" that sub-field can be changed only by a person logged in as root. It will be skipped in *chfn(1)* calls for any other user. If restricted sub-fields exist, placing them at the end of the gecos field will enhance execution speed.

There is no "and" condition and if the beginning of the line and the end of the line are significant they must be included in the pattern. If no pattern exists any input will be accepted.

The "or" construct has beed added to enhance the restricted environment of regular expressions. If "¦" is used the patterns specified are checked in order. The first match stops the scan.

EXAMPLES

The following is an example file. It promts for a name and accepts any input; then asks for a work phone and checks for a number of the form XXX-XXXX; delivery station is requested next and a number for mail delivery is requested; the user is then asked for a home phone number and its form checked; finally, if the user is super user the home machine entry is requested. Note that all patterns include start of line and end of line indications.

```
Name;
Work Phone (Example: 555-1212);^[1-9][0-9][0-9]-[0-9][0-9][0-9][0-9]
Delivery Station (Example: 77-215);^[Y0-9][0-9]-[0-9][0-9][0-9]$
Home Phone;^[1-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
!Home Machine (Example: tekecs);^tektronix$¦^tekecs$¦^gumby$
```

FILES

/usr/lib/chfn

CAVEATS

If you are reading this page prior to the manual page for *chfn(1)* it won't make much sense.

Regular expressions must be legal or changes will not be accepted. No attempt is made to check the existing password file entries for correctness, e.g. changes made with *vipw(8)* are not verified.

This file allows local modification of the restrictions placed on users for access to the password file, it helps avoid mistakes. It does not stop the super user from making changes outside the *chfn(1)* utility.

If changes are made to existing *passwd(5)* sequences the prompts will be unrelated to the existing values until all users have updated their information. Additions to the end are the easiest to make. Changes to the information stored here also affects *finger(1n)* output and must be coordinated with the expected information there.

SEE ALSO

chfn(1), finger(1n), passwd(5), vipw(8).

NAME

core — format of memory image file

SYNOPSIS

#include <machine/param.h>

DESCRIPTION

The UTek 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 core file consists of the *u.* area, whose size (in pages) is defined by the UPAGES manifest in the *<machine/param.h>* file. The *u.* area starts with a **user** structure as given in *<sys/user.h>*. The remainder of the core file consists first of the data pages and then the stack pages of the process image. The amount of data space image in the core file is given (in pages) by the variable *u_dsize* in the *u.* area. The amount of stack image in the core file is given (in pages) by the variable *u_ssize* in the *u.* area.

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

SEE ALSO

adb(1), setrlimit(2), sigvec(2).

NAME

cpio — format of cpio archive

DESCRIPTION

The *header* structure, when the **—c** option of *cpio(1)* is not used, is:

```
struct {
        short   h_magic,
                h_dev;
        ushort  h_ino,
                h_mode,
                h_uid,
                h_gid;
        short   h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the **—c** option is used, the *header* information is described by:

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11o%6o%11o%s",
        &Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
        &Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
        &Longtime, &Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file are recorded in an element of the array of varying length structures, **archive**, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat(2)*. The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the **archive** always contains the name *TRAILER!!!*. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

CAVEATS

On some systems, the value of *Hdr.h_namesize* must be less than 128. On this system, the value is restricted only by the maximum length of a pathname (currently 1024). Care must be taken if the **cpio** archives are to be used on other systems.

SEE ALSO

ar(1), cpio(1), find(1), tar(1), stat(2), ar(5), tar(5).

NAME

daemontab — daemon process description file

DESCRIPTION

The file */etc/daemontab* is used by the command **/etc/daemon** to find the names of **daemon** programs, such as spoolers and network communication programs.

The **daemontab** file may contain lines of three types. The first type is a comment, which is any line that begins with the character '#'. These lines are ignored by **daemon.**

The next type of line is read directly by **daemon** and is of the form:

[**—K***sig*] [**—Y***sig*] [**—w***time*] *path* [*args*]

The **—K** option is used to specify the default signal to be used to kill the program. The signal may either be a number or a word as listed by the command **kill –l**. The **—Y** option is used to specify the default signal to be used to synchronize the program. The signal may either be a number or a word as listed by the command **kill –l**. The **—w** option is used to specify the default time to wait after attempting to kill a process before checking to see whether the process was really killed or not. The *path* is the full pathname of the program, and the *args* are the arguments to the command. For example, the line

```
-YHUP -K2 -w10 /etc/foo_daemon -l -t15
```

specifies that **/etc/foo_daemon** is a daemon (at least when run with the options **–l** and **–t15**) and is to be killed with the signal 2 (or interrupt). A successful kill will take up to 10 seconds to be reflected by the system; the program is to be synchronized with the signal HUP (or signal 1), and if it is dead 10 seconds after the synchonization is attempted, there is something wrong. It is important to note that the lines:

```
-K3 /etc/foo_daemon
-YHUP -w5 /etc/foo_daemon -l
-K2 -w10 /etc/foo_daemon -t15
```

all differ, both from one another and from the first example. The command and its arguments *together* specify a distinct program invocation. Spaces and tabs are not significant, and are reduced to a single space.

The third type of line in the **daemontab** file begins and ends with the backquote character (‘). This type of line is executed by **daemon** via *popen(3s),* and the output from the execution is taken as a list of program names as if they were listed in the **daemontab** file. This feature makes it

possible to start different **daemons** depending on the state of the system. For example, your system may have a package which requires one **daemon** to be running if one peripheral is hooked up, and a different **daemon** otherwise. The line:

```
`/usr/pkg/which_daemon`
```

would cause **daemon** to execute the command **/usr/pkg/which_daemon** and use the output as the name of the **daemon** that should be running. This is similar to executing the command

```
/etc/daemon [ options ] `/usr/pkg/which_daemon`
```

since the shell performs the same action with backquoted commands (watch out for shells which turn newlines into spaces).

FILES

/etc/daemontab The daemon process description file.

CAVEATS

It is very important to realize that any user can execute the command *daemon,* even if only the superuser can actually perform any actions on the programs. Therefore, you must be very careful to make sure that the commands which appear in backquotes do not do anything other than print the names of daemons.

There is no way to specify the default signal or wait time with this type of line. Otherwise, the lines produced are interpreted the same as with program names directly specified in the daemontab file.

All lines in */etc/daemontab* are limited to 1024 characters in length. Also, output from backquoted commands is limited to 1024 characters.

SEE ALSO

sh(1sh), kill(1), ps(1), popen(3s), daemon(8).

NAME

devdes — Device Description file for system configuration

DESCRIPTION

A device description file contains all information about a device needed for system configuration, *sysconf(8)*. The file should be named by signature name suffixed with *.d*. A # indicates the rest of the line is a comment. *$* separates entries if multiple devices are described in a file.

A device description file consists of six static fields followed by a keywords indicating further information. The six static fields are:

the signature name,

the device driver name,

signature id (2 hex digits),

a flag word,

the name of the attach routine

a description of the device in quotes.

The possible settings of the flag word are:

REQ_DEV	1	Device is required
NO_DISPLAY	2	Don't display
CONTRLR	4	Device is a controller
CONTROLLED	8	Device is a controlled device

The possible keywords are:

Character	following fields are Character Switch table entries.
Block	following fields are entries for Block switch table.
Iotab	following fields are entries for the 6200 I/O table. Iotab is not valid for a 6100 workstation.
Makedev	following lines are a shell script to be used in making special device files. Lines terminate when a keyword or the end of file is reached.
Controls	list of devices supported by controller.
Define	integer variable which needs to be defined for driver useage. Two values are required; the first is used when driver is not active; the second when the driver is active.
Alias	this device is considered an alias for device named in the next field. The same device driver supports both devices.

FILES

/usr/sys/conf/descrip	Directory for device descriptions for system configuration

SEE ALSO

sysconf(8).

NAME

dir — format of directories

SYNOPSIS

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

DESCRIPTION

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

```
/*
 * 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) &~

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];
};
```

The library routines for manipulating directories use the structure _dirdesc for storing directory pointers (as with FILE in the standard I/O library). The library routines are *closedir(3c), opendir(3c), readdir(3c),* scandir(3c), *seekdir(3c),* and *telldir(3c).*

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), closedir(3c), opendir(3c), readdir(3c), scandir(3c), seekdir(3c), telldir(3c).

NAME

diskpart — results of IDSKIOCIDDRV rigid disk ioctl

SYNOPSIS

#include ⟨sys/ioctl.h⟩
#include ⟨machine/idiskioctl.h⟩

ioctl(fd, IDSKIOCIDDRV, argp)
int fd;
struct id_drvstat *argp;

DESCRIPTION

The IDDRV command is used to obtain information about a rigid disk partition (this does not apply to floppy disks). The file descripter must be opened on any raw device associated with the disk of interest. For example, information about partition 2 (normally swap space) on the internal disk of a 6130 would be obtained by opening **/dev/rdw00[a-p]** and calling *ioctl* with the **IDSKIOCIDDRV** command.

The structure **id_drvstat** can be found in the include file ⟨**machine/idiskioctl.h**⟩ and is shown below:

```
/*
 * Copyright (C) 1983, Tektronix Inc. - All rights Reserved
 *
 */

/*
 * struct id_drvstat - status of a drive -- the result of an
 *  IDDRV command. These structures should match those defined
 *  in the maintenance block structure.
 */

#define IDPARTCNT       16              /* partitions/drive */
struct  id_volume_id {
        char    idvi_disk_id[8];        /* 8 bytes of ID */
        char    idvi_diskname[8];       /* disk type name */
};
struct  id_phys_desc {
        unsigned short  idpy_ncyl;      /* nbr of cylinders */
        unsigned char   idpy_nhead;     /* number of heads */
        unsigned char   idpy_nsect;     /* USED sectors/track */
        unsigned char   idpy_nspare;    /* SPARE sectors/track */
        unsigned char   idpy_bps;       /* coded data sector size */
        unsigned short  idpy_rpm;       /* rotational speed */
};
struct id_partinfo {
        unsigned short  idpi_ncyls;     /* size (in cylinders) */
        unsigned short  idpi_cylofst;   /* starting cylinder number */
        char            idpi_type[2];   /* 2 char type (e.g. "US") */
```

```
        char            idpi_val[2];    /* 2 byte type-specific data */
};
#define idpi_blksiz     idpi_val[0]     /* coded fs blksize for "UF" */
#define idpi_frgsiz     idpi_val[1]     /* coded fs fragsize for "UF" *

struct id_drvstat {
        struct id_volume_id idds_vid;
        struct id_phys_desc idds_phys;
        struct id_partinfo  idds_part[IDPARTCNT];
};

/*
 * values for id_phys_desc.idpy_bps
 */

#define IDMB_BPS_128    0
#define IDMB_BPS_256    1
#define IDMB_BPS_512    2
#define IDMB_BPS_1024   3

/*
 * values for id_partinfo.idpi_blksiz and id_partinfo.idpi_frgsiz
 * powers of 2, representing upper and lower limits
 */

#define IDMB_BLKL       ((char)12)
#define IDMB_BLKH       ((char)13)
#define IDMB_FRGL       ((char)9)
#define IDMB_FRGH       ((char)12)

/*
 * convert encoded value of id_partinfo.idpi_blksiz to integer
 */

#define idmb_blktosiz(code)     (1 << (int)(code))

/*
 * convert encoded value of id_partinfo.idpi_frgsiz to integer
 */

#define idmb_frgtosiz(code)     (1 << (int)(code))
```

The following is a summary of the possible values of **idpi_type**:

UF UTek file system partition.

US UTek paging space.

XX Unused partition.

RW Read/Write, a misnomer, used exclusively for the data space partition.

DG Diagnostic partition.

MP Maintenance partition.

WD Whole disk partition.

EXAMPLES

This example program estimates how many page pairs can be held by the standard swap space on a 6130.

```
#define    stratos
#include <stdio.h>
#include <sys/ioctl.h>
#include <machine/idiskioctl.h>

#define PAIRSIZE        1024

static  struct  id_drvstat      info_buffer;

main()
{
        register int    fd;
        register struct id_drvstat *argp;
        register int    bytes;

        *argp = &info_buffer;
        fd = open("/dev/rdw00b", 0);
        ioctl(fd, IDSKIOCIDDRV, argp);
        if (strncmp(argp->idds_part[1].idpi_type, "US", 2)
                != 0) {
                fprintf(stderr, "failed consistency test.");
                exit(1);
        }
        switch (argp->idds_phys.idpy_bps) {
        case IDMB_BPS_128:   bytes = 128;   break;
        case IDMB_BPS_256:   bytes = 256;   break;
        case IDMB_BPS_512:   bytes = 512;   break;
        case IDMB_BPS_1024:  bytes = 1024;  break;
        }
        bytes *= argp->idds_phys.idpy_nsect;
        bytes *= argp->idds_phys.idpy_nhead;
        bytes *= argp->idds_part[1].idpi_ncyls;
        printf("swap space will hold %d page pairs\n",
                bytes / PAIRSIZE);
}
```

SEE ALSO

ioctl(2), dh(4), dw(4).

NAME

errtag — file format of the errtag files

DESCRIPTION

The *msghlp(1)* command provides help about error messages. It reads the files in the */usr/lib/errtags* directory for this information. The format of the help files in this directory is as follows:

```
* comment
* comment
-str1
text
-str2
text
* comment
text
-str3
text
```

The *str?* that matches the key is found and the following text lines are printed. Comments are ignored.

FILES

/usr/lib/errtags	directory containing files of message text.
/usr/lib/errtags/<non-numeric prefix>	the file searched if the message is not found in */usr/lib/errtags/helploc.*
/usr/lib/errtags/default	the file searched if the argument is all numeric.
/usr/lib/errtags/helploc	the file searched if the argument has a non-numeric prefix.
/usr/lib/errtags/cmds	contains the syntax lines for the commands.

SEE ALSO

msghlp(1).

NAME

forms — MDQS valid forms file

DESCRIPTION

The file */usr/lib/mdqs/forms* is used by **MDQS** commands to find the names of valid forms that can be associated with devices or requests.

The **forms** file may contain any number of lines of the form:

⟨name⟩ [⟨separator⟩ ⟨alias⟩ . . .]

The **name** is the name of a valid **form** available to any **MDQS** command that needs to specify a **form.** The **seperator** is any white space or a coma. The **alais** is any other name that can be used in place of the original **name.**

CAVEATS

If the **forms** file does not exist any form name is considered valid. All lines in */usr/lib/mdqs/forms* are limited to 132 characters in length.

FILES

/usr/lib/mdqs/forms The MDQS valid forms file.

SEE ALSO

lpr(1mdqsd), qmod(1mdqs), qconf(5mdqs), qdev(8mdqs).

NAME

fs, inode — format of file system volume

SYNOPSIS

#include ⟨sys/types.h⟩
#include ⟨sys/fs.h⟩
#include ⟨sys/inode.h⟩
#include ⟨sys/param.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. Sectors are 512 bytes in length.

The actual file system begins at sector 16 with the *super block*. The layout of the super block as defined by the include file ⟨*sys/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 */
char    fs_fmod;                /* super block modified flag */
char    fs_clean;               /* file system is clean flag */
char    fs_ronly;               /* mounted read-only flag */
char    fs_flags;               /* currently unused flag */
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 (defined in ⟨sys/param.h⟩) can be optionally broken into 2, 4, or 8 pieces, each of which is addressable; these pieces may be DEV_BSIZE (defined in ⟨sys/dir.h⟩), 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 defined in ⟨sys/fs.h⟩.

The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined.

The root inode, inode 2, is the root of the file system. (Inode 0 can't be used for normal purposes and inode 1 was once used for linking bad blocks, so inode 2 is used for the root inode.) 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 super-user 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 (defined in ⟨sys/fs.h⟩) 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) (defined in ⟨sys/fs.h⟩).

MINBSIZE (defined in ⟨sys/fs.h⟩) 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 (defined in ⟨sys/fs.h⟩) 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 (defined in ⟨sys/fs.h⟩).

The path name on which the file system is mounted is maintained in *fs_fsmnt*. MAXMNTLEN (defined in ⟨sys/fs.h⟩) 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 (defined in ⟨sys/fs.h⟩). 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 (defined in ⟨sys/fs.h⟩) 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 (defined in ⟨sys/fs.h⟩) 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 UTek 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 ⟨*sys/inode.h*⟩.

SEE ALSO

fstab(5).

NAME

fstab — static information about the filesystems

SYNOPSIS

#include <fstab.h>

DESCRIPTION

The file */etc/fstab* contains descriptive information about the various file systems. */etc/fstab* is only *read* by programs, and not written; it is the duty of the system administrator to properly create and maintain this file. The order of records in */etc/fstab* is important because **fsck, mount,** and **umount** sequentially iterate through */etc/fstab* while executing.

The fstab structure is defined in *fstab.h* as:

```
struct fstab {
        char *fs_spec; /* block special device name */
        char *fs_file; /* file system path prefix */
        char *fs_type; /* rw,ro,sw or xx */
        int  fs_freq;  /* dump frequency, in days */
        int  fs_passno;/* pass number on parallel dump */
};
```

The special file name is the **block** special file name, and not the character special file name. If a program needs the character special file name, the program must create it by appending a "r" after the last "/" in the special file name.

Fs_type may be one of the following, from *fstab.h:*

```
#define FSTAB_RW        "rw"    /* read-write device */
#define FSTAB_RO        "ro"    /* read-only device */
#define FSTAB_RQ        "rq"    /* read-write with quotas */
#define FSTAB_SW        "sw"    /* swap device */
#define FSTAB_XX        "xx"    /* ignore totally */
```

If *fs_type* is "rw" or "ro" then the file system whose name is given in the *fs_file* field is normally mounted read-write or read-only on the specified special file. If *fs_type* is "rq", then the file system is normally mounted read-write with disk quotas enabled. The *fs_freq* field is used for these file systems by the *dump(8)* command to determine which file systems need to be dumped. The *fs_passno* field is used by the *fsck(8)* program to determine the order in which file system checks are done at reboot time. The root file system should be specified with a *fs_passno* of 1, and other file systems should have larger numbers. File systems within a drive should have distinct numbers, but file systems on different drives can be checked on the same pass to utilize parallelism available in the hardware.

If *fs_type* is "sw" then the special file is made available as a piece of swap space by the *swapon(8)* command at the end of the system reboot procedure. The fields other than *fs_spec* and *fs_type* are not used in this case.

If *fs_type* is "rq" then at boot time the file system is automatically processed by the *quotacheck(8)* command and disk quotas are then enabled with *quotaon(8)*. File system quotas are maintained in a file "quotas", which is located at the root of the associated file system.

If *fs_type* is specified as "xx" the entry is ignored. This is useful to show disk partitions which are currently not used.

The proper way to read records from */etc/fstab* is to use the routines *getfsent, getfsspec, getfstype, and getfsfile.*

FILES

/etc/fstab

SEE ALSO

getfsent(3c).

NAME

getdate — time and date format for MDQS

DESCRIPTION

Getdate converts common time specifications to standard UTek format. The input format is used for time specification with the -a options of mdqs commands batch and lpr. The format is a character string defined as follows:

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

date A *date* is a specific month and day, and possibly a year. Acceptable formats are *mm/dd* [*/yy*] and *monthname dd* [, *yy*] If omitted, the year defaults to the current year; if a year is specified as a number less than 100, 1900 is added.

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

relative time

Specifications relative to the current time are also accepted. The format is [*number*] *unit* [*ago* acceptable units are *year*, *month*, *fortnight*, *week*, *day*, *hour*, *minute*, *second*, *today*, *now*, *this*, *tomorrow*, and *yesterday*.

The actual date is formed as follows:

First, any absolute date and/or time is processed and converted. Using that time as the base, day-of-week specifications are added.

Next, relative specifications are used. If a date or day is specified, and no absolute or relative time is given, midnight is used.

Finally, a correction is applied so that the correct hour of the day is produced after allowing for daylight savings time differences.

Most common abbreviations for days, months, and so forth are allowable. In particular, they may have upper- or lowercase first letters and three-letter abbreviations for any of them, with or without a trailing period, are recognized. Units, such as *week*s, may be specified in the singular or plural. Time zone and meridian values may be in upper- or lowercase, and with or without periods.

EXAMPLES

For the following examples the current time is Jan 16, 1985 11:05 AM.

```
tomorrow 2 PM
```

This date will be Jan 17, 1985 14:00.

```
20 minutes
```

This date will be Jan 16, 1985 13:05.

```
next thu 13:30
```

This date will be Jan 24, 1985 13:30.

```
April 1 5:30 PM
```

This date will be April 1, 1985 17:30.

```
1 hour ago
```

This date will be Jan 16, 1985 10:05.

CAVEATS

The daylight savings time correction can get confused if handed times between midnight and 2:00 am on the days that the reckoning changes.

SEE ALSO

ctime(3c).

NAME

gettytab — terminal configuration data base

SYNOPSIS

/etc/gettytab

DESCRIPTION

Gettytab is a simplified version of the *termcap(5t)* 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.)

Refer to *termcap(5t)* 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
bd	num	0	backspace delay
bk	str	0377	alternate end of line character (input break)
cb	bool	false	useCRTt 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 enviroment
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	line connected to a MICOM port selector
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 enviroment)
ub	bool	false	do unbuffered output (of prompts etc)
uc	bool	false	terminal is known upper case only
un	str	none	default user name to give to login
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 before upon completion are derived from the boolean flags specified. If the derivation should prove inadequate, any (or all) of these three may be overriden 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 **im** and login message **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.

CAVEATS

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.

The delay capability is questionable. Apart form its general lack of flexibility, some of the delay algorithms are not implemented. The terminal driver should support sane delay settings.

Currently *login(1)* sets the environment, so any environment settings done in **gettytab** will be overwritten.

Termcap format is hard to use; something more rational would be an improvement.

SEE ALSO

termcap(5t), getty(8).

NAME

gps — graphical primitive string, format of graphical files

DESCRIPTION

GPS is a format used to store graphical data. Several routines have been developed to edit and display GPS files on various devices. Also, higher level graphics programs such as *plot(1g)* and *vtoc(1g)* produce GPS format output files.

A GPS is composed of five types of graphical data or primitives.

GPS PRIMITIVES

lines The *lines* primitive has a variable number of points from which zero or more connected line segments are produced. The first point given produces a *move* to that location. (A *move* is a relocation of the graphic cursor without drawing.) Successive points produce line segments from the previous point. Parameters are available to set *color*, *weight*, and *style* (see below).

arc The *arc* primitive has a variable number of points to which a curve is fit. The first point produces a *move* to that point. If only two points are included, a line connecting the points will result; if three points a circular arc through the points is drawn; and if more than three, lines connect the points. (In the future, a spline will be fit to the points if they number greater than three.) Parameters are available to set *color, weight,* and *style.*

text The *text* primitive draws characters. It requires a single point which locates the center of the first character to be drawn. Parameters are *color*, *font*, *textsize*, and *textangle.*

hardware The *hardware* primitive draws hardware characters or gives control commands to a hardware device. A single point locates the beginning location of the *hardware* string.

comment A *comment* is an integer string that is included in a GPS file but causes nothing to be displayed. All GPS files begin with a comment of zero length.

GPS PARAMETERS

color *Color* is an integer value set for *arc, lines,* and *text* primitives.

weight *Weight* is an integer value set for *arc* and *lines* primitives to indicate line thickness. The value **0** is narrow weight, **1** is bold, and **2** is medium weight.

style *Style* is an integer value set for *lines* and *arc* primitives to give one of the five different line styles that can be drawn on TEKTRONIX 4010 series storage tubes. They are:

0 solid
1 dotted
2 dot dashed
3 dashed
4 long dashed

font An integer value set for *text* primitives to designate the text font to be used in drawing a character string. (Currently *font* is expressed as a four–bit *weight* value followed by a four–bit *style* value.)

textsize *Textsize* is an integer value used in *text* primitives to express the size of the characters to be drawn. *Textsize* represents the height of characters in absolute *universe-units* and is stored at one–fifth this value in the size–orientation (*so*) word (see below).

textangle *Textangle* is a signed integer value used in *text* primitives to express rotation of the character string around the beginning point. *Textangle* is expressed in degrees from the positive x–axis and can be a positive or negative value. It is stored in the size–orientation (*so*) word as a value 256/360 of it's absolute value.

ORGANIZATION

GPS primitives are organized internally as follows:

lines	*cw points sw*
arc	*cw points sw*
text	*cw point sw so* [*string*]
hardware	*cw point* [*string*]
comment	*cw* [*string*]

cw *Cw* is the control word and begins all primitives. It consists of four bits that contain a primitive–type code and twelve bits that contain the word–count for that primitive.

point(s) *Point(s)* is one or more pairs of integer coordinates. *Text* and *hardware* primitives only require a single *point*. *Point(s)* are values within a Cartesian plane or *universe* having 64K (—32K to +32K) points on each axis.

sw *Sw* is the style–word and is used in *lines, arc,* and *text* primitives. For all three, eight bits contain *color* information. In *arc* and *lines* eight bits are divided as four bits *weight* and four bits *style*. In the *text* primitive eight bits of *sw* contain the *font*.

so *So* is the size–orientation word used in *text* primitives. Eight bits contain text size and eight bits contain text rotation.

string *String* is a null–terminated character string. If the string does not end on a word boundary, an additional null is added to the GPS file to insure word–boundary alignment.

SEE ALSO

abs(1g), af(1g), bar(1g), bel(1g), bucket(1g), ceil(1g), cor(1g), cusum(1g), cvrtopt(1g), dtoc(1g), erase(1g), exp(1g), floor(1g), gamma(1g), gas(1g), gd(1g), ged(1g), graphics(1g), gtop(1g), hardcopy(1g), hilo(1g), hist(1g), hpd(1g), intro(1g), label(1g), list(1g), log(1g), lreg(1g), mean(1g), mod(1g), pair(1g), pd(1g), pie(1g), plot(1g), point(1g), power(1g), prime(1g), prod(1g), ptog(1g), qsort(1g), quit(1g), rand(1g), rank(1g), remcom(1g), root(1g), round(1g), siline(1g), sin(1g), subset(1g), td(1g), tekset(1g), title(1g), total(1g), ttoc(1g), var(1g), vtoc(1g), whatis(1g), and *yoo(1g).*

NAME

group — group file

DESCRIPTION

Group contains for each group the following information:

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

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

This file resides in directory */etc*. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

CAVEATS

The *passwd(1)* command won't change the passwords.

SEE ALSO

passwd(1), setgroups(2), crypt(3c), initgroups(3c), passwd(5).

NAME

hardlink — hard link specification file format for use with hardlink

DESCRIPTION

Hardlink(8) specification file consists of lines where each line contains all the files that are to be hard linked together. The files on a given line may be separated by any amount of white space (i.e. spaces and tabs). Each specification is separated from the next by a new–line.

Any line beginning with a **%** will be considered a comment. If the first line in the file begins with a comment, that comment will be used as a verbose description of the hardlink specification file when *hardlink(8)* runs. All blank lines are ignored.

EXAMPLES

An example of a **hardlink** file follows that would link all the different ways **vi** can be called:

/bin/vi /bin/view /bin/ex /bin/e /bin/edit

SEE ALSO

ln(1), hardlink(8).

NAME

hosts — host name data base

DESCRIPTION

The **hosts** file contains information regarding the known hosts on the local network. Only hosts not running the Tek *nameserver(8n)* are listed. For each host a single line should be present with the following information:

official host name
Internet address
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official host data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional "." notation using the *inet_addr* routine from the Internet address manipulation library, *inet(3n)*. Host names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/hosts

CAVEATS

The *nameserver(8n)* reads this file on startup, so changes will not be reflected until it is restarted.

Only the first entry is used in the case of duplicates.

SEE ALSO

gethostent(3n), nameserver(8n).

NAME

hosts.dfs.access — control remote access to local files.

DESCRIPTION

The file **hosts.dfs.access** determines which users on which remote hosts may access the local file system. This file is read by the distributed file system daemon, *dfsd(8n)* when it starts up and also whenever the file is changed. An example **hosts.dfs.access** file follows:

```
# allow access by the following users:
host1        clarence
host2        doug
host3        allen
host3        steve
```

Note that # is a comment character.

Remember that **hosts.dfs.access** determines if the daemon will attempt to execute the system call on behalf of the requesting host. However, the standard UTek owner–group–other protection scheme will ultimately determine the accessibility of the file by the remote process.

It is recommended that for editing **hosts.dfs.access** you use *vidfs(8n)*. It will make a number of checks on the validity of the entries in the access file. When setting up **hosts.dfs.access** keep two points in mind. First, users mentioned in **hosts.dfs.access** must already be in the password file (*/etc/passwd*). They do not, however, have to have login privileges nor do they need a home directory. You can prevent them from logging on by setting their encrypted password to '*' or any other single letter (because no password encrypts to a single character). Secondly, the local userid assigned to the remote user wishing to access the local file system must match the userid assigned to that remote user on the remote host.

The file is read only when *dfsd* notices that it has changed. The daemon checks to see if the file has changed at most once a minute.

From version 2.0 of */etc/dfsd*, it is possible to specify an 'alias' in **hosts.dfs.access**, for example:

```
host1        root
host1        leon    root
```

The first entry allows access by *root* from *host1*. However for security reasons allowing access by *root* is not often appropriate. The second entry allows access by *root* from *host1*, but the local system treats the request as if it came from *leon* and so access is based on *leon*'s rights, not those of *root*. Note that if two users have the same alias, only one of those aliases will take effect. In other words it is not possible for multiple users to have an entry with the same alias because when a request is received from that alias, there is no way of knowing whose access rights to use. This issue is resolved by the dfs daemon by assigning rights

based on the first entry it finds in its internal tables (which are based on **hosts.dfs.access**).

From version 3.0 of */etc/dfsd*, it is possible to specify only an host name in **hosts.dfs.access**, for example:

```
host1           # allow all accesses except root
host1   root    # also allow root access
```

Specifying only a hostname allows access by all users on that host except root. Remember that even when specifying only a hostname, each user from that remote host that wants to access the local system must be in the local passwd file.

FILES

/etc/hosts.dfs.access	DFS access database file.

SEE ALSO

dfsd(8n), vidfs(8n).

NAME

hosts.equiv, .rhosts — control remote access for rsh, rcp, rlogin and rcmd.

DESCRIPTION

The files */etc/hosts.equiv* and *.rhosts* determine which users on which hosts may access the local file system. System utilities like *rlogin(1n), rsh(1n), and rcp(1n)* use these files.

Access is based on user login names. Therefore it is important from a security standpoint that all hosts allowed access to the local file system are under the same administration (or at least cooperating closely) to prevent accidently assigning the same user login name to two different individuals.

The file */etc/hosts.equiv* is meant to be used by system administrators to govern which other hosts are allowed access to the local file system. Typically only hostnames are specified in this file. Any remote access attempt from any remote user (except root) on a host named in */etc/hosts.equiv* will be permitted assuming 1) the remote user has an account on the local machine, and 2) the permissions for the local account allow accessing the target file. It is also possible to limit access to a particular user on a particular host by specifying the username after the hostname. An example *hosts.equiv* file follows:

```
host1
host2
```

The file *.rhosts* is meant to be used by individual users to allow access from their accounts on other remote hosts or to allow access by other remote users to the local user's account. The file must be located in the user's home directory. An example *.rhosts* file follows:

```
host1 peter
host2 root
```

In other words, remote accesses from *peter* on *host1* will be allowed. If this example .rhosts file appeared in user *scarlett*'s home directory, then requests from *peter* on *host1* will execute on the local system as if submitted by *scarlett*. The format of *.rhosts* is the same as for the *hosts.equiv* file but there is a slight change in interpretation. If a hostname is listed but there is no username accompanying it, then access will only be permitted for a user on the remote host with the same user login name as the account on the local host in which the *.rhosts* file is located. In other words, listing only a hostname in a .rhosts file does not allow access by every user on that remote host.

The file is read each time access is attempted, so as soon as the file is modified, the latest version of that file will be in effect.

FILES

/etc/hosts.equiv

.rhosts

CAVEATS

Other non–Tektronix systems may be far stricter about the format of the *.rhost* and *hosts.equiv* files, i.e. a hostname must begin in the first column, and the delimiter between the hostname and username must be a single space.

SEE ALSO

rcp(1n), rlogin(1n), rsh(1n), rcmd(3n).

NAME

fs, inode — format of file system volume

SYNOPSIS

#include ⟨sys/types.h⟩
#include ⟨sys/fs.h⟩
#include ⟨sys/inode.h⟩
#include ⟨sys/param.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. Sectors are 512 bytes in length.

The actual file system begins at sector 16 with the *super block*. The layout of the super block as defined by the include file ⟨*sys/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 */
        char    fs_fmod;        /* super block modified flag */
        char    fs_clean;       /* file system is clean flag */
        char    fs_ronly;       /* mounted read-only flag */
        char    fs_flags;       /* currently unused flag */
        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 (defined in ⟨sys/param.h⟩) can be optionally broken into 2, 4, or 8 pieces, each of which is addressable; these pieces may be DEV_BSIZE (defined in ⟨sys/dir.h⟩), 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 defined in ⟨sys/fs.h⟩.

The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined.

The root inode, inode 2, is the root of the file system. (Inode 0 can't be used for normal purposes and inode 1 was once used for linking bad blocks, so inode 2 is used for the root inode.) 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 super-user 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 (defined in ⟨sys/fs.h⟩) 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) (defined in ⟨sys/fs.h⟩).

MINBSIZE (defined in ⟨sys/fs.h⟩) 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 (defined in ⟨sys/fs.h⟩) 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 (defined in ⟨sys/fs.h⟩).

The path name on which the file system is mounted is maintained in *fs_fsmnt*. MAXMNTLEN (defined in ⟨sys/fs.h⟩) 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 (defined in ⟨sys/fs.h⟩). 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 (defined in ⟨sys/fs.h⟩) 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 (defined in ⟨sys/fs.h⟩) 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 UTek 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 ⟨*sys/inode.h*⟩.

SEE ALSO

fstab(5).

NAME

magic — magic number file

DESCRIPTION

The **magic** file */usr/lib/magic* contains information about certain kinds of special files that exist in the system, like object code files, compacted files, and archive files. The utility *file(1)* uses the **magic** file to print information about files. The subroutine *notmagic(3c)* uses the **magic** file to determine if a file is a special type of file, not suitable for printing on a terminal.

The **magic** file contains three types of lines. Lines beginning with the character '#' are comments and are ignored. Lines beginning with '0' are considered to be magic number specification lines. If a line begins with a '>', it is a continuation of the previous line and contains information about following bytes.

The format of a specification line is:

```
0    type    number    description
```

All fields are separated by tabs. The type may be one of **byte, short, long,** or **string.** *Short* numbers are interpreted as *unsigned short.* If the magic *number* is not a string, it may be an octal, hex, or decimal number. Octal numbers are specified by beginning them with a '0'. Hex numbers must begin with '0x'. Decimal numbers do not begin with '0' or '0x'. The *description* is a string that describes the type of file the magic number describes. It may contain one **printf** style specifier (such as '%d') which *file* will replace by the value of the magic number.

The following are examples of magic number specification lines. The first line describes a compacted text file (see *compact(1)*), and the second describes a long format archive file (see *ar(5)*).

```
0       short           017777          Compacted text
0       string          !<ARCH>         ASCII archive (long
format)
```

The format for a continuation line is:

```
>offset    type    op    description
```

Again, fields are separated by tabs. The *offset* is the number of bits that should be checked in the next field, whose size is specified by the type field. The *op* is one of '=', '>', or '<' followed by a number, the character 'x', or a string. If the *op* is '>' followed by a number, the corresponding data from the file must be a number that is greater than the number given after the '>'. The characters '<' and '=' are similar in function and mean 'less than' and 'equal to'. The character 'x' means that any number will match. If the *type* field is "string", the *op* field is a string to be matched. The special string ".*" matches any string. This is useful for printing null-terminated strings found at a known place in the file. The *description*

field is as before, except that a '%' specifier is replaced by the value compared against *op* field.

The following lines show the description for an old style executable object file.

```
0       short           0407            Old executable
>16     long            >0              not stripped
```

The '>16' in the second line implies that the next value to be checked starts at the 16th byte in the file. The '>0' in the *op* field specifies that the file matches the description (''not stripped'') if the value is greater than 0.

This is the description for a file which contains the magic number 0177512 followed by a null terminated string which is the file description.

```
0       long            0177512         Described file
>4      string          .*              -- %s
```

FILES

/usr/lib/magic The magic number file.

CAVEATS

Most software that uses the **magic** file will not check the format of the file very carefully. See the manual page for *file(1)*. This utility has the ability to check the format.

The programs **vi, ex, e, edit,** and **view** can only handle 300 magic numbers of type long and/or short.

SEE ALSO

ex(1), file(1), more(1), notmagic(3c).

NAME

man — manual page control files and directories

DESCRIPTION

The system contains various commands which work with manual pages. This document describes the manual page naming conventions, the layout of the manual page file directories, and the format of the system control files used by these commands.

Manual Page File Names

Each manual page file name is of the form *title.section* . The *title* is typically the name of the command, file, subroutine, or concept that the page describes, but a page may refer to a logical grouping of these. The *section* consists of a number from 1 to 8 followed by zero or more alphabetic characters. For example, this manual page is in a file called *man.5man*, and the manual page describing the Bourne shell built-in command *type* is in a file called *type.1sh*. If this name format is not followed, the *man* command may not be able to find the page. The case of section names is ignored by the *man* command, so section '3sh' is equivalent to '3SH'.

Manual Page Directory Layout

The manual page commands expect to look in a directory and find manual page files in the subdirectories 'man[1-8]' and 'cat[1-8]', and a special database file called *whatis* (none of these are required) The directories 'man[1-8]' are expected to contain the manual page sources, and the directories 'cat[1-8]' are expected to contain the formatted pages. The number at the end of the directory name refers to the section number of the manual page. For example, the directories 'man1' and 'cat1' would contain manual page files with names of the form '*.1*'.

Each command uses the subdirectories differently. The command **man** looks only in 'cat[1-8]' for the formatted pages. The command **catman** reformats the pages in 'man[1-8]' that are newer (have been modified more recently than) the corresponding pages in 'cat[1-8]'. The commands **help, section,** and **buildif** work with the manual page index format tables (described in *manindex(5man)).* The command **makewhatis** builds the special *whatis* database from the files in 'cat[1-8]'.

Manual Page Control Files

The directory */usr/lib/man* contains two manual page control files: *directories* and *sections,* which are used by the various commands to decide which actions to take.

The *directories* file contains lines of the form

```
man-directory command-directory actions
```

The *man-directory* is the name of a directory which contains manual page subdirectories and a *whatis* database.

The *command-directory* is the name of a directory which contains the commands corresponding to the manual pages. For example, the directory */usr/man* contains manual pages for the commands contained in

/bin, */usr/bin*, and*/etc*. (Since there may be more than one command directory which corresponds to a manual page directory, multiple entries beginning with the same *man-directory* are allowed.) This correspondence is used by the **man** command to base manual page directory searching order on the contents of the PATH environment variable.

The *actions* part of the line is a set of letters which tell the command **catman** what to do with the manual pages in the directories. The valid actions letters are **f**,**i**, and **w**, which are described in the manual page for *catman(8man)*.

If a line begins with a '#', the line is ignored as a comment.

The *sections* file contains the default section ordering used by the **man** command and is a complete list of the known section names. The section names are separated by spaces, tabs, and newlines.

In addition, there may be items of the form [1–8]+. These are used by the **catman** command to decide where new subsections that appear should go. For example, if the sections *3, 3c, 3s, 3n,* and *3f* exist and the users tend not to need Fortran (section 3f) pages, the *sections* file might contain a the sequence "3 3c 3s 3n 3+ 3f". If *catman* finds a new manual page whose section name is *3e,* it would replace the '3+' with "3e 3+", resulting in the sequence "3 3c 3s 3n 3e 3+ 3f". If there are no + specifiers corresponding to a section, new section names are added to the end of the file. See the manual page for *catman* for more information.

FILES

/usr/lib/man/directories	Description of directories where manual pages are found.
/usr/lib/man/sections	List of known manual page sections.
*man[1-8]/**	Manual page source files.
*man[1-8]/**	Formatted manual page files.
whatis	Special manual page description database.

CAVEATS

The name 'x[1–8]' corresponds to the list of names "x1 x2 x3 x4 x5 x6 x7 x8" and not to a single name.

The *sections* file may not contain comments.

SEE ALSO

apropos(1man), buildif(1man), help(1man), makewhatis(1man), man(1man), section(1man), whatis(1man), manindex(5man), whatis(5man), catman(8man).

NAME

manindex — manual page index format structure

DESCRIPTION

In order to work with the commands *help(1man)* and *section(1man),* formatted manual pages must have a index format table at the end of the file. This table is built by the command *buildif(1man),* which may be automatically invoked by *catman(8man).*

The table consists of three sections, the header, the data, and the foot. The header is a line consisting of a formfeed (^L), the word '%%index%%', and a newline. The data section consists of lines which tell where the useful data for the sections of the manual page is found. This is described in more detail later. The foot of the table consists of the word '%%index%%', a 12-digit, 0-padded decimal number which tells how many bytes of data are in the table (header and foot included), and a newline. The entire table is in printable ASCII characters.

The data section of the index format table consists of lines with the following form:

section:*begin*,*length*; . . . ⟨newline⟩

The *section* is a two-letter abbreviation for the section name. The following table shows the section names and their abbreviations:

NAME	na
SYNOPSIS	sy
DESCRIPTION	de
OPTIONS	op
EXAMPLES	ex
FILES	fi
DIAGNOSTICS	di
VARIABLES	va
RETURN VALUE	rv
CAVEATS	ca
SEE ALSO	se
REFERENCES	re

These abbreviations correspond to commands in **help** and the section-list in **section.** The *begin* portion of the line is the decimal offset (beginning at 0) in the manual entry at which point the data for the section begins. The *length* portion is the length of this data portion. There may be multiple *begin*,*length* pairs for each section. The first pair always exists. Subsequent pairs exist when the section contains a page boundary. If a section doesn't exist in a manual entry, no line is generated in the table for it.

EXAMPLES

Assume that a manual page contains the sections NAME, SYNOPSIS, DESCRIPTION, EXAMPLES, RETURN VALUE, CAVEATS, and SEE

ALSO, and that the DESCRIPTION section contains two page breaks and that the RETURN VALUE section contains one page break. The following shows a possible index format table (note that newlines are given as '\n' and the formfeed is given as '\f').

```
\f%%index%%\n
na:72,60;\n
sy:132,139;\n
de:271,1772;2349,1741;4396,361;\n
ex:4757,905;\n
rv:5662,189;6157,205;\n
ca:6362,767;\n
se:7129,277;\n
%%index%%000000000148\n
```

Note that this data begins 148 bytes from the end of the file.

CAVEATS

The command *man(1man)* knows not to print the index format data whereas other programs do not. It is best to only use **man** to print out manual pages.

SEE ALSO

apropos(1man), buildif(1man), help(1man), makewhatis(1man), man(1man), section(1man), whatis(1man), man(5man), whatis(5man), catman(8man).

NAME

mh — mh mail message format

DESCRIPTION

This section paraphrases the format of mail text messages.

ASSUMPTIONS

Messages are expected to consist of lines of text. Graphics and binary data are not handled.

No data compression is accepted. All text is clear ASCII 7-bit data.

LAYOUT

A general "memo" framework is used. A message consists of a block of information in a rigid format, followed by general text with no specified format. The rigidly formatted first part of a message is called the header, and the free-format portion is called the body. The header must always exist, but the body is optional.

THE HEADER

Each header item can be viewed as a single logical line of ASCII characters. If the text of a header item extends across several real lines, the continuation lines are indicated by leading spaces or tabs.

Each header item is called a component and is composed of a keyword or name, along with associated text. The keyword begins at the left margin, may contain spaces or tabs, may not exceed 63 characters, and is terminated by a colon (:). Certain components (as identified by their keywords) must follow rigidly defined formats in their text portions.

The text for most formatted components (e.g., "Date:" and "Message-Id:") is produced automatically. The only ones entered by the user are address fields such as "To:", "cc:", etc. Addresses are assigned mailbox names and host computer specifications. The rough format is "mailbox at host", such as "Borden at Rand-UTek". Multiple addresses are separated by commas. A missing host is assumed to be the local host.

THE BODY

A blank line signals that all following text up to the end of the file is the body. (A blank line is defined as a pair of ⟨end-of-line⟩ characters with *no* characters in between.) No formatting is expected or enforced within the body.

Within **MH**, a line consisting of dashes is accepted as the header delimiter. This is a cosmetic feature applying only to locally composed mail.

MESSAGE NAME BNF

```
msgs: =          msgspec ¦
                 msgs msgspec
msgspec: =       msg ¦
```

```
                    msg-range ¦
                    msg-sequence
msg: =              msg-name ¦
                    ⟨number⟩
msg-name: =         "first" ¦
                    "last"  ¦
                    "cur"   ¦
                    "."     ¦
                    "next"  ¦
                    "prev"
msg-range: =        msg"-"msg ¦
                    "all"
msg-sequence: =           msg":"signed-number
signed-number: =          "+"⟨number⟩ ¦
                    "\^—"⟨number⟩ ¦
                    ⟨number⟩
```

Where ⟨number⟩ is a decimal number in the range 1 to 999. Msg-range specifies all of the messages in the given range and must not be empty. Msg-sequence specifies up to ⟨number⟩ of messages, beginning with *msg* (in the case of first, cur, next, or ⟨number⟩), or ending with *msg* (in the case of prev or last). +⟨number⟩ forces "starting with *msg*", and —⟨number⟩ forces "ending with number". In all cases, *msg* must exist.

SEE ALSO

mh(1mh).

NAME

mh_profile — user parameters for MH message handler

DESCRIPTION

Each user of **mh** is expected to have a file named **.mh_profile** in his or her home directory. This file contains a set of user parameters used by some or all of the **mh** family of programs. Each line of the file is of the format

profile-component: *value*

The currently defined profile components are exemplified below:

Path: Mail — Locates **mh** transactions in directory "Mail".

Current–Folder: inbox
Keeps track of currently open folder.

Editor: prompter — Defines editor to be used by *comp(1mh)*, *repl(1mh)*, and *forw(1mh)*.

Msg—Protect: 644 — Defines octal protection bits for message files. See *chmod(1)* for an explanation of the octal number.

Folder—Protect: 711 — Defines protection bits for folder directories.

program: default switches
Sets default switches to be used whenever the mh program *program* is invoked. For example, one could override the *Editor:profile* component when replying to messages by adding a component such as:

```
repl: -editor prompter
```

cur—*read-onlyfolder*: 172
Keeps track of the last message seen in the specified read–only folder. In folders to which write access is permitted, the current–message value is kept in a file called *cur* within that folder.

prompter—next: ed — Names the editor to be used on exit from *prompter(1mh)*

The following profile elements are used whenever an **mh** program invokes some other program such as *refile(1mh)* or *ls(1)*. The **mh_profile** can be used to select alternate versions of these programs if the user wishes. The default values are given in the examples.

fileproc: */usr/tek/refile*
installproc: */usr/lib/tek/mh/install-mh*
lproc: */usr/ucb/more*
lsproc: *ls*
mailproc: */usr/tek/mail*
prproc: */bin/pr*
scanproc: */usr/tek/scan*

sendproc: */usr/tek/send*
showproc: */usr/ucb/more*
delete–prog: */bin/rm*

Normally, *rmm(1mh)*, rather than removing a message in file X will rename the file to ,X. If a user provides a *delete-prog* profile entry, the specified program will be used to remove the file.

FILES

$HOME/.mh_profile	Personal MH configuration file.

SEE ALSO

mh(1mh).

NAME

mtab — mounted file system table

SYNOPSIS

#include <fstab.h>
#include <mtab.h>

DESCRIPTION

Mtab resides in directory */etc* and contains a table of devices mounted by the **mount** command. **Mount** adds entries to the table; **umount** removes entries.

The table is a series of *mtab* structures, as defined in *<mtab.h>*:

```
struct mtab {
        char    m_path[32];             /* mounted on pathname */
        char    m_dname[32];            /* block device pathname
        char    m_type[4];              /* read-only, quotas */
};
```

Each entry contains the null-padded name of the place where the special file is mounted, the null-padded name of the special file, and a type field, one of those defined in *<fstab.h>*.

The special file has all its directories stripped away; that is, everything through the last '/' is thrown away. The type field indicates if the file system is mounted read-only, read-write, or read-write with disk quotas enabled.

This table is present only as information to users. It does not matter to **mount** if there are duplicated entries nor to **umount** if a name cannot be found.

FILES

/etc/mtab

SEE ALSO

mount(8).

NAME

.netrc — remote machine login/passwords

DESCRIPTION

The **.netrc** file in a user's home directory contains a list of accounts on other machines. It is used by the **ftp** and **rdump** commands to provide auto-login.

The format is a keyword followed by a value seperated by a space, comma, tab or newline. Used keywords are: machine, login, password. These are keywords not currenly used but recognized: default, notify, write, yes, y, no, n, command, force.

Since this file contains passwords to accounts on other systems the programs that use this file will print an error message and not use the information if it is readable by group or other. This is to encourage the user to protect this sensitive file.

EXAMPLES

```
machine bigvax,login joeuser,password fatchance
machine tekworks,login wiseguy
```

FILES

$HOME/.netrc

SEE ALSO

ftp(1n), telnet(1n), rexec(3x), rdump(8n).

NAME

network.conf — non–volatile storage for network configuration status

DESCRIPTION

The **network.conf** file contains one line with the current hostname, and one line containing the host ID as was last set by the user or by *netconfig(8n)* from an address of one of the network interfaces. Next is a blank line. Following these are another line with either the string *net_enabled*, or *net_disabled,* and another line with either the string *dfs_enabled* or *dfs_disabled.*

The network address is specified in the conventional "." notation using the *inet_addr* routine from the Internet address manipulation library, *inet(3n).* The host name may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/network.conf

CAVEATS

This file is for reading only and is maintained solely by *netconfig(8n).*

SEE ALSO

netconfig(8n), inet(3n).

NAME

networks — network name data base

DESCRIPTION

The **networks** file contains information regarding the known networks which are reachable. 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

CAVEATS

The *nameserver(8n)* may be expanded to handle this function.

SEE ALSO

getnetent(3n).

NAME

oldar — old archive (library) file format

DESCRIPTION

The old archive file format is not supported by current software. These files may be converted to the new format by *arcv(1)*. Programs that work with archives know how to distinguish the old and new archive files, and usually print a message saying that the file is in the old format.

Old archive files are those from the 32v and Third Berkeley edition archive programs. These files have the magic number (unsigned short) 0177545 at the start, followed by the constituent files, each preceded by a file header. The header layout is

```
struct oar_hdr {
        char    oar_name[14];
        short   oar_sdate[2];
        char    oar_uid;
        char    oar_gid;
        unsigned short  oar_mode;
        short   oar_ssize[2];
};
```

The name is a blank–padded string. The date is the modification date of the file at the time of its insertion into the archive.

The date and size fields should be converted to long values for use. They are arrays of short values in the structure in order to make the structure elements contiguous for use with *read(2)*.

There is no provision for empty areas in an archive file.

Unlike the new archive format, the encoding of the header is not portable across machines, and files do not have to begin on an even boundary.

CAVEATS

Old format files must be converted to the new archive format before use. This must be done by using *arcv(1)* and not by editing the file, since this may result in unrecoverable data.

The programs *more(1)* and *ex(1)* do not allow viewing or editing of old format archives.

SEE ALSO

ar(1), arcv(1), ar(5).

NAME

passwd — password file

DESCRIPTION

Passwd contains for each user the following information:

name (login name, contains no upper
encrypted password
numerical user ID
numerical group ID
user's real name, office, extension, home
initial working directory
program to use as Shell

The name may contain **&**, meaning insert the login name. This information is set by the *chfn(1)* command and used by the *finger(1)* command.

This 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, then */bin/sh* is used.

This file resides in directory */etc*. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

Appropriate precautions must be taken to lock the file against changes if it is to be edited with a text editor; *vipw(8)* does the necessary locking.

FILES

/etc/passwd

SEE ALSO

chfn(1), login(1), passwd(1), finger(1), getpwent(3), group(5), vipw(8).

NAME

phones — remote host phone number data base

DESCRIPTION

The file */etc/phones* contains the system-wide private phone numbers for the *tip(1n)* 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(5n)* file and the phone number is constructed from [0123456789-=*%]. 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.

Only one phone number per line is permitted. However, if more than one line in the file contains the same system name *tip(1n)* will attempt to dial each one in turn, until it establishes a connection.

FILES

/etc/phones	System-wide phone numbers

SEE ALSO

tip(1n), remote(5n).

NAME

protocols — protocol name data base

DESCRIPTION

The **protocols** file contains information regarding the known protocols used in the local network. 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

CAVEATS

These numbers are hard coded into many programs and the kernel (see */usr/include/netinet/in.h*, */usr/include/sys/socket.h*)

SEE ALSO

getprotoent(3n).

NAME

prskeywords — keyword descriptions for prs

DESCRIPTION

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile(5sccs)*) have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by *prs* consists of: (1) the user supplied text; and (2) the appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either *Simple* (S), in which keyword substitution is direct, or *Multi-line* (M), in which keyword substitution is followed by a carriage return.

User supplied text is any text other than recognized data keywords. A tab is specified by **\t** and carriage return/new–line is specified by **\n**.

TABLE 1. SCCS Files Data Keywords

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu:	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	*D*˜or˜*R*	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th:::Tm:::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS:˜:DS: ...	S
:Dx:	Deltas excluded (seq #)	"	:DS:˜:DS: ...	S
:Dg:	Deltas ignored (seq #)	"	:DS:˜:DS: ...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	*yes*˜or˜*no*	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	*yes*˜or˜*no*	S
:BF:	Branch flag	"	*yes*˜or˜*no*	S
:J:	Joint edit flag	"	*yes*˜or˜*no*	S
:LK:	Locked releases	"	:R: ...	S
:Q:	User defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	*yes*˜or˜*no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(1sccs) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(1sccs) string	N/A	:Z::Y:˜:M:˜:I::Z:	S
:Z:	*what*(1sccs) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file path name	N/A	text	S

* :Dt:˜ = ˜:DT:˜:I:˜:D:˜:T:˜:P:˜:DS:˜:DP:

SEE ALSO

admin(1sccs), delta(1sccs), get(1sccs), prs(1sccs), rmdel(1sccs), sccshelp(1sccs), sccsfile(5sccs).

NAME

qconf — MDQS configuration file

SYNOPSIS

/etc/qconf

DESCRIPTION

There are four sections to the Multi-Device Queuing System configuration file, each separated by a row of hyphens. The first section specifies MDQS parameter values, the second specifies devices used, the third specifies queue names, and the fourth specifies device/queue/server mappings.

The */etc/sysadmin* program should be used for all but exceptional cases of initializing and updating the MDQS configuration file. The help facilities of this program are useful for additional explanation of */etc/qconf* parameters.

In */etc/qconf,* the "#" character is a comment character signifying that the rest of the line is to be ignored. Extra spaces, tabs and blank lines are ignored, and double-quoted strings are treated as single tokens.

Entries in the first section have the form "parameter value". The following parameters are recognized:

print-queue ⟨*queue*⟩
print-forms ⟨*form*⟩
print-prior ⟨*priority*⟩
print-hdr ⟨*headerfile*⟩
print-hdrdir ⟨*directory*⟩
print-limit ⟨*pages*⟩

The above parameters control the default behavior of the **lpr** program. The *queue* must be specified in the third section of */etc/qconf,* and may be directed to a remote queue (in the fourth section of */etc/qconf)* via the **netsend** server. The *form* is used to direct entries from the default queue to the default device. The *form* must be associated with the device via the **qdev** program; this is handled automatically when using the **sysadmin** program. The *form* must be a valid form as specified in the MDQS **forms** file. *Priority* is in the range 0–10, with 0 being the highest priority. *Print-hdr* specifies the default file to be used as part of the banner page logo. *Print-hdrdir* specifies the directory which may contain header files for shared use (see **lpr —H**). *Print-limit* specifies the maximum number of pages a printing request may output. A limit of 0 indicates no limit.

batch-queue ⟨*queue*⟩
batch-forms ⟨*form*⟩
batch-prior ⟨*priority*⟩

These parameters specify defaults for the **batch** program. The meanings of *queue, form,* and *priority* are similar to the meanings for the print parameters defined above.

console ⟨*filename*⟩
is the file opened by **mdqsd** as *stderr*.

scanwait ⟨*number*⟩
sets the default time in seconds that **mdqsd** will sleep if there are no new requests or finished requests. It is recommended that *number* be 60 so that the delayed queue will be checked once a minute in the absence of new activity.

openwait ⟨*number*⟩
specifies the number of seconds the daemon will wait to retry opening a device if a device open fails.

maxfailures ⟨*number*⟩
If this variable is non–zero, the daemon will flag a device as failed if the server on that device fails *number* times. If this happens, the device can be restarted by disabling and re–enabling the device with the **qdev** program. If disabling and re–enabling the device doesn't work, the MDQS daemon can be killed and restarted via the *daemon(8)* program.

sysmgr ⟨*address*⟩
Specifies where to mail orphan notices. This address defaults to *mdqs*.

netwait ⟨*number*⟩
Specifies the amount of time (in minutes) to delay the retry of a request that failed due to network errors.

Section two contains definitions for devices, where each line is of the form "logical–device real–device forms status". *Logical-device* is the parameter used to map devices to queues in the fourth section of */etc/qconf,* and may be used as a parameter to **qdev**. The *real-device* should be a real device name, i.e., it should begin with */dev/*. Examples of real devices used for printer ports are */dev/tty1* and */dev/hc**.

The *net* is a special *logical-device* which should always map here to */dev/null* and should always map to the **netsend** server in the fourth section. The **batch** program should always submit to queues mapped to logical devices mapped to */dev/null.* The *forms* field specifies what forms are associated with the device. *Anyform* indicates that this device can accept requests regardless of what forms were specified for the request. The *status* is a set of symbolic flags used to control the behavior of a device. *Skipmsg* disables the sending of completion messages on successful completion of a request on the device. *Roundrobin* causes the device to use a roundrobin algorithm in selecting requests from several queues.

The third section simply contains queue names and an optional status field, one per line. Queue names can be specified in submit programs such as **lpr** and in status programs such as **lpq.** The *status* field has one option form=*form* which will set the forms field of all requests submitted to this queue with the form of *form* if no form was explicitly designated when the request was submitted.

The fourth section specifies the mapping of queues, devices, and servers. A line in this section is of the form *queue device server* . *Queue* and *device* must have been previously defined in sections three and two respectively. A mapping specifies that requests from *queue* are to be serviced by the *device* (if the forms match those associated with the device) and the request is to be processed using the **server** program. The **server** process is run as setuid to the user id of the user who submitted the request whether the request is run locally or remotely.

If multiple queues are mapped to one device, a request is taken from the queue with the first listed queue/device mapping for this device, and if this queue is empty, the request is taken from the queue with the next queue/device mapping for this device, etc. (this can be overridden by using the *roundrobin* status flag in section two).

A queue can hold requests directed to several devices, and these requests can be partitioned by associating forms with a device.

FILES

/etc/qconf */usr/lib/mdqs/forms*
/usr/lib/mdqs/remote.access

CAVEATS

/etc/sysadmin is the normal and preferred way to initialize and update */etc/qconf*. When */etc/qconf* is modified, */etc/mdqsd* should be killed and restarted – see *mdqsd(8mdqs)*. *Rsh* privileges are required in order to execute remote requests.

SEE ALSO

forms(5mdqs), mdqsd(8mdqs), remote.access(5mdqs), lpserver(8mdqs), sysadmin(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 optinal 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.

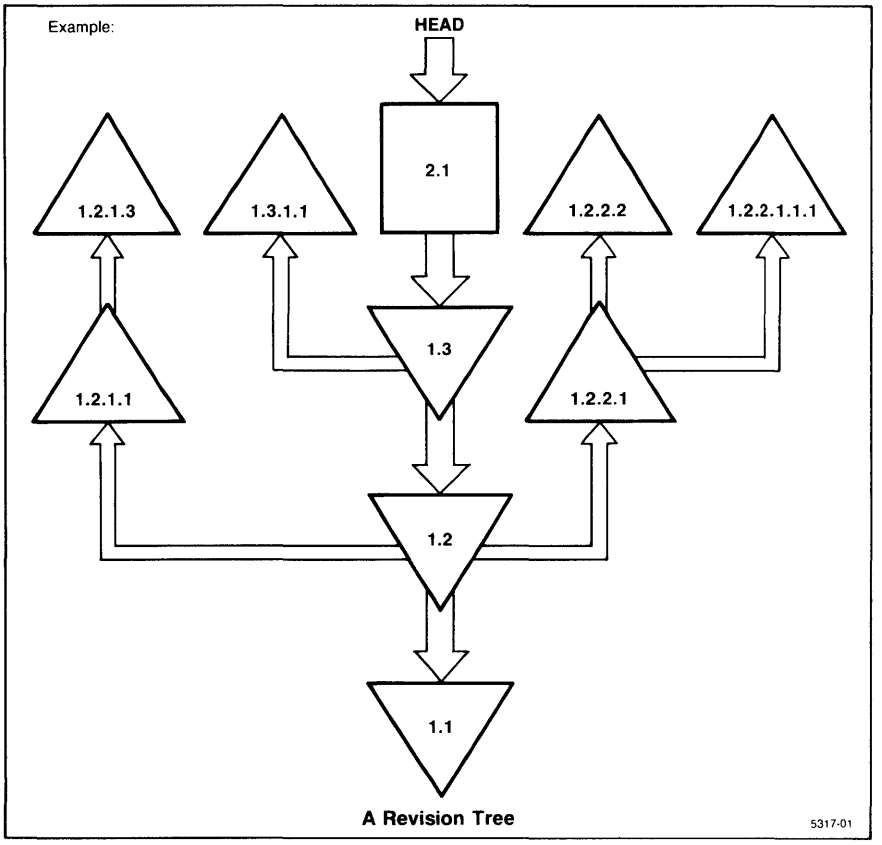
<rcstext>	::=	<admin> {<delta>}* <desc> {<deltatext>}*
<admin>	::=	**head** {<num>};
		access {<id>}*;
		symbols {<id> : <num>}*;
		locks {<id> : <num>}*;
		comment {<string>};
<delta>	::=	<num>
		date <num>;
		author <id>;
		state {<id>};
		branches {<num>}*;
		next {<num>};
<desc>	::=	**desc** <string>
<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 ($n \geq 2$) (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.



Maximum Number of Revisions

When an RCS file contains 700 or more revisions, all RCS commands except for *ident* and *rlog* **—c** will print a warning message (if possible) saying that the maximum number of revisions is about to be reached. When the file contains 719 revisions, no further checkins are allowed. This maximum applies to the total number of revisions in all branches. Starting a new branch will not release any space.

There are two things that can be done when this happens. The first is to delete some of the revisions using the **—o** flag of the *rcs* command. This should be done with some care, making sure that significant modifications are kept separate.

The other method of fixing this problem is to make a copy of the RCS file and delete all of the old revisions in the original file. For example, if the RCS file *prog.c,v* has 715 revisions (1.1 through 1.716), the following commands will save the first 700 revisions in another file, and leave the last 15 revisions where they can be easily found.

```
cp prog.c,v prog.old,v
rcs -o1.701-1.716 prog.old
rcs -o1.1-1.700 prog.c
```

Revision 1.348 can be retrieved by the command **co –r 1.348 prog.old**. Revisions after 1.700 can be checked out of *prog.c,v*. Note that the **rcs** command may take a while to delete 700 revisions.

SEE ALSO

ci(1rcs), co(1rcs), ident(1rcs), rlog(1rcs), rcs(1rcs), rcsdiff(1rcs), rcsintro(1rcs), rcsmerge(1rcs).

NAME

remote — remote host description file

DESCRIPTION

The systems known by *tip(1n)* and their attributes are stored in an ASCII file which is structured somewhat like the *termcap(5t)* 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 comes 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 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) UTek device(s) to open to establish a connection. If this file refers to a terminal line, *tip(1n)* attempts to perform an exclusive open on the device to insure only one user at a time has access to the port.

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; see *phones(5n)*.

tc (str) Indicates that the list of capabilities is continued in the named description. This is used primarily to share common capability information.

EXAMPLES

The following example shows the use of the capability continuation feature:

```
UTek-1200:\
                :dv=/dev/cau0:el=^D^U^C^S^Q^O@:du:at=ventel:ie=#$%:oe=^D:br#1200:
arpavax|ax:\
                :pn=7654321%:tc=UNIX-1200
```

FILES

/etc/remote

SEE ALSO

tip(1n), phones(5n).

NAME

remote.access — control remote access to MDQS services.

DESCRIPTION

The file **remote.access** determines which users on which remote hosts can access local MDQS services. This file is read by the MDQS **netrecv** program when it receives a request from a remote user that does not have *rsh* privileges on the local host. An example **remote.access** file follows:

```
# allow access by the following users:
#⟨remote host⟩       ⟨local user⟩    ⟨remote user⟩
host1               clarence
host1               leon            root
host2               doug            *
*                   allen
```

Note that # is a comment character. The first entry allows access by *clarence* from *host1*. For security reasons, allowing access by *root* is not often appropriate. The second entry allows access by *root* from *host1*, but the local system treats the request as if it came from *leon* and so access is based on *leon*'s rights, not those of *root*. Note that if two users have the same alias, only one of those aliases takes effect. In other words, it is not possible for multiple users to have an entry with the same alias because when a request is received from that alias, there is no way of knowing whose access rights to use. This issue is resolved by assigning rights based on the first entry it finds in the table. The third entry allows access by any user from *host2*, but the local system treats the request as if it came from *doug*. The fourth entry allows access by *allen* from any system.

Remember that **remote.access** determines if the MDQS will accept a request from a remote user that does not already have *rsh* privileges. When setting up **remote.access** keep in mind that the local users mentioned must already be in the password file (*/etc/passwd*). They do not, however, have to have login privileges, nor do they need a home directory. You can prevent them from logging on by setting their encrypted password to '*' or any other single letter (because no password encrypts to a single character).

FILES

/usr/lib/mdqs/remote.access
MDQS remote access database file.

SEE ALSO

mdqsd(8mdqs), qconf(5mdqs).

ÅME

sccsfile — format of SCCS file

ESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the **ASCII SOH** (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form **DDDDD** represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

@hDDDDD

The value of the checksum is the sum of all characters, except those of the first line. The **@h** provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

@s DDDDD/DDDDD/DDDDD
@d ⟨type⟩ ⟨**SCCS ID**⟩ yr/mo/da hr:mi:se ⟨pgmr⟩ **DDDDD DDDD**
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m ⟨**MR** number⟩
.
.
.
@c ⟨comments⟩ ...
.
.
.
@e

The first line (**@s**) contains the number of lines inserted/deleted/unchanged respectively. The second line (**@d**)

contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@m** lines (optional) each contain one **MR** number associated with the delta; the **@c** lines contain comments associated with the delta.

The **@e** line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new–lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines **@u** and **@U**. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see *admin(1sccs)* for more information on their use). Each flag line takes the form:

@f ⟨flag⟩ ⟨optional text⟩

The following flags are defined:

@f t ⟨type of program⟩
@f v ⟨program name⟩
@f i
@f b
@f m ⟨module name⟩
@f f ⟨floor⟩
@f c ⟨ceiling⟩
@f d ⟨default–sid⟩
@f n
@f j
@f l ⟨lock–releases⟩
@f q ⟨user defined⟩
@f z ⟨reserved for use in interfaces⟩

The **t** flag defines the replacement for the **%Y%** identification keyword. The **v** flag controls prompting for **MR** numbers in addition to comments; if the optional text is present it defines an **MR** number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when

the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **—b** keyletter may be used on the **get**^ command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the **%M%** identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a **get**^ command. The **n** flag causes **delta**^ to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes **get**^ to allow concurrent edits of the same base SID. The **l** flag defines a **list**^ of releases that are **locked**^ against editing (see *get(1sccs)* with the **—e** keyletter). The **q** flag defines the replacement for the **%Q%** identification keyword. **z** flag is used in certain specialized interface programs.

Comments

Arbitrary text surrounded by the bracketing lines **@t** and **@T**. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert*, *delete*, and *end*, represented by:

@I DDDDD
@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1sccs), delta(1), get(1sccs), prs(1).

NAME

services — service name data base

DESCRIPTION

The **services** file contains information regarding the known services available on the local network. 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

CAVEATS

Many programs do not read this file to find their port numbers, therefore some port numbers can not be changed.

The *nameserver(8n)* does not provide this information since it is not machine specfic.

SEE ALSO

getservent(3n).

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 debugger *sdb(1)* and the compilers. 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
 * 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 symbolic debugger, *sdb*(1), uses the following n_type values:

```
#define N_GSYM    0x20 /* global symbol: name,,0,type,0 */
#define N_FNAME   0x22 /* procedure name (f77 kludge): 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 the **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 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 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), sdb(1), a.out(5).

NAME

sysdef — System Definition file for system configuration

DESCRIPTION

A system definition file describes the configuration of a system for use by *sysconf(8)*.

A # indicates the rest of the file is a comment.

Keywords are used to indicate interpretation of the rest of the line. The active device drivers for a kernel are listed in the format:

```
device   sn
cdevice  sn  sn_controller
```

device and **cdevice** are keywords indicating *sn* is the signature name of a device. The keyword **controller** may be used in place of **device** to indicate a controller. **cdevice** indicates the device named by *sn* is controlled by the device named *sn_controller*.

The keyword **option** indicates the next field contains the name of a pre-linked kernel object for system configuration.

```
option   object_file
```

Parameters which may be set for a kernel are listed in the format:

```
int      variable value
```

All *variables* are names of integers and *value* is assumed to be an integer.

The device number of files to be used as the root device, the dump device and the argument holding device are given by:

```
dev      rootdev        major   minor
dev      dumpdev        major   minor
dev      argdev         major   minor
```

The current timezone setting is shown as:

```
timezone         minutes          dst
```

minutes is a an integer value indicating the number of minutes west of Greenwich and *dst* is a integer indicating the type of daylight savings time in affect.

SEE ALSO

sysconf(8).

NAME

tar — tape archive file format

DESCRIPTION

Tar, (the tape archive command) dumps several files into one, in a medium suitable for transportation.

A "tar tape" or file is a series of blocks. Each block is of size TBLOCK. A file on the tape is represented by a header block which describes the file, followed by zero or more blocks which give the contents of the file. At the end of the tape are two blocks filled with binary zeros, as an end–of–file indicator.

The blocks are grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the **b** keyletter on the *tar(1)* command line — default is 20 blocks) is written with a single system call; on nine–track tapes, the result of this write is a single tape record. The last group is always written at the full size, so blocks after the two zero blocks contain random data. On reading, the specified or default group size is used for the first read, but if that read returns less than a full tape block, the reduced block size is used for further reads.

The header block looks like:

```
#define TBLOCK  512
#define NAMSIZ  100

union hblock {
        char dummy[TBLOCK];
        struct header {
                char name[NAMSIZ];
                char mode[8];
                char uid[8];
                char gid[8];
                char size[12];
                char mtime[12];
                char chksum[8];
                char linkflag;
                char linkname[NAMSIZ];
        } dbuf;
};
```

Name is a null–terminated string. The other fields are zero–filled octal numbers in ASCII. Each field (of width w) contains w–2 digits, a space, and a null, except *size* and *mtime*, which do not contain the trailing null. *Name* is the name of the file, as specified on the *tar* command line. Files dumped because they were in a directory which was named in the command line have the directory name as prefix and */filename* as suffix. *Mode* is the file mode, with the top bit masked off. *Uid* and *gid* are the user and group numbers which own the file. *Size* is the size of the file in bytes. Links and symbolic links are dumped with this field specified as zero. *Mtime* is the modification time of the file at the time it was dumped.

Chksum is a decimal ASCII value which represents the sum of all the bytes in the header block. When calculating the checksum, the *chksum* field is treated as if it were all blanks. *Linkflag* is ASCII '0' if the file is "normal" or a special file, ASCII '1' if it is an hard link, and ASCII '2' if it is a symbolic link. The *filename* linked–to, if any, is given in *linkname,* with a trailing null. Unused fields of the header are binary zeros (and are included in the checksum).

The first time a given inode number is dumped, it is dumped as a regular file. The second and subsequent times, it is dumped as a link instead. Upon retrieval, if a link entry is retrieved, but not the file it was linked to, an error message is printed and the tape must be manually re–scanned to retrieve the linked–to file.

The encoding of the header is designed to be portable across machines.

CAVEATS

Names or linknames longer than NAMSIZ produce error reports and cannot be dumped.

SEE ALSO

tar(1).

NAME

tcp_servers — tcpd services list

DESCRIPTION

The **tcp_servers** file contains a information regarding the services which are started by *tcpd(8n)*. For each service a single line should be present with the following information:

name
command
arguments

The *name* is the name of the service as listed in *services(5n)*. The *command* is either the full path or path relative to the server directory (*/etc/tcp_services*) of the command being run. The arguments are the passed to the service when it is run.

Items are separated by any number of blanks and/or tab characters. A # indicates the beginning of a comment.

The following are examples of service specification lines.

```
# Tcpd configuration file sample
login   rlogind
shell   rshd
smtp    /usr/lib/sendmail -bs
echo    /bin/cat -u
daytime /bin/date
```

FILES

/etc/tcp_servers

CAVEATS

Commands are executed directly, rather than being interpreted by *sh(1sh)*, so no redirection is possible.

SEE ALSO

services(5n), tcpd(8n).

NAME

termcap — terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminal attributes; it is often used by *vi(1)*, *curses(3t)*, and other programs requring terminal values to be selected. 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 the number of lines affected

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

dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give ":ei=:" if **ic**
eo	str		Can erase overstrikes with a blank
EP	bool		Set even parity (used by *tset(1)*)
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half–line down (forward 1/2 linefeed)
HD	bool		Half–duplex terminal (used by *tset(1)*)
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	str		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
LC	bool		Terminal can send lower case (used by *tset(1)*)
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		Enter blinking mode.
md	str		Enter bold mode.
me	str		Exit 'modes' (**mb, md, mh, mk, mr**).
mh	str		Enter dim mode.
mi	bool		Safe to move while in insert mode
mk	str		Enter concealed mode.
ml	str		Memory lock on above cursor.
mr	str		Enter reverse video mode.

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)
NL	bool		Disable return-newline mapping (used by *tset(1)*)
NP	bool		Turn off parity (used by *tset(1)*)
ns	bool		Terminal is a CRT but doesn't scroll.
OP	bool		Set odd parity (used by *tset(1)*)
os	bool		Terminal overstrikes
pb	num		Minimum baud rate requiring padding (used by *tset(1)*)
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with **is**)
rf	str		Name of file containing **ir**
rs	str		Terminal reset string
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)
st	str	(P)	Set tab at current position (used by *tset(1)*)
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
UC	bool		Terminal can send upper case only (used by *tset(1)*)
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue
ul	bool		Terminal underlines without special sequences
up	str		Upline (cursor up)
us	str		Start underscore mode
vb	str		Visible bell (may not move cursor)
ve	str		Sequence to end open/visual mode
vs	str		Sequence to start open/visual mode
xb	bool		Beehive (f1 = escape, f2 = ctrl C)
xn	bool		A newline is ignored after a wrap (Concept)
xr	bool		Return acts like **ce** \r \n (Delta Data)
xs	bool		Standout not erased by writing over it (HP 264?)
xt	bool		Tabs are destructive, magic so char (Teleray 1061)

A Sample Entry

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

```
c1¦¦¦c100¦¦¦concept100:is=\EU\Ef\E7\E5\E8\E1\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#
        :dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:li#24:mi:n
        :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 empty fields may be included for readability (shown 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 that 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 a *, 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' to specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A **\E** maps to an ESCAPE character, ^**x** maps to a control-x for any appropriate x; the sequence **\n** gives a newline, **\r** a return, **\t** a tab, **\b** a backspace, and **\f** a 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**, 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*.1* Then position the cursor before the **abc** and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the **abc** shifts over to the **def** which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not not falling into one of these two classes.

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

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

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

Highlighting, underlining, and visible bells

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

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

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

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

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

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

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**. Otherwise the

keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kl, kr, ku, kd,** and **kh** respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0, k1, ..., k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0, l1, ..., l9**. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the **termcap** 2 letter codes can be given in the **ko** capability, for example, ":ko=cl,ll,sf,sb:", which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

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

Miscellaneous

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

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

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

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

Similar Terminals

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

termlib routines search the entry from left to right, and since the **tc** capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be 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

CAVEATS

Ex allows only 2048 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 2048.

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.

The **ti** and **te** entries are specifically for the purpose of setting up the terminal for cursor motion. It should not be used to clear the screen. The program *more(1)* may need to use these strings in some cases, and improper setting of these entries may cause problems.

The **ul** entry tells programs that the terminal will perform underlining when given the sequence ^H_x or _^Hx (^H is a backspace). It does not mean that the terminal has the capability to do underlining via an escape sequence.

SEE ALSO

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

NAME

ttys — terminal initialization data

DESCRIPTION

The **ttys** file is read by the **init** program and specifies which terminal special files are to have a process created for them so that people can log in. Each terminal special file is a one–line entry in the **ttys** file.

The first character of a line in the **ttys** file is either '0' or '1'. If the first character on the line is a '0', the **init** program ignores that line. If the first character on the line is a '1', the **init** program creates a login process for that line. The second character on each line is used as an argument to *getty(8)*, which performs such tasks as baud–rate recognition, reading the login name, and calling **login**. For normal lines, the character is '0'; other characters can be used, for example, with hard–wired terminals where speed recognition is unnecessary or which have special characteristics. (**Getty** will have to be fixed in such cases.) The remainder of the line is the terminal's entry in the device directory, */dev*.

FILES

/etc/ttys

SEE ALSO

login(1), gettytab(5), getty(8), init(8).

NAME

ttytype — data base of terminal types by port

SYNOPSIS

/etc/ttytype

DESCRIPTION

Ttytype is a database with information about the kind of terminal attached to each tty port on the system. There is one line per port, giving the terminal type (as a name listed in *termcap(5t),* a space, and the name of the **tty**, minus */dev/*.

This information is read by *tset(1)* and by *login(1)* to initialize the *TERM* variable at login time.

SEE ALSO

login(1), tset(1).

NAME

types — primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data **types** defined in the include file *<sys/types.h>* are used in UTek system code; some data of these **types** are accessible to user code, as follows:

```
/*
 * Basic system types and major/minor device constructing/busting mac
 */

/* major part of a device */
#define major(x) ((int)(((unsigned)(x)>>16)&0177777))

/* minor part of a device */
#define minor(x) ((int)((x)&0177777))

/* device to slot or controller number */
#define devslot(x)      ((int)(((x)>>8)&0377))

/* device to unit number and device-dependent flags */
#define devunit(x)      ((int)((x)&0377))

/* make a device number from major/minor pair */
#define makedev(x,y)    ((dev_t)(((x)<<16) | (y)))

/* make a device number from major/slot/unit triple */
#define devno(x,y,z)    ((dev_t)(((x)<<16) | ((y)<<8) | (z)))


typedef unsigned char  u_char;
typedef unsigned short u_short;
typedef unsigned int   u_int;
typedef unsigned long  u_long;
typedef unsigned short ushort;/* sys III compat */


typedef struct   _physadr { int r[1]; } *physadr;
typedef struct   label_t {
        int      val[8];
} label_t;
typedef char *  gaddr_t;/* global address */

typedef struct   _quad { long val[2]; } quad;
typedef long     daddr_t;
typedef char *   caddr_t;
typedef u_long   ino_t;
```

```
typedef long    swblk_t;
typedef int     size_t;
typedef int     time_t;
typedef long    dev_t;
typedef int     off_t;

typedef struct  fd_set { int fds_bits[1]; } fd_set;
#endif
```

The form *daddr_t* is used for disk addresses except in an inode 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

adb(1), lseek(2), time(3c), time(3f), fs(5).

NAME

utmp — login records

SYNOPSIS

#include <utmp.h>

DESCRIPTION

The **utmp** file records information about who is currently using the system. The file is a sequence of entries with the following structure declared in the include file:

```
struct utmp {
        char    ut_line[8];             /* tty name */
        char    ut_name[8];             /* user id */
        char    ut_host[16];            /* host name, if remo
        long    ut_time;                /* time on */
};
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of *time(3c)*.

FILES

/etc/utmp

/usr/adm/wtmp

SEE ALSO

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

NAME

uuencode — format of an encoded uuencode file

DESCRIPTION

Files output by *uuencode(1n)* consist of a header line, followed by a number of body lines, and a trailer line. *Uudecode(1n)* 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

mail(1mh), uucp(1n), uudecode(1n), uuencode(1n), uusend(1n).

NAME

whatis — manual page description database

DESCRIPTION

The commands **apropos** and **whatis** search files known as **whatis**databases. These databases are created by the command **makewhatis** and should exist in each manual page directory.

A **whatis** database will contain a listing for each manual page in the manual page directories. The file is sorted by section number, but pages are not sorted within each section.

Each database entry has three fields separated by tabs, and no entry may contain tabs. The entries are separated by newlines. The first field of the entry is the manual page name. The second field of the entry is the manual page section. The last field is the manual page description. Most manual pages will have a NAME section like:

```
command - what this command does
```

This will cause one entry of the form:

```
command<TAB>section<TAB>description
```

Some manual pages will have a NAME section like:

```
command1, command2, ... - what these commands do
```

In this case, there will be a database entry for each of 'command1', 'command2', and so forth. Each entry will have the same section and description fields, but each name field will be different. (These correspond to links made by *makewhatis* for multiple commands.)

FILES

whatis	Special manual page description database.

SEE ALSO

apropos(1man), buildif(1man), help(1man), makewhatis(1man), man(1man), section(1man), whatis(1man), man(5man), manindex(5man), catman(8man).

NAME

wsdumptable — static information about dumping filesystems

DESCRIPTION

The file */etc/wsdumptable* contains descriptive information about various workstations and filesystems that will be dumped by **wsdump(8n).** The file is split up into two sections. The first section contains information on how often to dump at each level.

level (dump level number, range 0–9; see *dump(8)*).
frequency (how often to dump at this level – in days)
groups (how many groups to dump at this level each night)

The second section contains information on each filesystem to be dumped.

workstation (workstation that the filesystem resides on)
filesystem (name of the filesystem)
levels (the levels to dump this filesystem at)
group (the group that this filesystem belongs to)

This is an ASCII file. Each field within each entry is separated from the next by a colon. Each entry is separated from the next by a new–line. Lines starting with the '#' character are treated as comments. Lines starting with the '–' character are treated as section delimiters. */etc/wsdumptable* is only *read* by programs, and not written; it is the duty of the system administrator to properly create and maintain this file. This file can be maintained using the program **viwsb(8).** The order of records in */etc/wsdumptable* will reflect the order in which filesystems will be dumped.

EXAMPLES

```
# wsdumptable
#⟨level⟩:⟨freq⟩:⟨groups⟩
0:30:1
1:7:2
9:1:9
-------------------------
#⟨workstation⟩:⟨filesystem⟩:⟨levels⟩:⟨group⟩
kokomo:/dev/dw00a:019:A
duke:/dev/dw00a:019:B
pejs:/dev/dw00a:019:C
```

In this example level 0 dumps are done every 30 days, level 1 dumps are done every 7 days and level 9 dumps are done every day. Level 0 dumps are done to at most 1 group each day, level 1 dumps are done to at most 2 groups each day and level 9 dumps are done to at most 9 groups each day. The file system */dev/dw00a* on *kokomo* participates in dumps on levels 0, 1 and 9 and is a member of group 'A'. The file system */dev/dw00a* on *duke* participates in dumps on levels 0, 1 and 9 and is a member of group 'B'. The file system */dev/dw00a* on *pejs* participates in dumps on levels 0, 1 and 9 and is a member of group 'C'.

FILES

/etc/wsdumptable

SEE ALSO

viwsb(8n), wsdump(8n).

NAME

wtmp — login records

SYNOPSIS

#include ⟨**utmp.h**⟩

DESCRIPTION

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.

FILES

/etc/utmp	Current state of the machine.
/usr/adm/wtmp	Record of */etc/utmp*.

SEE ALSO

login(1), who(1n), init(8).

NAME

environ — user environment

SYNOPSIS

extern char **environ;

DESCRIPTION

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

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

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

TERM The kind of terminal for which output is to be prepared. This information is used by commands like **nroff** which may exploit special terminal capabilities. See */etc/termcap* (*termcap(5t)*) for a list of terminal types.

SHELL The filename of the user's login shell.

TERMCAP

The string describing the terminal in TERM, or the name of the termcap file, see *termcap(5t)*,*termcap(t)*.

EXINIT A startup list of commands read by *ex(1)*, *edit(1)*, and *vi(1)*.

USER The login name of the user.

MAIL The pathname of the user's incoming mail drop.

Further names may be placed in the environment by the **export** command and *name* = *value* arguments in *sh(1sh)*, or by the **setenv** command if you use *csh(1csh)*. Arguments may also be placed in the environment at the point of an *execve(2)*. It is unwise to conflict with certain *sh(1sh)* variables that are frequently exported by *.profile* files: *MAIL, PS1, PS2, IFS.*

SEE ALSO

csh(1csh), ex(1), login(1), sh(1sh), execve(2), termcap(3t), termcap(5t).

NAME

eqnchar — special character definitions for eqn

SYNOPSIS

eqn /usr/pub/eqnchar [*filenames*] | **troff** [*options*]

neqn /usr/pub/eqnchar [*filenames*] | **troff** [*options*]

DESCRIPTION

Eqnchar contains **troff** and **nroff** character definitions for constructing characters that are not available on the Graphic Systems typesetter. These definitions are primarily intended for use with **eqn** and **neqn**. It contains definitions for the following characters:

ciplus	⊕	||	‖	*square*	□
citimes	⊗	*langle*	⟨	*circle*	○
wig	∼	*rangle*	⟩	*blot*	■
-wig	≃	*hbar*	ℏ	*bullet*	•
>wig	≳	*ppd*	⊥	*prop*	∝
<wig	≲	*<->*	⟷	*empty*	∅
=wig	≅	*<=>*	⟺	*member*	∈
star	⋆	|	∤	*nomem*	∉
bigstar	★	|>	⊳	*cup*	∪
=dot	≐	*ang*	∠	*cap*	∩
orsign	∨	*rang*	∟	*incl*	⊆
andsign	∧	*3dot*	⋮	*subset*	⊂
=del	≜	*thf*	∴	*supset*	⊃
oppA	∀	*quarter*	¼	*!subset*	⊄
oppE	∃	*3quarter*	¾	*!supset*	⊅
angstrom	Å	*degree*	°		

FILES

/usr/pub/eqnchar

NAME

mailaddr — mail addressing description

DESCRIPTION

Mail addresses are based on the ARPANET protocol listed at the end of this manual page. These addresses are in the general format

```
user@domain
```

where a domain is a hierarchical dot separated list of subdomains. For example, the address

```
eric@monet.Berkeley.ARPA
```

is normally interpreted from right to left: the message should go to the *ARPA* name tables (which do not correspond exactly to the physical *ARPANET*), then to the *Berkeley* gateway, after which it should go to the local host *monet*. When the message reaches *monet* it is delivered to the user *eric*.

Unlike some other forms of addressing, this does not imply any routing. Thus, although this address is specified as an *ARPA* address, it might travel by an alternate route if that was more convenient or efficient. For example, at Berkeley the associated message would probably go directly to *monet* over the Ethernet rather than going via the *Berkeley ARPANET* gateway.

Abbreviation. Under certain circumstances it may not be necessary to type the entire domain name. In general anything following the first dot may be omitted if it is the same as the domain from which you are sending the message. For example, a user on *calder.Berkeley.ARPA* could send to *eric@monet* without adding the *.Berkeley.ARPA* since it is the same on both sending and receiving hosts.

Certain other abbreviations may be permitted as special cases. For example, at Berkeley *ARPANET* hosts can be referenced without adding the *.ARPA* as long as their names do not conflict with a local hostname.

Compatibility. Certain old address formats are converted to the new format to provide compatibility with the previous mail system. In particular,

```
host:user
```

is converted to

```
user@host
```

to be consistent with the *rcp(In)* command.

Also, the syntax:

```
host!user
```

is converted to:

```
user@host.UUCP
```

This is normally converted back to the "host!user" form before being sent on for compatibility with older UUCP hosts.

The current implementation is not able to route messages automatically through the UUCP network. Until that time you must explicitly tell the mail system which hosts to send your message through to get to your final destination.

Case Distinctions. Domain names (i.e., anything after the "@" sign) may be given in any mixture of upper and lower case with the exception of UUCP hostnames. Most hosts accept any mixture of case in user names, with the notable exception of MULTICS sites.

Differences with ARPA Protocols. Although the UTek addressing scheme is based on the ARPA mail addressing protocols, there are some significant differences.

At the time of this writing the only "top level" domain defined by ARPA is the *.ARPA* domain itself. This is further restricted to having only one level of host specifier. That is, the only addresses that ARPA accepts at this time must be in the format *user@host.ARPA* (where *host* is one word). In particular, addresses such as:

```
eric@monet.Berkeley.ARPA
```

are not currently legal under the ARPA protocols. For this reason, these addresses are converted to a different format on output to the ARPANET, typically:

```
eric%monet@Berkeley.ARPA
```

Route-addrs. Under some circumstances it may be necessary to route a message through several hosts to get it to the final destination. Normally this routing is done automatically, but sometimes it is desirable to route the message manually. An address that shows these relays are termed "route–addrs." These use the syntax:

```
<@hosta,@hostb:user@hostc>
```

This specifies that the message should be sent to *hosta*, from there to *hostb*, and finally to *hostc*. This path is forced even if there is a more efficient path to *hostc*.

Route–addrs occur frequently on return addresses, since these are generally augmented by the software at each host. It is generally possible to ignore all but the *user@host* part of the address to determine the actual sender.

Postmaster. Every site is required to have a user or user alias designated "postmaster" to which problems with the mail system may be addressed.

CSNET. Messages to CSNET sites can be sent to "user.host@UDel–Relay".

BERKELEY

The following comments apply only to the Berkeley environment.

Hostnames. Many of the old familiar hostnames are being phased out. In particular, single character names as used in Berknet are incompatible with the larger world of which Berkeley is now a member. For this reason the following names are being phased out. You should notify any correspondents of your new address as soon as possible.

OLD		NEW
j	ingvax	ucbingres
p		ucbcad
r	arpavax	ucbarpa
v	csvax	ucbernie
n		ucbkim
y		ucbcory

The old addresses will be rejected as unknown hosts sometime in the near future.

What's My Address? If you are on a local machine, say *monet*, your address is

```
yourname@monet.Berkeley.ARPA
```

However, since most of the world does not have the new software in place yet, you will have to give correspondents slightly different addresses. From the ARPANET, your address would be:

```
yourname%monet@Berkeley.ARPA
```

From UUCP, your address would be:

```
ucbvax!yourname%monet
```

Computer Center. The Berkeley Computer Center is in a subdomain of Berkeley. Messages to the computer center should be addressed to:

```
user%host.CC@Berkeley.ARPA
```

The alternate syntax:

```
user@host.CC
```

may be used if the message is sent from inside Berkeley.

For the time being Computer Center hosts are known within the Berkeley domain, i.e., the ".CC" is optional. However, it is likely that this situation will change with time as both the Computer Science department and the Computer Center grow.

Bitnet. Hosts on bitnet may be accessed using:

```
user@host.BITNET
```

SEE ALSO

mail(1), sendmail(8mh).

NAME

man — macros to typeset manual

SYNOPSIS

nroff —man.IR filename ...

troff —man.IR filename ...

DESCRIPTION

These macros are used to lay out pages of this manual. A skeleton page may be found in the file */usr/man/man0/xxx*.

Any text argument *t* may be zero to six words. Quotes may be used to include blanks in a 'word'. If *text* is empty, the special treatment is applied to the next input line with text to be printed. In this way .I may be used to italicize a whole line, or .SM followed by .B to make small bold letters.

A prevailing indent distance is remembered between successive indented paragraphs, and is reset to default value upon reaching a non–indented paragraph. Default units for indents *i* are ens.

Type font and size are reset to default values before each paragraph, and after processing font and size setting macros.

These strings are predefined by **—man**:

*R '®', '(Reg)' in **nroff.**

*S Change to default type size.

Request	Cause Break	If no Argument	Explanation
.B *t*	no	*t* = n.t.l.*	Text *t* is bold.
.BI *t*	no	*t* = n.t.l.	Join words of *t* alternating bold and italic.
.BR *t*	no	*t* = n.t.l.	Join words of *t* alternating bold and Roman.
.DT	no	.5i 1i . . .	Restore default tabs.
.HP *i*	yes	*i* = p.i.*	Set prevailing indent to *i*. Begin paragraph with hanging indent.
.I *t*	no	*t* = n.t.l.	Text *t* is italic.
.IB *t*	no	*t* = n.t.l.	Join words of *t* alternating italic and bold.
.IP *x i*	yes	*x* = ""	Same as .TP with tag *x*.
.IR *t*	no	*t* = n.t.l.	Join words of *t* alternating italic and Roman.
.LP	yes	–	Same as .PP.
.PD *d*	no	*d* = .4v	Interparagraph distance is *d*.
.PP	yes	–	Begin paragraph. Set prevailing indent to .5i.
.RE	yes	–	End of relative indent. Set prevailing indent to amount of starting .RS.
.RB *t*	no	*t* = n.t.l.	Join words of *t* alternating Roman and bold.
.RI *t*	no	*t* = n.t.l.	Join words of *t* alternating Roman and italic.

.RS *i*	yes	*i* = p.i.	Start relative indent, move left margin in distance *i*. Set prevailing indent to .5i for nested indents.
.Rv *r t*	yes	*r* = "" *t* = n.t.l	Set Return Value type to *r* and the description text to *t*.
.SH *t*	yes	*t* = n.t.l.	Subhead.
.SM *t*	no	*t* = n.t.l.	Text *t* is small.
.TH *n s x v m*	yes	–	Begin page named *n* of section *s; x* is extra commentary, e.g. 'local', for page foot center; *v* alters page foot left, e.g. '4th Berkeley Distribution'; *m* alters page head center, e.g. 'Brand X Programmer's Manual'. Set prevailing indent and tabs to .5i.
.TP *i*	yes	*i* = p.i.	Set prevailing indent to *i*. Begin indented paragraph with hanging tag given by next text line. If tag doesn't fit, place it on separate line.
.Va *v t*	yes	*v* = "" *t* = n.t.l	Set Variable to *v* and the description text to *t*.

* n.t.l. = next text line; p.i. = prevailing indent

FILES

/usr/lib/tmac/tmac.an

/usr/man/man0/xxx

CAVEATS

Relative indents don't nest.

SEE ALSO

man(1man).

NAME

me — macros for formatting papers

SYNOPSIS

nroff —me [*options*] *filename...*
troff —me [*options*] *filename...*

DESCRIPTION

This package of **nroff** and **troff** macro definitions provides a canned formatting facility for technical papers in various formats. When producing 2-column output on a terminal, filter the output through *col(1)*.

The macro requests are defined below. Many **nroff** and **troff** requests are unsafe in conjunction with this package, however these requests may be used with impunity after the first .pp:

.bp	begin new page
.br	break output line here
.sp n	insert n spacing lines
.ls n	(line spacing) n = 1 single, n = 2 double space
.na	no alignment of right margin
.ce n	center next n lines
.ul n	underline next n lines
.sz +n	add n to point size

Output of the **eqn, neqn, refer,** and *tbl(1)* preprocessors for equations and tables is acceptable as input.

REQUESTS In the following list, "initialization" refers to the first .pp, .lp, .ip, .np, .sh, or .uh macro. This list is incomplete.

Request	Initial Value	Cause Break	Explanation
.(c	–	yes	Begin centered block
.(d	–	no	Begin delayed text
.(f	–	no	Begin footnote
.(l	–	yes	Begin list
.(q	–	yes	Begin major quote
.(x *x*	–	no	Begin indexed item in index *x*
.(z	–	no	Begin floating keep
.)c	–	yes	End centered block
.)d	–	yes	End delayed text
.)f	–	yes	End footnote
.)l	–	yes	End list
.)q	–	yes	End major quote
.)x	–	yes	End index item
.)z	–	yes	End floating keep
.+ + *m H*	–	no	Define paper section. *m* defines the part of the paper, and can be **C** (chapter), **A** (appendix), **P** (preliminary, e.g., abstract, table of contents, etc.), **B** (bibliography), **RC** (chapters renumbered from page one each chapter), or **RA** (appendix renumbered from page one).
.+c *T*	–	yes	Begin chapter (or appendix, etc., as set by .+ +). *T* is the chapter title.

.1c	1	yes	One column format on a new page.
.2c	1	yes	Two column format.
.EN	–	yes	Space after equation produced by **eqn** or **neqn**.
.EQ *x y*	–	yes	Precede equation; break out and add space. Equation number is *y*. The optional argument *x* may be *I* to indent equation (default), *L* to left–adjust the equation, or *C* to center the equation.
.TE	–	yes	End table.
.TH	–	yes	End heading section of table.
.TS *x*	–	yes	Begin table; if *x* is *H* table has repeated heading.
.ac *A N*	–	no	Set up for ACM style output. *A* is the Author's name(s), *N* is the total number of pages. Must be given before the first initialization.
.b *x*	no	no	Print *x* in boldface; if no argument switch to boldface.
.ba +*n*	0	yes	Augments the base indent by *n*. This indent is used to set the indent on regular text (like paragraphs).
.bc	no	yes	Begin new column
.bi *x*	no	no	Print *x* in bold italics (nofill only)
.bx *x*	no	no	Print *x* in a box (nofill only).
.ef *'x'y'z'*	''''	no	Set even footer to *x y z*
.eh *'x'y'z'*	''''	no	Set even header to *x y z*
.fo *'x'y'z'*	''''	no	Set footer to *x y z*
.hx	–	no	Suppress headers and footers on next page.
.he *'x'y'z'*	''''	no	Set header to *x y z*
.hl	–	yes	Draw a horizontal line
.i *x*	no	no	Italicize *x;* if *x* missing, italic text follows.
.ip *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.lp	yes	yes	Start left–blocked paragraph.
.lo	–	no	Read in a file of local macros of the form .*x. Must be given before initialization.
.np	1	yes	Start numbered paragraph.
.of *'x'y'z'*	''''	no	Set odd footer to *x y z*
.oh *'x'y'z'*	''''	no	Set odd header to *x y z*
.pd	–	yes	Print delayed text.
.pp	no	yes	Begin paragraph. First line indented.
.r	yes	no	Roman text follows.
.re	–	no	Reset tabs to default values.
.sc	no	no	Read in a file of special characters and diacritical marks. Must be given before initialization.
.sh *n x*	–	yes	Section head follows, font automatically bold. *n* is level of section, *x* is title of section.
.sk	no	no	Leave the next page blank. Only one page is remembered ahead.
.sz +*n*	10p	no	Augment the point size by *n* points.
.th	no	no	Produce the paper in thesis format. Must be given before initialization.
.tp	no	yes	Begin title page.

.u *x*	–	no	Underline argument (even in **troff**). (Nofill only).
.uh	–	yes	Like .sh but unnumbered.
.xp *x*	–	no	Print index *x*.

FILES

/usr/lib/tmac/tmac.e

*/usr/lib/me/**

SEE ALSO

tbl(1).

NAME

mm — the MM macro package for formatting documents

SYNOPSIS

mm [*options*] [*filenames*]

nroff —mm [*options*] [*filenames*]

nroff —cm [*options*] [*filenames*]

mmt [*options*] [*filenames*]

troff —mm [*options*] [*filenames*]

DESCRIPTION

This package provides a formatting capability for a very wide variety of documents. The manner in which a document is typed in and edited is essentially independent of whether the document is to be eventually formatted at a terminal or is to be phototypeset. See the references below for further details.

The **—mm** option causes **nroff** to use the non–compacted version of the macro package, while the **—cm** option results in the use of the compacted version, thus speeding up the process of loading the macro package.

FILES

/usr/lib/tmac/tmac.m	pointer to the non–compacted version of the package
/usr/lib/macros/mm[nt]	non–compacted version of the package
/usr/lib/macros/cmp.n.[dt].m	compacted version of the package
/usr/lib/macros/ucmp.n.m	initializers for the compacted version of the package

SEE ALSO

mm(1), *mmt*(1), *nroff*(1), *troff*(1).

NAME

ms — macros for formating manuscripts

SYNOPSIS

nroff —ms [options] file ...
troff —ms [options] file ...

or

ms [options] file ...

DESCRIPTION

This package of *nroff* and *troff* macro definitions provides a canned formating facility for technical papers in various formats. When producing 2–column output on a terminal, filter the output through *col(1)*. Output of the *neqn(1)*, *refer(1)*, and *tbl(1)* preprocessors for equations and tables is acceptable as input.

REQUESTS

Following is a complete list of available ms formating requests. See the manual(s) for detailed descriptions of the various requests.

.1C	One column format on a new page.
.2C	Two column format.
.AB	Begin abstract.
.AD *f*	Set right margin adjustment on (*f* = 1 or missing) or off (*f* = 0).
.AE	End abstract.
.AI	Author's institution follows.
.AN *c*	Define auto increment number *c*.
.AU	Author's name follows.
.B *x*	Print *x* in boldface; if no argument switch to boldface.
.BC	Begin new column when in .2C mode.
.BD	Start centered block display which may extend over page boundaries.
.BE	End text to be boxed; print it (also known as .B2).
.BP	Begin a new page.
.BR	Begin a new line ("break" the line).
.BS	Start text to be enclosed in a box (also known as .B1).
.BT	Print page footer at bottom of page. May be redefined.
.BU	Start a bullet item (indented paragraph with bullet label).
.BX *word*	Print *word* in a box.
.CD	Start centered display which may extend over page boundaries.

.CN	Tektronix Labs confidentiality note. May be redefined.
.COL	Pipe output through *col*(1) if necessary. Must be first.
.CS *f*	Enter constant spacing mode if *f* is missing; leave constant spacing mode if *f* is 0. Ignored in *nroff*.
.DA	Place current date at bottom of each page.
.DE	End displayed text.
.DR	This is a draft document.
.DS *x*	Start of displayed text, to appear verbatim line–by–line. *x* = I for indented display (default), *x* = L for left–justified on the page, *x* = C for centered, *x* = B for make left–justified block, then center whole block. Implies .KS.
.EH	End heading.
.FE	End footnote.
.FS *x*	Start footnote. *x* is optional label to be placed to the left of the footnote.
.HL	Draw a horizontal line across the page.
.HS *x y*	Specify heading style. O indicates outline form; I indicates indented numbered sections.
.HY *f*	Set hyphenation on (*f* = 1 or missing) or off (*f* = 0).
.I *x*	Italicize *x*. If no argument switch to italics. Underline in *nroff*.
.ID	Start indented display which may extend over page boundaries.
.IE	End an indented section.
.IOC	IOC style. Must be first. .TO, .FR, .CC, .SU, .DA, .TI, .PL give information for IOC header.
.IP *l x y*	Start indented paragraph, with hanging label *l*. Text indentation is *x* spaces; label is indented *y* spaces.
.IS	Start indented section.
.JU *'l'c'r'*	Justify line, with *l* left–justified, *c* centered, and *r* right–justified.
.KE	End keep. Put preceding text on next page if not enough room.
.KF	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
.KS	Start keeping following text.
.LD	Start left–justified display which may extend over page boundaries.

.LG	Make letters larger. Ignored in *nroff*.
.LP	Start left–blocked paragraph.
.LS *n*	Set line spacing to *n* lines (2 for double–spacing).
.LT	Business letter style. Must be first.
.ND *date*	Use date supplied in place of actual date.
.NE *n*	Need *n* lines on page; page eject if not enough.
.NH *n*	Same as .SH, with section number supplied automatically. Numbers are multilevel, like 1.2.3, where *n* tells what level is wanted (default is 1).
.NL	Make letters normal size.
.P1	Include header at top of page 1 (normally suppressed).
.PC	Print header preceding table of contents. May be redefined.
.PN *n*	Set page number of next page to *n*.
.PP	Start paragraph. First line indented.
.PT	Print page header at top of page. May be redefined.
.PX	Print header preceding index. May be redefined.
.QE	End quoted material.
.QP	Start quoted paragraph (indented and shorter).
.QS	Start quoted material (indented and shorter).
.R	Roman text follows.
.RD *file*	Read input from *file*.
.RE	End relative indent section.
.RP	Released paper style. Must be first.
.RS	Start level of relative indentation. Following .IP's are measured from current indentation.
.SE	End a section of text to be sorted.
.SH	Section head follows; font automatically bold.
.SM	Make letters smaller. Ignored in *nroff*.
.SO	Sort following text.
.SP *n*	Space *n* lines (1 if missing).
.SZ *n*	Set character size. Ignored in *nroff*.
.TA *x...*	Set tabs.
.TC *text*	Place *text* in the table of contents and also include in text.
.TE	End table.

.TH	End heading section of table.
.TL	Title of document follows.
.TM *x*	Technical memo style, with optional number *x*. Must be first.
.TR *x*	Technical report style, with optional number *x*. Must be first.
.TS *x*	Start table; if *x* is H table has repeated heading.
.UL *word*	Underline argument (even in *troff*).
.UX	'UNIX'; first time used, add footnote 'UNIX is a trademark of Bell Laboratories.'
.XN *text*	Add *text* to index without a page number.
.XX *text*	Add *text* to index with current page number.

IN–LINE COMMANDS

\space	Unpaddable Space Character
\e	Echo Backslash Character
\%	Suppress Hyphenation
\F*x*	Switch to Font *x* (Also \f)
\s*n*	Set Character Size to *n* Points
\s±*n*	Increase/Decrease Size by *n* Points
\(*xy*	Special Character *xy*
\o'...'	Overstrike Characters
\"	Ignore Rest of Input Line (For Comments)
*{	Start Superscript
*}	End Superscript
*[	Start Subscript
*]	End Subscript
**x*	Increment and Print Auto Number *x*
\n*x*	Print Auto Number *x* (no incr.)
*(DT	Today's Date
*(DY	Today's Date (Changeable via .ND)
*(DW	Day of the Week
\n(PN	Current Page Number

STRING/NUMBER REGISTERS

.ds LH	Left Portion of Page Header (Initially Null)
.ds CH – \\n(PN –	Center Portion of Page Header
.ds RH	Right Portion of Page Header

.ds LF	Left Portion of Page Footer
.ds CF	Center Portion of Page Footer (*(DY if.DA)
.ds RF	Right Portion of Page Footer
.ds NF R	Normal Text Font
.ds HF B	Heading Font (.SH/.NH)
.ds PD 1v	Paragraph Separation (.PP/.DS/.SP -- 0.5v if **—Tvpr**)
.ds DI Distribution	Default for Missing .TO Argument in IOC
.nr LL 6i	Line Length (6.5i for IOC)
.nr LT 6i	Header/Footer Length (6.5i for IOC)
.nr FL 6i–3n	Footnote Line Length
.nr PO 0	Page Offset (Appropriate Value if **—Tvpr**)
.nr HM 1i	Top Margin (Header in Middle of Margin)
.nr FM 1i	Bottom Margin (Footer in Middle of Margin)
.nr PI 5n	Paragraph (.PP/.IP/.IS) Indent
.nr QI 5n	Quoted Section (.QP/.QS) Indent
.nr NI 4n	Auto Indent for Numbered Sections (.HS I)
.nr PS 10	Character Point Size (Range 6 to about 18)
.nr VS 12	Vertical Spacing (Normally PS+2)
.nr CS 24	Constant Spacing Character Width (.CS)

FILES

/usr/lib/tmac/tmac.s*

SEE ALSO

nroff(1), tbl(1).

NAME

term — conventional names for terminals

DESCRIPTION

Certain commands use these terminal names. They are maintained as part of the shell environment (see *sh(1sh)*, *environ(7)*).

adm3a	Lear Seigler Adm–3a
2621	Hewlett–Packard HP262? series terminals
hp	Hewlett–Packard HP264? series terminals
c100	Human Designed Systems Concept 100
h19	Heathkit H19
mime	Microterm mime in enhanced ACT IV mode
1620	DIABLO 1620 (and others using HyType II)
300	DASI/DTC/GSI 300 (and others using HyType I)
33	TELETYPE® Model 33
37	TELETYPE Model 37
43	TELETYPE Model 43
735	Texas Instruments TI735 (and TI725)
745	Texas Instruments TI745
dumb	terminals with no special features
dialup	a terminal on a phone line with no known characteristics
network	a terminal on a network connection with no known characteristics
4014	Tektronix 4014
vt52	Digital Equipment Corp. VT52

The list goes on and on. Consult */etc/termcap* (see *termcap(5t)*) for an up–to–date and locally correct list.

Commands whose behavior may depend on the terminal either consult TERM in the environment, or accept arguments of the form **—Tterm**, where **term** is one of the names given above.

CAVEATS

The programs that ought to adhere to this nomenclature do so only fitfully.

SEE ALSO

clear(1), ex(1), more(1), sh(1sh), stty(1), nroff, tset(1), ul(1), termcap(3t), termcap(5t), ttytype(5), environ(7).

NAME

arp — address resolution display and control

SYNOPSIS

arp *hostname*
arp –a [*vmunix*] [*kmem*]
arp –d *hostname*
arp –s *hostname IEEE802.3_addr* [*temp*] [*pub*]
arp –f *filename*

DESCRIPTION

The **arp** program displays and modifies the Internet–to–IEEE802.3(Ethernet) address translation tables used by the address resolution protocol (*arp(4n)*).

With no flags, the program displays the current ARP entry for *hostname*. *Hostname* can be specified as either the name(*hostname(1n)*) selected for the workstation or as the internet address(*inet(4n)*).

OPTIONS

—a The program displays all of the current ARP entries by reading the table from the file *kmem* (default */dev/kmem*) based on the kernel file *vmunix* (default */vmunix*).

—d A super–user may delete an entry for the host called *hostname.*

—s Create an ARP entry for the host called *hostname* with the IEEE802.3 address *IEEE802.3_addr* An Example would be:

```
arp -s hosty 8:0:11:0:6:11 [ temp ][ pub ]
```

In this example *hostname* is `hosty` and the *IEEE802.3_addr* is `8:0:11:0:6:11`. The IEEE802.3 address components are hexadecimal and must be seperated by colons.
If *temp* had been specifed the entry would be temporary and would be flushed after 20 minutes. Also if *pub* had been specified the entry will be "published", e.g., this system will respond to ARP requests for *hostname* even though the hostname is not its own.

—f Causes the file *filename* to be read and multiple entries to be set in the ARP tables. Entries in the file should be of the form

```
hostname IEEE802.3_addr [ temp ] [ pub ]
```

with argument meanings as given above.

SEE ALSO

arp(4n), ifconfig(8n).

NAME

bootsrvd — boot server

SYNOPSIS

/etc/bootsrvd [**—f** *configfile*] [**—i** *interval*] [**—l** *logfile*] [**—p** *port*] [**—r** *retries*] [**—t** *timeout*]

DESCRIPTION

Bootsrvd comprises the server side of the server←remote "boot from LAN" function. The server monitors the UDP boot service port (see *services(5n)*) for boot request messages from remote stations and creates a child process to handle each received request. Each child opens its own UDP communications channel to the remote station, then determines if the request is for boot file download or for service port information. If the request is for download of a boot file, the child searches the boot server database for the name of a file associated with the remote station's Internet address, or for a default boot file associated with the identification string included in the remote station's boot request. If an applicable file is found and is available, it is then transferred to the remote station. Conversely, if the request is for service port information, the child determines the port number and protocol for the requested service and returns this information to the remote station.

When **bootsrvd** is invoked, it builds a database of download files from the information contained in the "*/etc/bootsrv.conf*" configuration file. This file specifies host–specific download files for remote stations according to each station's Internet address. Additionally, this file contains the default download file specifications.

The configuration file is checked periodically to determine if the information has changed. A check of the configuration file can be forced by sending a hangup signal (SIGHUP) to the boot server process.

Each download file is assumed to be an *a.out* file executable on the respective remote station. When **bootsrvd** opens this file for download, it constructs a load address map from the information contained in the **exec** header and symbol table of the file; hence, the download file should not be stripped.

The server transfers the file using a message format containing the load address, the byte count, and the data block. For each block of data, the server sends a download message, then waits for the remote station to echo the message. The echoed message is then compared to the transmitted message to verify a successful transfer. A limited number of retries are attempted in the event the remote station does not respond to a message or the echoed message differs from the transmitted message. If a transfer ultimately fails, the server closes the connection and the child process terminates.

The server starts remote execution by transferring a download message with a byte count of —1.

OPTIONS

—f *configfile*
Specifies an alternate configuration file. The default file is "*/etc/bootsrv.conf*".

—i *interval*
Specifies the number of seconds between configuration file checks. If *interval* is *0*, then no periodic checking is performed, and the configuration file is checked only when the boot server process receives a hangup signal. The default interval is 60 seconds.

—l *logfile*
Specifies an alternate log file to receive the error messages generated by **syslog**.

—p *port*
Specifies an alternate boot service port.

—r *retries*
Specifies the maximum number of retries per message. The default is 2.

—t *timeout*
Specifies the maximum number of seconds to wait for a response to each message from the remote station. If *timeout* is *0*, then the boot server process waits forever for a message response. The default timeout is 5 seconds.

EXAMPLES

The following invocation of **bootsrvd** specifies a 5 minute configuration file check interval and a 10 second message timeout value.

```
/etc/bootsrvd -i 300 -t 10
```

FILES

/etc/services	This file contains the definition of the UDP boot service port.
/etc/bootsrv.conf	This is the default boot server configuration file.
/usr/lib/bootsrv	This is the directory in which the boot server builds its database files. The "*boot_conf*" file in this directory is a working copy of the information contained in the configuration file.

DIAGNOSTICS

Bootsrvd prints error messages to the system log file or a specified alternate file via *syslog(3c)*.

RETURN VALUE

[0]	**Bootsrvd** is running.
[USAGE]	Incorrect command line syntax. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

[NP_ERR] An error occurred that was not a system error. Execution terminated.

CAVEATS

Since the boot port is privileged, only the super–user may invoke **bootsrvd**.

Each download file listed in the configuration file should be specified with its full pathname. Additionally, since the load address is derived from the symbol table, the file should not be stripped. Due to functional requirements, the download file should be linked without the *"crt0.o"* startup file. (This implies that the download program will terminate via **return** rather than **exit**.) Further, the download file for the 6130 systems should have a text segment origin of 8000 (hex) or greater due to the storage of critical data in low memory. The following *Makefile* example illustrates the linking procedure:

```
ORIGIN = 8000

progname: $(OBJS)
        $(LD) -o progname -e _main -T $(ORIGIN) $(OBJS) $(LOADLIBES)
```

NOTE: The remote station will repeat the "boot from LAN" procedure if the download program return value is 0.

SEE ALSO

syslog(3c), udp(4n), a.out(5), bootsrv.conf(5n), services(5n).

NAME

catman — format manual pages and build auxiliary files

SYNOPSIS

/etc/catman [**—f** *command*] [**—i**] [**—p**] [**—n**] [**—s**] [**—v**] [**—w**] [*—number*] [*directory ...*]

DESCRIPTION

Catman is used to format manual pages which have been changed and build **whatis** databases and index format tables.

In order to find pages in need of formatting, **catman** compares the modification dates of the files in the subdirectories *man[1-8]* (or only the sections specified) against those in the corresponding *cat* subdirectory (see *man(5man)* for the directory structure requirements). If the *man* file is newer than the *cat* file, or the *cat* file is empty or nonexistent, the file is formatted (see Formatting below).

Usage

There are three ways to use **catman**. The first, and most often used, way is to execute **catman** nightly to format the system manual page files. This can be done by having an entry in */usr/lib/crontab* (see EXAMPLES). In this case, directory names are obtained from the file */usr/lib/man/directories*. If the *actions* field of the entry (see *man(5man)*) contains the character *f*, the manual pages are formatted. If the *actions* field contains an *i*, the pages are formatted and the program **/usr/lib/buildif** is invoked for each page to build and add index format tables (for use by the commands *help(1man)* and *section(1man)*) to the manual page. If the *actions* field contains a *w*, the pages are formatted and the program **/usr/lib/makewhatis** is invoked in order to rebuild the **whatis** database. In these last two cases, the action is only taken if pages were reformatted.

The second use of **catman** is to process personal manual page directories. When the **—p** option is given, the file *$HOME/.manrc* is read to obtain the names of personal manual page directories. Each of these directories is checked for manual pages in need of formatting, and those pages are formatted. In addition, the index format tables and whatis database are rebuilt.

The third use of **catman** is to only reformat certain directories. In this case, the directories are listed on the command line and are formatted just as in the case of personal manual page directories (see EXAMPLES).

Formatting

By default, all manual pages are formatted using the command

```
nroff -man ⟨ source ⟩ formatted
```

(The command line is executed by */bin/sh.)* If the **—f** option is given, the given command replaces the "nroff –man".

If the first 1024 characters of the manual page source file contains the word *$Compile:*, the text following the **:** up to a newline or the sequence **$$** is used in place of **nroff**–man. This command may contain a **%f**, which is replaced by the source file name. In this case, the *< source* portion is not included in the command line. This feature is very useful for manual pages which require preprocessing by commands such as **tbl** and **neqn** and postprocessing by **col**.

Examples of *$Compile* lines are:

```
\" $Compile: tbl ¦ nroff -man ¦ col
\" $Compile: neqn %f ¦ tbl ¦ nroff -man ¦ col $$
```

The Sections File
When all directories have been processed, the set of sections seen is compiled. If personal or command–line specified directories were processed, the list of known sections is printed (this information can be used to update the *$HOME/.manrc* file or system sections file). If the system directories were processed, the data is used to rebuild the file */usr/lib/man/sections*. In this case, the section ordering is preserved. All new sections are placed before the '+' entry for that section number. If there is no '+' entry for the section number, the new section is placed at the end of the file. Sections that no longer exist are deleted. For example, if the sections file contains '2 2x 2+' and manual pages whose sections are '2d' and '2n' are seen and no section '2x' manual pages were seen, the sections file will contain '2 2d 2n 2+'.

It is very important to note that giving sections to format on the command line inhibits the rebuilding of the sections file and causes the sections to be printed.

Directory Creation
If a *cat* directory does not exist, an attempt is made to create it. If the file exists but is not a directory, **catman** will abort.

OPTIONS

—f *command*
Format all files using the given command. This overrides the default use of **nroff –man** and any *$Compile* directives in the source.

—i Don't build index format tables. This overrides the *i* in the *actions* field of entries in */usr/lib/man/directories*.

—p Format "personal" directories found in *$HOME/.manrc*.

—n Print commands to be executed but do not execute them. This results in a list of commands that can be executed, including creation of directories. Double quotes will surround each file name.

—v Versbose. Print commands as they are executing. When executing **makewhatis**, the **—v** is also given so that any errors will be reported.

—w
Don't build **whatis** databases. This overrides the *w* in the *actions* field of entries in */usr/lib/man/directories*.

—number
Format only the given section. There may be more than one of these options. Use of this option with the system directories inhibits rebuilding of the sections file.

EXAMPLES

Catman is usually used to format the system directories. The following entry, when placed in */usr/lib/crontab*, will cause the changed files to be reformatted each night at 2:30am.

```
30 02 * * * /etc/catman -v
```

This invocation will cause sections 1, 2, and 3 of the directories listed in "personal" entries of the file *$HOME/.manrc* to be formatted. The commands executed and known sections list will be printed.

```
/etc/catman -p -v -1 -2 -3
```

The following command will print the commands required to bring the manual pages in */usr/tman* up to date. The known suffix list will not be printed.

```
/etc/catman -n -s /usr/tman
```

The following command will execute commands to bring the manual pages in */usr/man/man1* up to date. The known suffix list will be printed, but */usr/lib/man/sections* will have to be updated by hand.

```
/etc/catman -1 /usr/man
```

FILES

/usr/lib/man/directories	Manual page search directory information.
/usr/lib/man/sections	Known manual page sections list.
$HOME/.manrc	Searched for "personal" manual page directory names.

/usr/lib/makewhatis	Whatis database-building command.
/usr/lib/buildif	Index format file builder.

VARIABLES

PATH	The user's execution path.
HOME	The user's home directory.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.

CAVEATS

Only manual pages with proper suffixes for the section are checked for reformatting. For example, the file */usr/man/man3/at.1* would not be formatted since it has a section specifier of *1* instead of *3*.

Manual pages preprocessed by **tbl** and/or **eqn** should always be postprocessed by **col**.

SEE ALSO

apropos(1man), buildif(1man), col(1), help(1man), more(1), neqn(1), nroff(1), section(1man), tbl(1), whatis(1man), man(5man), manindex(5man), whatis(5man), cron(8).

NAME

chargen — tty test character pattern generator

SYNOPSIS

/etc/tcp_services/chargen

DESCRIPTION

Chargen generates an endless character string useful for debugging and network measurement. The generated pattern is a "barber pole" made up of the 95 printing ASCII characters.

Chargen is run by *tcpd(8n)* when a connection is made on the "ttytst" service specification; see *services(5n)*.

SEE ALSO

tcpd(8n).

NAME

chown — change owner

SYNOPSIS

/etc/chown [**—f**] [**—l**] *owner filename ...*

DESCRIPTION

Chown changes the owner of *filename* to *owner*. The owner may be either a decimal UID or a login name found in the password file.

Only the super-user can change owner, in order to simplify as yet unimplemented accounting procedures.

OPTIONS

—f Force. No error messages about nonexistent files or files that can not have ownership changed.

—l Follow symbolic links. Normally, the ownership of the symbolic link itself is changed. The **—l** option causes **chown** to follow the symbolic links and change the ownership of the file pointed to.

EXAMPLES

The following invocation will change the owner of the file temp to root:

```
/etc/chown root temp
```

FILES

/etc/passwd	the password file

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.

SEE ALSO

chgrp(1), chmod(1), ln(1), chown(2), group(5), passwd(5).

NAME

clri — clear inode

SYNOPSIS

/etc/clri *filesystem i-number ...*

DESCRIPTION

N.B.: Clri has been replaced by *fsck*(8) for normal file system repair work.

Clri writes zeros on the inodes numbered *i-number* on the specified *filesystem*. *Filesystem* is a special file name referring to a device containing a file system. The command should only be used in emergencies and extreme care should be exercised.

Read and write permission is required on *filesystem*. The inode becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an inode which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the inode is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated inode, so the whole cycle is likely to be repeated again and again.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.

CAVEATS

If the file is open, **clri** is likely to be ineffective.

SEE ALSO

fsck(8).

NAME

comply — check files against specification

SYNOPSIS

comply [**—D**][**—R**][**—S**][**—c** *comment_char*][**—d** *field_delim*] [**—e**][**—f**][**—g** *grpfile*][**—h**][**—l**][**—m**][**—p** *passwdfile*] [**—r** *pseudoroot*][**—v**] *specification_file ...*

DESCRIPTION

Comply checks files in the filesystem against a thoses listed in the specification file. One or more specification files can be placed on the command line. Briefly, each line of the specification file is a description about a certain file in the filesystem. The specification contains the information about the name of the file in the filesystem, who owns the file (owner), the group ownership, the file type, the major/minor device numbers if it is a device, the permission mode, the size in bytes, the hard link count, the RCS revision number, the checksum, and the symbolic link target if the file is a symbolic link are checked. See **bom(5)** for details of file format. Whenever a match does not occur an error is printed. If the **—f** option is used, *comply* will attempt to fix the filesystem version of the file to match the specification.

Users may use *comply* to check their file protection modes. Additionally, for system files, *comply* is recommended to be run from the initialization process through the file /etc/rc (see *init*(8) and *rc*(8)), or periodically by *cron*(8).

OPTIONS

—C
Causes checking of checksums on regular files and directories.

—D
Causes checking of major/minor numbers on devices.

—R
Causes checking of RCS identification numbers on regular files.

—S
Causes checking of sizes on regular files and directories.

—c *comment_char*
Changes the comment character interpretation in the specification file to *comment_char* (default is **%**).

—d *field_delim*
Changes the field deliminter character interpretation in the specification file to *field_delim*. (default is **TAB**).

—e Causes *comply* to do only an existence check of the files. It does NOT check file type, owner, links, size, etc..

—f Causes *comply* to try to fix the filesystem according to the specification file (i.e. change owner) if it can. In addition, *comply* will create missing zero length files, missing directories, and symbolic links as necessary.

—g *grpfile*
Use *grpfile* as the group file for checking (default is **/etc/group**).

—h Causes only a help message to be printed. No other action will be taken no matter what other options are specified.

—l Check that the file hard link count is 'greater than or equal' to the one in the specification, rather than exactly 'equal'.

—m
Check that the file mode is 'greater' than the one in the specification, rather than exactly 'equal'.

—p *passwdfile*
Use *passwdfile* as the password file for checking (default is **/etc/passwd**).

—r *pseudoroot*
Prepend *pseudoroot* to each file name being checked. (Default is null).

—v Prints all errors encountered. (verbose). In addition, this prints all comment lines from the beginning of the specification file upto the first non–comment line.

RETURN VALUE

[0]	No errors occurred.
[1]	Errors occurred.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Comply is unable to fix hard link counts, RCS ids, and checksums. Nor is it currently able to create devices.

SEE ALSO

bom(5), cron(8), stat(2).

NAME

conflict — search for alias/password conflicts

SYNOPSIS

/usr/lib/mh/conflict [**—mail** *user name*]

DESCRIPTION

Conflict is a program which checks to see that inconsistencies between the Rand MH alias file (*aliases(5MH)*) and the *passwd(5)* file have not been introduced. In particular, a line in the alias file may be "tom: jones" (because the user "jones" likes to be called Tom), but if "tom" is also a valid user name for someone else, then that user will no longer receive any mail; his mail will be received by "jones" instead!

Conflict also checks for mailboxes in */usr/spool/mail* which do not belong to a valid user. It assumes that no user name will start with '.', and thus ignores files in */usr/spool/mail* which begin with '.'. It also checks for entries in the *group(5)* file which do not belong to a valid user, and for users who do not belong to any group in the group file. This last test is local to Rand, and will not be performed unless the —DRAND flag was set at compile time.

If the *—mail* flag is set, then the results will be sent to the specified *user name*. Otherwise, the results are sent to the standard output.

Conflict should be run under **Cron,** or whenever system accounting takes place.

FILES

/etc/MailAliases

/etc/passwd

/etc/group

*/usr/spool/mail/**

SEE ALSO

group(5), passwd(5).

NAME

cron — clock daemon

SYNOPSIS

/etc/cron

DESCRIPTION

Cron executes commands at specified dates and times according to the instructions in the file */usr/lib/crontab*. Since **cron** never exits, it should only be executed once. This is best done by running **cron** from the initialization process through the file */etc/rc*; see *init(8)*.

Crontab consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify the minute (0–59), hour (0–23), day of the month (1–31), month of the year (1–12), and day of the week (1–7 with 1 = Monday). Each of these patterns may contain a number in the range above; two numbers separated by a minus meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the Shell. The other lines are made available to the command as standard input.

Crontab is examined by **cron** every minute.

EXAMPLES

The following example runs the program **atrun** every 15 minutes.

```
0,15,30,45 * * * * /usr/lib/atrun
```

The next example shows how *cron(8)* can be used to run jobs as super user (root).

```
0 0 * * * su daemon < /usr/local/lib/uucp.daily
```

FILES

usr/lib/crontab

RETURN VALUE

[NO_ERRS] Command completed without error.

SEE ALSO

at(1).

NAME

daemon — daemon process handler

SYNOPSIS

/etc/daemon [—**K***sig*] [—**Y***sig*] [—**k**] [—**r**] [—**s**] [—**v**] [—**w***time*] [—**y**] [*program ...*]

DESCRIPTION

Daemon is used to stop, start, synchronize, and report status of daemon processes. The *program* arguments must be full pathnames of executable programs followed by any arguments. If no program names are specified, the file */etc/daemontab* is read for program names, —**K,** —**Y,** and —**w** options, and commands to execute to obtain program names (see *daemontab(5)*).

If the —**k,** —**K,** or —**r** options are given, the programs are killed. The default kill signal is TERM (15), which is overriden with the —**K** option on the command line or in the **daemontab** file. After the kill signal is sent, **daemon** waits for 5 seconds (or the time specified by the —**w** option) and checks to make sure that the process was killed. Failures are always reported; sucesses only if the —**v** is specified.

If the —**s** or —**r** options are given, the programs that were not running or were successfully killed are executed. The only kind of failure possible is that no process slots are available.

The —**y** and —**Y** options specify that the program is to be "synchronously restarted". This is done by sending the specified signal to the process, which is supposed to tell the process to reread its configuration data. If */etc/daemontab* is used, only entries which are preceded by the —**Y** will be processed. If the —**y** is given, no programs may be specified on the command line, since there is no default synchronization signal. These restrictions exist because not all daemons can synchronize in this way. After the signal is sent to the process, **daemon** waits for the time specified by the —**w** option (or 5 seconds by default) and then checks to see if the process has died. Death of a process in this case is marked as a failure, since the intention was for the process to keep going. The —**y** and -**Y** options may not be used with the —**k,** —**K,** —**s,** and —**r** options.

If the —**v** option is given, all actions taken are reported as they happen. Finally, a table is printed after all other actions are taken. The table looks like the following:

```
Pid  Status Action          Name
---  ------ ------          ----
```

number status action taken program name with arguments

The *Pid* field contains the process ID number of the program. If the program isn't running, this field is blank. The *Status* field contains the same information as the *STAT* field printed by *ps(1)*. If the program isn't running, this field will be *(none)*. The *Action* field contains a description of what action was taken or why action was not taken. The *Name* field contains the name of the program and its arguments.

Only the superuser may specify options other than **—v.** If no options are given, **—v** is set.

OPTIONS

—K*sig*
Kill programs with the given signal number or name. This option overrides all options given in the **daemontab** file. Execute the command "kill –l" for a list of signal names.

—Y*sig*
Send the specified signal to all programs listed on the command line, or to all programs preceded by a **—Y** in the **daemontab** file. The signal given overrides the signals specified in the **daemontab** file. This option can not be given with **—K, —k, —s,** or **—r.**

—k Kill programs with default signal, which is either TERM (15) or the signal specified by the **—K** option in the **daemontab** file.

—r Restart. Equivalent to giving both the **—k** and **—s** options.

—s Start the programs if they are not running or were succesfully killed.

—v Verbose. Print the results of executing commands from the **daemontab** file, actions taken, and a summary table after all actions are taken.

—w*time*
If the processes are being killed (or restarted), wait *time* seconds before checking to make sure the process is dead. If the processes are being synchronized, wait *time* seconds before checking to make sure that the process is still running. The default time is 5 seconds. This option overrides all *—w* options given in the **daemontab** file.

—y Synchronize processes by sending the signal specified in the **daemontab** file. If no **—Y** is given for an entry in the **daemontab** file, the entry is not processed. This option may not be applied to programs listed on the command line, and may not be given with the **—K, —k, —s,** and **—r** options.

EXAMPLES

This example will print a report about all of the programs listed in */etc/daemontab*. No action is taken.

```
/etc/daemon
```

This example will restart all of the programs listed in the **daemontab** file, as well as those named by executing commands listed in the daemontab file, and print a summary table at the end.

```
/etc/daemon -v -r
```

This example will attempt to kill the program **/etc/lookd** that is running with the argument *debug* with the signal *INT*, will wait 5 seconds before checking to see whether the kill succeeded, and will restart the program if it did succeed. No table is printed.

```
/etc/daemon -K INT -s "/etc/lookd debug"
```

FILES

/etc/daemontab	Program names and commands to execute for more names.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Arguments to programs are significant. If the program **/etc/foo** is specified and **/etc/foo -bar** is running, they are not considered to be the same program.

Since there is no way to tell if a daemon that is started actually survived, the success of starting a program is determined by whether or not **daemon** was able to get a new process slot and whether or not the program is executable. To find out if the daemons actually got started, execute **daemon** again without any options except **—v.**

Some daemons take a long time to die after a kill signal is executed in order to clean up and finish what they are doing.

SEE ALSO

kill(1), ps(1), daemontab(5).

NAME

dfformat — format a floppy disk

SYNOPSIS

/etc/dfformat *[interleave] [dev]*

DESCRIPTION

Dfformat formats a two–sided, double density, 48 TPI floppy disk on a Tektronix 6200 Series workstation. The optional arguments specify the sector interleave factor and the formatting device. Absent arguments cause the default equivalents to be assumed. The default drive and interleave factor are, */dev/rdf* and 4, respectively.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Because of hardware differences between Tektronix workstations, the optimum interleave factor on a 6200 Series workstation is not guaranteed to be optimal on any other series of workstation, such as the 6100 Series.

SEE ALSO

df(4).

NAME

dfsd — Distributed File System Daemon

SYNOPSIS

/etc/dfsd [**—p** *port number*]

DESCRIPTION

Dfsd is the daemon for the Distributed File System. It receives system call commands (like open, close, read, write, etc.) from the operating system on a remote host, and executes them on the local system. When **dfsd** receives a request it spawns a child process to handle it. The child checks the access permission file (see *hosts.dfs.access(5n) for the format of that file*) and if the access is permitted, sets up the umask as set on the requesting host, the groupids as permitted for the remote user on the local host (in */etc/group*) and the effective and real userids as received from the remote host. The child then executes the appropriate system call and returns the result to the requesting kernel (which in turn passes the response to the requesting process). The only exception to this approach is for the *execve(2)* system call. In that case the daemon copies the file to be exec'd back to the requesting host. The file is then executed on the requesting host.

Access permission is based on userid, not username, so the userid on the remote system must match the userid of the username in *hosts.dfs.access*.

The daemon is started automatically whenever the workstation is rebooted. Initially the workstation owner will be prompted by *netconfig(8n)* to enable or disable the Distributed File System. If the response is affirmative, **dfsd** will be started by the *rc.net* file whenever the system is rebooted. **Netconfig** records if it should start up **dfsd** in the file */etc/network.conf*.

Dfsd (as well as other network daemons) logs its errors in */usr/adm/syslog*.

OPTIONS

—p *port number*
Normally **dfsd** listens on the port defined in */etc/services*. This default may be overwritten by specifying an alternate port number. However, to change the port number used by the kernel to establish remote connections, the kernel must be patched.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

SEE ALSO

hosts.dfs.access(5n), syslog(8).

NAME

discard — discard all input

SYNOPSIS

/etc/tcp_services/discard

DESCRIPTION

Discard reads its standard input and throws it away. It is used to implement the TCP discard service.

SEE ALSO

tcpd(8n).

NAME

dmesg — collect system diagnostic messages to form error log

SYNOPSIS

/etc/dmesg [**—i**] [**—f** *kernelfile*] [**—c** *corefile*]

DESCRIPTION

Dmesg looks in a system buffer for recently printed diagnostic messages and prints them on the standard output. The messages are those printed by the system if hardware errors occur and occasionally when system tables overflow non-fatally.

OPTIONS

—i incrementally computes the new messages since the last time it was run with the **—i** flag and places these on the standard output. This is typically used with *cron*(8) to produce the error log */usr/adm/messages* by running the command

```
/etc/dmesg —i ›› /usr/adm/messages
```

every 10 minutes.

—f *kernelfile* File used to get a pointer to the system message buffer. Defaults to */dev/cvt*.

—c *corefile* File in which the current system message buffer is found. The pointer obtained from *kernelfile* is used as an offset into *corefile*. Defaults to */dev/kmem*.

FILES

/usr/adm/messages	error log (conventional location)
/usr/adm/msgbuf	scratch file for memory of **—i** option

CAVEATS

The **—i** flag should only be mentioned in *crontab* since the scratch file is global and using it elsewhere may cause messages to be lost from */usr/adm/messages*.

The system error message buffer is of small finite size. As *dmesg* is run only every few minutes, not all error messages are guaranteed to be logged.

Error diagnostics generated immediately before a system crash will never get logged.

SEE ALSO

cron(8).

NAME

dump — generalized dump utility

SYNOPSIS

/etc/dump *key* [*argument ...*] *filesystem*

DESCRIPTION

If a filesystem is specified, **dump** copies to specified media all files changed after a certain date in that *filesystem*. The *key* specifies the date and other options about the dump. *Key* consists of characters from the set **0123456789bfuJsdWnFSX.**

OPTIONS

0—9 This number is the 'dump level'. All files modified since the last date stored in the file */etc/dumpdates* for the same filesystem at lesser levels will be dumped. If no date is determined by the level, the beginning of time is assumed; thus the option **0** causes the entire filesystem to be dumped. Levels 1–9 provide for incremental dumps. For example if a filesystem is dumped with level 4 on Tuesday and level 5 on Wednesday, then Wednesday's dump contains those files on that filesystem which changed after the Tuesday dump.

b Use alternate buffer size. The number must follow key specifications and will be interpreted as number of 1k blocks (the default is 10k). The purpose of this option is to speed dumps to certain media. If the S option is specified, this is automatically set to 128 (i.e., 128k). This should not be used with flexible disk media.

f Place the dump on the next *argument* file or device instead of the default media. Target media can be 9 track tape, cartridge tape or flexible disk. The device can be local or remote (LAN access), where remote is indicated by a "node:" prefix to the pathname. If the device is remote it must be owned by daemon, as root privileges do not extend across the LAN.

u This is necessary for incremental dumps. If the dump completes successfully, write the date of the beginning of the dump on file */etc/dumpdates*. This file records a separate date for each filesystem and each dump level. The format of */etc/dumpdates* is readable by people, consisting of one free format record per line: filesystem name, increment level and *ctime(3c)* format dump date. */etc/dumpdates* may be edited to change any of the fields, if necessary.

s The size of the dump media is specified by the next *argument*. When the specified size is reached, **dump** will wait for media to be changed. The default size is 2000 feet for 9 track tapes 360k for flexible disk and 400 feet for cartridge tape.

d The density of the tape, expressed in BPI, is taken from the next *argument*. This is used in calculating the amount of tape used

per volume. The default is 1600 for 9 track tape, and 8000 for cartridge.

W **Dump** tells the operator what file systems need to be dumped. This information is gleaned from the files */etc/dumpdates* and */etc/fstab*. The **W** option causes **dump** to print out, for each file system in */etc/dumpdates* the most recent dump date and level, and highlights those file systems that should be dumped. If the **W** option is set, all other options and parameters are ignored, and **dump** exits immediately.

w Is like W, but prints only those filesystems which need to be dumped.

F Specifies flexible disk backup media (9–track tape is default).

S Specifies streaming cartridge tape backup media (9–track tape is default).

n Whenever **dump** requires operator attention, notify by means similar to a *wall(1)* all of the operators in the group ''operator''.

X Turn on debugging for remote dump operations (applies to rdump only). This will result in the remote tape handler program, rmt, putting a trace of what it does in /tmp/rmt.log (on the host machine).

Dump requires operator intervention on these conditions: end of media, end of dump, media write error, media open error or disk read error (if there are more than a threshold of 32). In addition to alerting all operators implied by the **key, dump** interacts with the operator on **dump's** control terminal at times when **dump** can no longer proceed, or if something is grossly wrong. All questions **dump** poses *must* be answered by typing *yes* or *no*, appropriately.

Since making a dump involves a lot of time and effort for full dumps, **dump** checkpoints itself at the start of each media volume. If writing that volume fails for some reason, **dump** will, with operator permission, restart itself from the checkpoint after the old tape has been rewound and removed, and a new tape has been mounted.

Dump tells the operator what is going on at periodic intervals, including usually low estimates of the number of blocks to write, the number of volumes it will take, the time to completion, and the time to the volume change. The output is verbose, so that others know that the terminal controlling **dump** is busy, and will be for some time.

EXAMPLES

```
/etc/dump  0fu  /dev/tc /dev/dw10a
```
(dump entire filesystem "/dev/dw10a" to local device)

```
/etc/rdump  0fu  nodename.name:/dev/rmt1  /dev/dw10a
```
(dump entire filesystem to remote device "/dev/rmt1" on host "nodename" under control of userid "name")

`/etc/dump 0fFu /dev/rdf /dev/dw10a`
(dump entire filesystem to local flexible disk)

`/etc/dump 0fbu /dev/rmt 60 /dev/dw10a`
(dump entire filesystem to local 9 track tape with buffering of 60k)

`/etc/dump w`
(ask dump to list filesystems that need to be dumped)

FILES

/dev/dw10a	default filesystem
/dev/tc	default target device (cartridge tape)
/etc/dumpdates	new format dump date record
/etc/fstab	Dump table: file systems and frequency
/etc/group	to find group *operator*

SEE ALSO

fstab(5), restore(8), rrestore(8), rdump(8).

NAME

dumpfs — dump file system information

SYNOPSIS

dumpfs filesystem ¦ device

DESCRIPTION

Dumpfs prints out the super block and cylinder group information for the file system or special device specified. The listing is very long and detailed. This command is useful mostly for finding out certain file system information such as the file system block size and minimum free space percentage.

Dumpfs should be run as super–user; it needs to open the file system for reading.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

SEE ALSO

disktab(5), fs(5), fsck(8), newfs(8), tunefs(8).

NAME

echod — echo input to output

SYNOPSIS

/etc/tcp_services/echod

DESCRIPTION

Echod copies its standard input to standard output. *Cat(1)* can not be used since it either buffers its input, or with the **-u** option, writes one char at a time.

Echod is run by *tcpd(8n)* when a connection is made on the *echo* service specification; see *services(5n)*.

SEE ALSO

tcpd(8n), nettest(8n).

NAME

fsck — file system consistency check and interactive repair

SYNOPSIS

/etc/fsck —p [*filesystem* ...]
/etc/fsck [**—b** *block#*] [**—n**] [**—y**] [*filesystem* ...]

DESCRIPTION

The first form of **fsck** preens a standard set of filesystems or the specified file systems. It is normally used in the script */etc/rc* during automatic reboot. In this case **fsck** reads the table */etc/fstab* to determine which file systems to check. It uses the information there to inspect groups of disks in parallel taking maximum advantage of I/O overlap to check the file systems as quickly as possible. Normally, the root file system will be checked on pass 1, other *root(a* partition) file systems on pass 2, other small file systems on separate passes (e.g., the *d* file systems on pass 3 and the *e* file systems on pass 4), and finally the large user file systems on the last pass (e.g., pass 5). A pass number of 0 in */etc/fstab* causes a disk to not be checked; similarly partitions which are not marked *rw* or *ro* are not checked.

The system takes care that only a restricted class of innocuous inconsistencies can happen unless hardware or software failures intervene. These are limited to the following:

Unreferenced inodes

Link counts in inodes too large

Missing blocks in the free list

Blocks in the free list also in files

Counts in the super–block wrong

These are the only inconsistencies which **fsck** with the **—p** option will correct; if it encounters other inconsistencies, it exits with an abnormal return status and an automatic reboot started by **/etc/rc** will then fail. For each corrected inconsistency one or more lines will be printed identifying the file system on which the correction will take place, and the nature of the correction. After successfully correcting a file system, **fsck** will print the number of files on that file system and the number of used and free blocks.

Without the **—p** option, **fsck** audits and interactively repairs inconsistent conditions for file systems. If the file system is inconsistent the operator is prompted for concurrence before each correction is attempted. It should be noted that a number of the corrective actions which are not fixable under the **—p** option will result in some loss of data. The amount and severity of data lost may be determined from the diagnostic output. The default action for each consistency correction is to wait for the operator to respond *yes* or *no*. If the operator does not have write permission **fsck** will default to a **—n** action.

Fsck has more consistency checks than its predecessors **check, dcheck, fcheck,** and **icheck** combined.

OPTIONS

The following flags are interpreted by **fsck**.

—b Use the **block** specified immediately after the flag as the super block for the file system. Here, *block* means a 512–byte block. Block 32 (in 512–byte blocks) is always an alternate super block.

—n Assume a no response to all questions asked by **fsck**; do not open the file system for writing.

—y Assume a yes response to all questions asked by **fsck**; this should be used with great caution as this is a free license to continue after essentially unlimited trouble has been encountered.

If no *filesystem* argument is given to **fsck** then a default list of file systems is read from the file */etc/fstab*.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one inode or the free list.
2. Blocks claimed by an inode or the free list outside the range of the file system.
3. Incorrect link counts.
4. Size checks:
   Directory size not of proper format.
5. Bad inode format.
6. Blocks not accounted for anywhere.
7. Directory checks:
   File pointing to unallocated inode.
   Inode number out of range.
8. Super Block checks:
   More blocks for inodes than there are in the file system.
9. Bad free block list format.
10. Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the *lost+found* directory. The name assigned is the inode number. The only restriction is that the directory *lost+found* must preexist in the root of the filesystem being checked and must have empty slots in which entries can be made. This is accomplished by making *lost+found*, copying a number of files to the directory, and then removing them (before **fsck** is executed).

Fsck must be run as super–user.

FILES

/etc/fstab	contains default list of file systems to check

DIAGNOSTICS

The diagnostics produced by **fsck** are intended to be self–explanatory.

RETURN VALUE

[0] Everything worked as expected.

[4] **Fsck** ran normally, except that the root file system was modified. **/etc/rc** usually reboots the system if it sees this return value.

[8] A fatal error occurred. The diagnostic message provided will give more information.

[12]
An internal inconsistency was found.

CAVEATS

Inode numbers for . and .. in each directory should be checked for validity.

There should be some way to start a **fsck —p** at pass *n*.

SEE ALSO

fstab(5), fs(5), mkfs(8), newfs(8), reboot(8).

NAME

ftpd — DARPA Internet File Transfer Protocol server

SYNOPSIS

/etc/tcp_services/ftpd [**—d**] [**—l**] [—ttimeout]

DESCRIPTION

Ftpd is the DARPA Internet File Transfer Prototocol server process. The server uses the TCP protocol and is started by *tcpd(8n)* when a connection is made on the port specified in the *ftp* service specification; see *services(5n)*.

The *ftp* server currently supports the following *ftp* requests; case is not distinguished.

Request	Description
ABOR	abort transfer in progress
ACCT	specify account (ignored)
ALLO	allocate storage (vacuously)
APPE	append to a file
CWD	change working directory
DELE	delete a file
HELP	give help information
LIST	give list files in a directory (''ls -lg'')
MODE	specify data transfer *mode*
NLST	give name list of files in directory (''ls'')
NOOP	do nothing
PASS	specify password
PORT	specify data connection port
QUIT	terminate session
RETR	retrieve a file
RNFR	specify rename-from file name
RNTO	specify rename-to file name
STAT	status of transfer, server or file
STOR	store a file
STRU	specify data transfer *structure*
TYPE	specify data transfer *type*
USER	specify user name
XCUP	change to parent of current working directory
XCWD	change working directory
XMKD	make a directory
XPWD	print the current working directory
XRMD	remove a directory

The remaining *ftp* requests specified in Internet RFC 765 are recognized, but not implemented.

A data transfer may be aborted or stated by sending the telnet chars *IAC IP* and out of band data message, followed by the ABOR or STAT command.

Ftpd interprets file names according to the "globbing" conventions used by *csh(1csh)*. This allows users to utilize the metacharacters ***?[]{}~**.

Ftpd authenticates users according to three rules.

1) The user name must be in the password data base, */etc/passwd*, and **not have a null password**. In this case a password must be provided by the client before any file operations may be performed.

2) The user name must not appear in the file */etc/ftpusers*.

3) If the user name is *anonymous* or *ftp*, an anonymous *ftp* account must be present in the password file (user *ftp*). In this case the user is allowed to log in by specifying any password (by convention this is given as the client host's name).

In the last case, **ftpd** takes special measures to restrict the client's access privileges. The server performs a *chroot(2)* command to the home directory of the *ftp* user. In order that system security is not breached, it is recommended that the *ftp* subtree be constructed with care; the following rules are recommended.

~ftp	Make the home directory owned by *ftp* and unwritable by anyone.
~ftp/bin	Make this directory owned by the super-user and unwritable by anyone. The program *ls(1)* must be present to support the list commands. This program should have mode 111.
~ftp/etc	Make this directory owned by the super-user and unwritable by anyone. The files *passwd(5)* and *group(5)* must be present for the **ls** command to work properly. These files should be mode 444.
~ftp/pub	Make this directory mode 777 and owned by *ftp*. Users should then place files which are to be accessible via the anonymous account in this directory.

OPTIONS

—d Each socket created will have debugging turned on (SO_DEBUG). With debugging enabled, the system will trace all TCP packets sent and received on a socket.

—l Each *ftp* session is logged on the standard output. This allows a line of the form `/etc/ftpd -l > /tmp/ftplog` to be used to conveniently maintain a log of *ftp* sessions.

—t*timeout*
Set the inactivity timeout period to *timeout*. By default the *ftp* server will timeout an inactive session after 60 seconds.

RETURN VALUE

[0] **Ftpd** is running.

[1] **Ftpd** is not running.

CAVEATS

The anonymous account is inherently dangerous and should avoided when possible.

The server must run as the super-user to create sockets with privileged port numbers. It maintains an effective user ID of the logged in user, reverting to the super-user only when binding addresses to sockets.

SEE ALSO

ftp(1n), tcpd(8n).

NAME

getty — set terminal mode

SYNOPSIS

/etc/getty [*type*]

DESCRIPTION

Getty is invoked by *init(8)* immediately after a terminal is opened, following the making of a connection. While reading the name **getty** attempts to adapt the system to the speed and type of terminal being used.

Init calls **getty** with *type*, an argument specified by the **ttys** file entry for the terminal line. *Type* can be used to make **getty** treat the line specially. It is used as an index into the *gettytab(5)* database to determine the characteristics of the line. If there is no *type* argument, or there is no table corresponding to *type* in **gettytab**, the *default* table is used. If there is no */etc/gettytab* a set of system defaults is used.

If indicated by the table located, **getty** will clear the terminal screen, print a banner heading, and prompt for a login name. Usually either the banner or the login prompt will include the system hostname. Then the user's name is read, a character at a time. If a null character is received, it is assumed to be the result of the user pushing the "break" ("interrupt") key. The speed is usually then changed and the "login:" is typed again; a second "break" changes the speed again and the "login:" is typed once more. Successive "break" characters cycle through the same standard set of speeds.

The user's name is terminated by a new–line or carriage–return character. The latter results in the system being set to treat carriage returns appropriately (see *tty(4)*).

The user's name is scanned to see if it contains any lower–case alphabetic characters; if not, and if the name is non–empty, the system is told to map any future upper–case characters into the corresponding lower–case characters.

Finally, *login* is called with the user's name as an argument.

Most of the default actions of **getty** can be circumvented, or modified, by a suitable **gettytab** table.

Getty can be set to timeout after some interval, which will cause dial–up lines to hang up if the login name is not entered reasonably quickly.

FILES

/etc/gettytab	data base describing terminal lines
/etc/ttys	terminal initialization data

CAVEATS

Currently, the format of */etc/ttys* limits the permitted table names to a single character.

SEE ALSO

login(1), ioctl(2), tty(4), gettytab(5), ttys(5), init(8).

NAME

halt — stop the processor

SYNOPSIS

/etc/halt [**—n**] [**—q**] [**—y**]

DESCRIPTION

Halt writes out sandbagged information to the disks and then stops the processor. The machine does not reboot, even if the auto–reboot switch is set on the console.

OPTIONS

—n Prevents the sync before stopping.

—q Causes a quick halt, no graceful shutdown is attempted.

—y This option is needed if you are trying to halt the system from a dialup.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.

SEE ALSO

reboot(8), shutdown(8).

NAME

hardlink — check and create hard links to files

SYNOPSIS

hardlink [**—c** *comment_char*][**—f**][**—h**][**—n**][**—r** *pseudoroot*] [**—v**] *specification_file ...*

DESCRIPTION

Hardlink checks files in the filesystem against a thoses listed in the specification file to be sure that the proper files are hard linked together. One or more specification files can be placed on the command line. Briefly, each line of the specification file is a description about a which files in the filesystem should be hard linked together. See **bom(5)** for details of file format. Whenever **hardlink** is unable to link files together, an error is printed.

Hardlink assumes files with non–zero size has having more importance than those that are zero length. In addition, **hardlink** will do a binary file compare to determine if two files are not originally linked together to see if they are the same, then **hardlink** will then remove one and link them together.

OPTIONS

—c *comment_char*
Changes the comment character interpretation in the specification file to *comment_char* (default is **%**).

—f Forcibly makes links to the first file in the specification line. It will automatically remove any existing files except the first listed.

—h Causes only a help message to be printed. No other action will be taken no matter what other options are specified.

—n Prints a shell script to stdout of what **hardlink** would do if it were to do anything. Stdout is suitable to be piped or given to /bin/sh.

—r *pseudoroot*
Append *pseudoroot* to each file name being checked. (Default is null).

—v Prints all errors encountered. (verbose). In addition, this prints all comment lines from the beginning of the specification file upto the first non–comment line.

DIAGNOSTICS

Each error message has a number enclosed in square brackets that indicate the line number of the input record causing the error or message.

RETURN VALUE

[0]	No errors occurred.
[1]	Errors occurred.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.

[P_ERR] A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Hardlink is unable to fix hard link counts, RCS ids, and checksums. Nor is it currently able to create devices.

SEE ALSO

cron(8), hardlink(5), link(2).

NAME

icheck — file system storage consistency check

SYNOPSIS

/etc/icheck [**—b** *numbers*] [**—s**] [*filesystem*]

DESCRIPTION

N.B.: *Icheck* has been replaced by *fsck(8)* for normal consistency checking.

Icheck examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. If *filesystem* is not specified, a set of default file systems is checked. The normal output of **icheck** includes a report of

The total number of files and the numbers of regular, directory, block special and character special files.

The total number of blocks in use and the numbers of single–, double–, and triple–indirect blocks and directory blocks.

The number of free blocks.

The number of blocks missing; i.e. not in any file nor in the free list.

OPTIONS

—b This option is followed by a list of block *numbers*; whenever any of the specified blocks turns up in a file, a diagnostic is produced.

—s Causes **icheck** to ignore the actual free list and reconstruct a new one by rewriting the super–block of the file system. The file system should be dismounted while this is done; if this is not possible (for example if the root file system has to be salvaged) care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, bad in–core copy of the super–block will not continue to be used. Notice also that the words in the super–block which indicate the size of the free list and of the i–list are believed. If the super–block has been curdled these words will have to be patched. The **—s** option causes the normal output reports to be suppressed.

Icheck is faster if the raw version of the special file is used, since it reads the i–list many blocks at a time.

FILES

Default file systems vary with installation.

DIAGNOSTICS

For duplicate blocks and bad blocks (which lie outside the file system) **icheck** announces the difficulty, the i–number, and the kind of block involved. If a read error is encountered, the block number of the bad block is printed and **icheck** considers it to contain 0. 'Bad freeblock' means that a block number outside the available space was encountered in the free list. '*N* dups in free' means that *n* blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.

CAVEATS

Since **icheck** is inherently two–pass in nature, extraneous diagnostics may be produced if applied to active file systems.

It believes even preposterous super–blocks and consequently can get core images.

The system should be fixed so that the reboot after fixing the root file system is not necessary.

SEE ALSO

fs(5), clri(8), fsck(8), ncheck(8).

NAME

ifconfig — configure network interface parameters

SYNOPSIS

/etc/ifconfig *interface* [*address*] [*parameters*]

DESCRIPTION

Ifconfig is used to assign an address to a network interface and/or configure network interface parameters. **Ifconfig** must be used at boot time to define the network address of each interface present on a machine; it may also be used at a later time to redefine an interface's address. The *interface* parameter is a string of the form *name*unit, e.g. *en0*, while the address is either a host name present in the host name data base, *hosts(5n)*, or a DARPA Internet address expressed in the Internet standard "dot notation".

OPTIONS

The following parameters may be set with **ifconfig**:

up Mark an interface *up*.

down
Mark an interface *down*. When an interface is marked *down*, the system will not attempt to transmit messages through that interface.

trailers
Enable the use of a **trailer** link level encapsulation when sending (default). If a network interface supports **trailers**, the system will, when possible, encapsulate outgoing messages in a manner which minimizes the number of memory to memory copy operations performed by the receiver.

—trailers
Disable the use of a "trailer" link level encapsulation.

arp Enable the use of the Address Resolution Protocol in mapping between network level addresses and link level addresses (default). This is currently implemented for mapping between DARPA Internet addreses and 10Mb/s Ethernet addresses.

—arp
Disable the use of the Address Resolution Protocol.

debug
Enable device driver specific debugging output.

—debug
Disable device driver specific debugging output.

Ifconfig displays the current configuration for a network interface when no optional parameters are supplied.

Only the super-user may modify the configuration of a network interface.

DIAGNOSTICS

Messages indicating the specified interface does not exit, the requested address is unknown, the user is not privileged and tried to alter an interface's configuration.

RETURN VALUE

[0] **Ifconfig** was successful.

[1] **Ifconfig** was unsuccessful.

[USAGE]	Incorrect command line syntax. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.

SEE ALSO

intro(4n), netstat(1n), rc(8).

NAME

init — process control initialization

SYNOPSIS

/etc/init

DESCRIPTION

Init is invoked inside UTek as the last step in the boot procedure. It normally then runs the automatic reboot sequence as described in *reboot(8)*, and if this succeeds, begins multi-user operation. If the reboot fails, it commences single-user operation by giving the super-user a shell on the console.

It is possible to pass parameters from the boot program to **init** so that single-user operation is commenced immediately. For the 6200 series, the parameter *S* will cause this. The parameter *M* will cause multi-user operation to begin after reboot, as described above.

When such single user operation is terminated by killing the single-user shell (i.e. by hitting ^D), **init** runs */etc/rc* without the reboot parameter. This command file performs housekeeping operations such as removing temporary files, mounting file systems, and starting daemons.

In multi-user operation, **init's** role is to create a process for each terminal port on which a user may log in. To begin such operations, it reads the file */etc/ttys* and forks several times to create a process for each terminal specified in the file. Each of these processes opens the appropriate terminal for reading and writing. These channels thus receive file descriptors 0, 1 and 2, the standard input and output and the diagnostic output. Opening the terminal will usually involve a delay, since the **open** is not completed until someone is dialed up and carrier established on the channel. If a terminal exists but an error occurs when trying to open the terminal **init** complains by writing a message to the system console; the message is repeated every 10 minutes for each such terminal until the terminal is shut off in */etc/ttys* and **init** is notified (by a *hangup* signal, as described below), or the terminal becomes accessible (init checks again every minute). After an **open** succeeds, */etc/getty* is called with argument as specified by the second character of the **ttys** file line. **Getty** reads the user's name and invokes **login** to log in the user and execute the Shell.

Ultimately the Shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of **init**, which has been waiting for such an event, wakes up and removes the appropriate entry from the file *utmp*, which records current users, and makes an entry in */usr/adm/wtmp*, which maintains a history of logins and logouts. The **wtmp** entry is made only if a user logged in successfully on the line. Then the appropriate terminal is reopened and **getty** is reinvoked.

Init catches the *hangup* signal (signal SIGHUP) and interprets it to mean that the file */etc/ttys* should be read again. The Shell process on each line which used to be active in **ttys** but is no longer there is terminated; a

new process is created for each added line; lines unchanged in the file are undisturbed. Thus it is possible to drop or add phone lines without rebooting the system by changing the **ttys** file and sending a *hangup* signal to the **init** process: use 'kill —HUP 1.'

Init will terminate multi–user operations and resume single–user mode if sent a terminate (TERM) signal, i.e. "kill —TERM 1". If there are processes outstanding which are deadlocked (due to hardware or software failure), **init** will not wait for them all to die (which might take forever), but will time out after 30 seconds and print a warning message.

Init will cease creating new **getty**'s and allow the system to slowly die away, if it is sent a terminal stop (TSTP) signal, i.e. "kill —TSTP 1". A later hangup will resume full multi–user operations, or a terminate will initiate a single user shell. This hook is used by *reboot(8)* and *halt(8)*.

FILES

/dev/console

*/dev/tty**

/etc/utmp

/usr/adm/wtmp

/etc/ttys

/etc/rc

DIAGNOSTICS

init: tty: cannot open.
A terminal which is turned on in the *rc* file cannot be opened, usually because the requisite lines are either not configured into the system or the associated device was not attached during boot–time system configuration.

WARNING: Something is hung (won't die); ps -axl advised.
A process is hung and could not be killed when the system was shutting down. This is usually caused by a process which is stuck in a device driver due to a persistent device error condition.

SEE ALSO

kill(1), login(1), sh(1sh), ttys(5), crash(8), getty(8), halt(8), rc(8), reboot(8), shutdown(8).

NAME

install — Install files from distribution media

SYNOPSIS

/usr/lib/install [**—tar**] [*source* [*text*]]

DESCRIPTION

Install reads files from *source* and copies them to their named destination files. The destination for each file is determined from the name as it is on the *source* relative to the current working directory.

Source defaults to **/dev/rdf** but may be any file. If *source* is **flexible disk, diskette** or **/dev/rdf**, *install(8)* will load the files from the diskette drive. If it is **tape, /dev/rmt** or **/dev/nrmt** then *install(8)* will load the files from the 9-track tape drive. If it is **cart, cartridge, /dev/tc** or **/dev/ntc** the files will be read from the cartridge tape unit. Any other *source* is assumed to be a file on the system.

Text defaults to **diskette** and is used in the *cpio(1)* prompt if *source* is distributed on more than one diskette, tape or cartridge.

Files on *source.* are assumed to be in *cpio(1)* format. If the **—tar** option is given, then the files are assumed to be in *tar(1)* format and *tar(1)* will be used in lieu of *cpio(1).*

Install is usually called by a shell script called *INSTALL* found on *source* by running the software installation option of */etc/sysadmin(8).* *Sysadmin(8)* will extract the file *INSTALL* from *source* and execute it.

After the files are read from *source,* if a file called *INSTALL.READ* is found in the current directory, it will be copied to the screen via *more(1).* *INSTALL.READ* is typically used to describe what has been installed and/or to give further instructions about finishing the installation of a particular package.

If a file called *INSTALL.EXEC* is found in the *source* , then it will be executed after the files are read from the *source. INSTALL.EXEC* is typically used to create file links, to move the files found on *source* to a different directory and/or to verify that required files are present, to cause **makedev(8)** to create special entries in /dev or any other tasks required by the software being installed.

OPTIONS

—tar Use *tar(1)* instead of *cpio(1)* to extract files from *source.*

EXAMPLES

A typical *INSTALL* script on *source* could be —

```
#!/bin/sh
cd /
/usr/lib/install cartridge "tape cartridge"
```

FILES

INSTALL.READ	Information to be read before installing files.
INSTALL.EXEC	Optional program on *source* to be executed after files have been extracted from *source.*

SEE ALSO

cpio(1), more(1), tar(1).

NAME

lpserver, plpserver, rawlpserver — line printer servers for MDQS

SYNOPSIS

lpserver [**—b baud**] [**—c**] [**—f flagging**] [**—h size**] [**—p parity**] [**—C columns**] [**—H headers**] [**—T trailers**]

DESCRIPTION

Lpserver, **plpserver** and **rawlpserver** are all line printer severs for **MDQS**.

Lpserver is the general line printer server. This server optimizes for overstriking and converts all control characters to their ^⟨*letter*⟩ patterns. Since this server catches all control characters, it cannot be used to control printers with escape sequences.

Plpserver is designed to be used with Printronix P150/P300/P600 printers. This server knows about these printers' special modes, such as **Plotmode**.

Rawlpserver is used when you want the data to go out the port completely unaltered. This server is especially good for outputting graphical data and escape sequences to control printer modes.

OPTIONS

—b baud
Sets the baud rate for the tty port. The default baud rate is 9600.

—c Sets the tty port in **CRMOD** which causes all **LF** characters to be output as **CR–LF**. This option implies a parity of **ODD** unless the parity is explicitly set. If the parity is explicitly set to **NONE**, **ODD** parity is still produced with the **–c** option.

—f flagging
Sets the flagging method used by the tty driver. The two values for flagging are **HW** and **SW** representing hardware and software respectively. Software flagging is the default.

—h size
Sets the amount of information presented on the banner page of the printout. The values are **LARGE**, **SMALL** and **NONE**. The default is **LARGE**.

—p parity
Sets the parity to be used by the tty driver. The values are **EVEN**, **ODD**, **SPACE** and **NONE**. The value of **NONE** allows 8–bit data transmission. **NONE** in conjunction with **rawlpserver** is useful in outputting graphical data or special control commands t[printers. The default setting for parity is **NONE**.

—C columns
Sets the maximum number of columns to be printed on for the banner page. This is particularly useful when the paper in the printer is only 80 columns wide and you do not want filenames on the banner page to print past the end of the paper. The default is 132.

—H headers
Sets the number of banner pages to print at the start of each file. The default is 1.

—T trailers
Sets the number of trailing banner pages to print at the end of each request. The default is 0.

EXAMPLES

The following example is for a Printronix printer running at 2400 baud with one trailing banner page.

```
/usr/lib/mdqs/plpserver -b 2400 -T 1
```

The following example is for a Centronix printer using hardware flagging.

```
/usr/lib/mdqs/lpserver -f HW
```

The following example is for a color hardcopy unit with a small banner page.

```
/usr/lib/mdqs/rawlpserver -h SMALL
```

DIAGNOSTICS

Diagnostics are passed back to the **MDQS** daemon and reported in the **MDQS** console log specified in the **qconf** file.

CAVEATS

These commands are **NEVER** called directly by the user. These commands are specified in the file **/etc/qconf** and are called by the **MDQS** daemon.

SEE ALSO

lpr(1mdqs), qconf(5mdqs), mdqsd(8mdqs), sysadmin(8), tty(4).

NAME

makedev — make system special files

SYNOPSIS

/dev/MAKEDEV package ¦ device [**-v**]

DESCRIPTION

MAKEDEV is a shell script normally used to install special files. It resides in the */dev* directory, as this is the normal location of special files.

Note: **MAKEDEV** should be run in the directory in which the devices are to be created (usually */dev*). Also, use either the package or the device argument, but not both.

Package is an assorted collection of devices; see below. *Device* is of the form *device-name?* where *device-name* is one of the supported devices listed in section 4 of this manual and "?" is a logical unit number. See the *6130 System Administration* manual for an explanation of the valid unit numbers associated with each device name.

Packages:

std	Create the *standard* devices for the system; e.g. */dev/console*, */dev/mem*, terminals, pseudo–terminals, disks.
local	Create those devices specific to the local site. This request causes the shell file */dev/MAKEDEV.local*, if it exists, to be executed. Site specific commands, such as those used to setup dialup lines as *ttyd?* should be included in this file.

Since all devices are created using *mknod(8)*, this shell script is useful only to the super–user.

OPTIONS

-v verbose

DIAGNOSTICS

Either self–explanatory, or generated by one of the programs called from the script. Use *sh -x MAKEDEV* in case of trouble.

SEE ALSO

intro(4n), config(8), mknod(8).

NAME

mdqsd — MDQS queue scheduling daemon

SYNOPSIS

/etc/mdqsd

DESCRIPTION

The daemon is started upon system boot. It can be stopped and restarted via the */etc/daemon* program. If the daemon is stopped, it will allow active requests to complete. Requests in the queue when the MDQS daemon is stopped will be requeued upon restart.

The daemon reads the file */etc/qconf* to determine the current configuration of the queueing system. The daemon will write status files in the directory */usr/spool/q/lock/home/adm* which are read by the status program, **qstat**.

EXAMPLES

To restart **mdqsd** :

```
/etc/daemon -k /etc/mdqsd
qstat   [wait until active jobs finish - says not running]
/etc/daemon -s /etc/mdqsd
```

FILES

/etc/qconf	— Queuing system configuration file
/etc/rc	— System boot commands
/etc/rc.mdqs	— MDQS system boot commands
/usr/spool/q	— Top of the queuing directories

CAVEATS

Mdqsd should only be killed with the TERM signal. See *daemon(8)*. **Mdqsd** should be restarted, using *daemon(8)*.

SEE ALSO

lpr(1mdqs), qstat(1mdqs), qconf(5mdqs), daemon(8), qdev(8mdqs).

NAME

mkfs — construct a file system

SYNOPSIS

/etc/mkfs *special size* [**nsect** [**ntrack** [**blksize** [**fragsize** [**ncpg** [**minfree** [**rps** [**nbpi**]]]]]]]]

DESCRIPTION

N.B.: file systems are normally created with the *newfs*(8) command.

Mkfs constructs a file system by writing on the special file *special.* The numeric *size* specifies the number of sectors in the file system. *Mkfs* builds a file system with a root directory and a *lost+found* directory. (see *fsck(8)*) The number of inodes is calculated as a function of the file system size. No boot program is initialized by **mkfs** (see *newfs(8)*).

Mkfs must be run as super–user, since it must open *special* for reading.

OPTIONS

The optional arguments allow fine tune control over the parameters of the file system.

nsect
Specifies the number of sectors per track on the disk.

ntrack
Specifies the number of tracks per cylinder on the disk.

blksize
Gives the primary block size for files on the file system. It must be a power of two, currently selected from 4096 or 8192.

fragsize
Gives the fragment size for files on the file system. The **fragsize** represents the smallest amount of disk space that will be allocated to a file. It must be a power of two currently selected from the range 512 to 8192.

ncpg
Specifies the number of disk cylinders per cylinder group. This number must be in the range 1 to 32.

minfree
Specifies the minimum percentage of free disk space allowed. Once the file system capacity reaches this threshold, only the super–user is allowed to allocate disk blocks. The default value is 10%.

rps If a disk does not revolve at 60 revolutions per second, the **rps** parameter may be specified.

nbpi
Specifies the number of inode blocks per cylinder group.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.

[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

There is no way to specify bad blocks.

SEE ALSO

fs(5), dir(5), fsck(8), newfs(8), tunefs(8).

NAME

mklost + found — make a lost + found directory for fsck

SYNOPSIS

/etc/mklost+found

DESCRIPTION

A directory *lost + found* is created in the current directory and a number of empty files are created therein and then removed so that there will be empty slots for *fsck(8)* to use. This command should not normally be needed since *mkfs(8)* automatically creates the *lost + found* directory when a new file system is created.

SEE ALSO

fsck(8), mkfs(8).

NAME

mknod — build special file

SYNOPSIS

/etc/mknod *name* [**b** ¦ **c**] *major minor*

DESCRIPTION

Mknod makes a special file. The first argument is the *name* of the entry. The second is *b* if the special file is block-type (disks, tape) or *c* if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g. unit, drive, or line number).

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

SEE ALSO

mknod(2), makedev(8).

NAME

mount, umount — mount and dismount file system

SYNOPSIS

/etc/mount [**—f**] [**—r**] [**—v**] [*special*] *name*

/etc/mount —a

/etc/umount [**—v**] *special*

/etc/umount —a

DESCRIPTION

Mount announces to the system that a removable file system is present on the device *special*. The file *name* must exist already; it must be a directory (unless the root of the mounted file system is not a directory). It becomes the name of the newly mounted root.

If only *name* is given without *special*, *name* must be an entry in the file */etc/fstab* (see *fstab(5)*).

Umount announces to the system that the removable file system previously mounted on device *special* is to be removed.

These commands maintain and update a table of mounted devices in */etc/mtab*. If invoked without an argument, **mount** prints the table.

Mount and **umount** must be run by the super–user.

OPTIONS

—a All of the file systems described in */etc/fstab* are attempted to be mounted or unmounted. In this case, *special* and *name* are taken from */etc/fstab*. The *special* file name from */etc/fstab* is the block special name.

—f If invoked with this option, **mount** will not actually mount any file systems, but */etc/mtab* will be updated as if it had.

—r Indicates to **mount** that the file system is to be mounted read–only.

—v This option will cause **mount** or **umount** to print its actions as it executes.

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.

FILES

/etc/mtab	mount table
/etc/fstab	file system table

DIAGNOSTICS

Not owner
The caller is not the super–user.

Permission denied
The file */etc/mtab* could not be updated.

RETURN VALUE

For both **mount** and **umount**:

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

For *umount*:

[NP_WARN]	An error warranting a warning message occurred. Execution continues.

CAVEATS

Mounting file systems full of garbage will crash the system.

Mounting a root directory on a non–directory makes some apparently good pathnames invalid.

SEE ALSO

mount(2), fstab(5), mtab(5).

NAME

namedbg — nameserver debugger

SYNOPSIS

/etc/namedbg [**—d**] [**—f** *hostname*] [**—a** *address*]

DESCRIPTION

Namedbg is a program for testing the *nameserver(8n)*. The debugger allows testing, inspecting or modifying the hostname tables inside the nameserver. It communicates with the **nameserver** over a *UTek domain* socket. **Namedbg** has two operational modes, interactive and non–interactive. The latter is initiated by specifying a command line argument to namedbg . For interactive mode the following commands are supported:

COMMANDS

help
Give short help list

quit
Leave the debugger.

dump
Print the nameserver's database with use counts. The *B* flag means this entry will be broadcast to other hosts which ask about it. The *P* flag means this entry is permanent.

add*host addr*
Add a new host entry (see CAVEATS).

delete*name*
Delete the all entries for *name*.

find*name*
Find the entry a hostname.

OPTIONS

—d Dump the entire nameserver database.

–f hostname
Call upon the nameserver to find the address of *hostname*.

–a address
Call upon the nameserver to find the host name corresponding to *address*. The *address* is specified in internet format, each byte converted to decimal and separated by '.' (e.g. 6.128.0.3).

DIAGNOSTICS

Most command errors generate the simple error message: *syntax error*. Type the command **help** for a list of commands.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[1]	Some unrecoverable error.

CAVEATS

Changes to the hostname database are not permanent, change */etc/hosts* if the change is permanent.

SEE ALSO

nameserver(8n).

NAME

nameserver — host name server daemon

SYNOPSIS

/etc/nameserver [—**d**] [—**l**localsocket] [—**r**maxretries] [—**t**timeout] [—**p**port#]

DESCRIPTION

Nameserver is the distributed server which maintains the host name and address database used by the *gethostbyname(3n)* and *gethostbyaddr(3n)* subroutines. Its operates on both a local Unix domain and a broadcast socket to the network.

The **Nameserver** broadcasts and receives on a Internet datagram socket. When initialized, it broadcasts the address and name of all interfaces on the host machine. The names are combinations of the hostname (returned by *gethostname(2)*) and the device name. For example:

Address	**NameAliases**
5.128.12.52 myname	myname0 mynamei10
6.128.12.65	mynamemyname1 mynamei11
7.0.0.100	myname myname2 mynameec0

The file */etc/hosts* is then read to create a list of hosts which are *not* running the nameserver. Duplicates in this file are ignored.

Nameserver is a transaction oriented server; it handles transactions sent to a Unix domain stream socket */tmp/name_socket*. Normally, the user should use the functions *gethostbyname(3n)* and *gethostbyaddr(3n)*, instead of communicating with the socket directly.

If a request is found in the internal database, the answer is sent right away. If not, then the request is broadcast to all networks to see if some other host knows the answer. If no answer is received in 3 seconds (this value can be changed with *-t* flag) then an error answer is sent.

The **nameserver** also answers requests from other hosts, and forwards requests on to other networks.

TESTING

If the **nameserver** appears to have problems it may be tested with the command *namedbg(8n)*.

OPTIONS

—**d** Debugging, don't fork off a daemon. Use */tmp/name_debug* for the local socket, and port 1163 for the remote socket.

—**l***localfile*
The Unix domain socket *localfile* is used instead of */etc/name_socket*.

—**t***secs*
Set timeout for retrying remote requests [default 3 secs].

—**r***n*
Make the broadcast request *n* times after failing before before returning "host unknown" [default 3 times].

—p*port*
Use Internet port number *port* instead of the entry *tekname* in */etc/services*.

PROTOCOL

The **nameserver** transmits and receives messages on the broadcast port in the *tekname* service specification, see *services(5n)*.

The local messages sent and received, are of the form (described in */usr/include/sys/nameserver.h)*:

```
#define MAXALIASES      7 /* max aliases for host */
#define NS_VERSION      3 /* version */

#define NSR_ERROR       0 /* type field values */
#define NSR_ANSWER      1 /* answer to a request */
#define NSR_GETNAME     2 /* Gethostbyname() */
#define NSR_GETADDR     3 /* Gethostbyaddr() */
#define NSR_STATUS      4 /* Dump hostname/addresss list */
#define NSR_DELNAME     5 /* delete all entries for name */
#define NSR_DELADDR     6 /* delete all entries for address */

struct ns_req {
        u_short nr_vers;/* version of nameserver*/
        u_short nr_type;/* type of request */
        u_long  nr_addr;/* address of host */
        char    nr_host[MAXHOSTNAMESIZE];
        char    nr_aliases[MAXALIASES][MAXHOSTNAMESIZE];
};
```

All fields are converted to network byte order prior to transmission.

The broadcast request/answer formats are larger since they include information on networks visited and a request ID.

```
#define MAXNETS 256

struct  nb_broad {
        struct ns_req                   nb_req;/* request */
        u_long      nb_reqid;           /* unique request id */
        u_long      nb_from;            /* originally from */
        u_long      nb_nets[MAXNETS];   /* nets visited */
};
```

When forwarding a request to another network, a host extends the *nb_nets* field to include all networks it is forwarding to. This allows a host to hosts on other networks (via a gateway). A request is never sent to a network already listed in the *nb_nets* list. The datagram can be shorter than the full structure size (answers have only request and id).

If no answer is received in 3 secs (set by **-t** option) then the request is resent. If after 3 tries (set by **-r** option) then a NSR_ERROR is returned.

FILES

/tmp/name_socket	local communication to nameserver
/etc/hosts	hosts **not** running the *nameserver.*

DIAGNOSTICS

hostname not set!!
The hostname was not set, this is usually done by *netconfig(8n)*, or *hostname(1n)*.

Invalid hostname
Only the following characters are allowed in a hostname: letters (upper or lower), digits, underline _, or minus sign -.

tekname/udp not in /etc/services
The Internet datagram port number used by the nameserver is not in the system file */etc/services*.

Nameserver prints error messages via *syslog(3c)* to the system log files, after it has disassociated at start up.

RETURN VALUE

[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.

CAVEATS

Forces the mapping from Internet address to hostname to be unique; only one */etc/hosts* entry can be used for a given address (second one is ignored).

Currently handles only Internet address.

Should be able to talk to Internet nameserver if on Arpanet.

The nameserver may cause *Host Unknown* messages when in fact the host is just down.

SEE ALSO

gethostbyaddr(3n), gethostbyname(3n), gethostent(3n), namedbg(8n).

NAME

ncheck — generate names from i–numbers

SYNOPSIS

/etc/ncheck [**—a**] [**—i** *i-number* ...] [**—s**] [*filesystem*]

DESCRIPTION

N.B.: For most normal file system maintenance, the function of **ncheck** is subsumed by *fsck(8)*.

Ncheck with no argument generates a pathname vs. i–number list of all files on a set of default file systems. Names of directory files are followed by */*..

OPTIONS

—a allows printing of the names . and .. , which are ordinarily suppressed.

—i *i-number*
reduces the report to only those files whose *i-numbers* follow.

—s reduces the report to special files and files with set–user–ID mode; it is intended to discover concealed violations of security policy.

A file system *filesystem* may be specified.

DIAGNOSTICS

When the file system structure is improper, '??' denotes the 'parent' of a parentless file and a pathname beginning with '...' denotes a loop.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.

SEE ALSO

sort(1), fsck(8), icheck(8).

NAME

netconfig — configure workstation for network operation

SYNOPSIS

/etc/netconfig [**—q**] [**—h** *hostname*] [**—i** *interfacename* **—a** *address*] [**—e** *net* ¦ *dfs*] [**—d** *net* ¦ *dfs*] [**—P**]

DESCRIPTION

Netconfig is used to configure a workstation for operation on a local area network. **Netconfig** allows the user to change the following network attributes on the workstation:

Hostname
Host ID
Distributed File System (enabled or disabled)
Standard Network Utilities (enabled or disabled)
Internet address for each network interface.

After changing any of these attributes the workstation must be rebooted. This is so the *nameserver(8n)* daemon is restarted with the proper new hostname and addresses, otherwise the network utilities will not work correctly. **Netconfig** writes these changes into the *network.conf(5n)* file. When rebooted *rc(8)* is executed which looks at the *network.conf(5n)* file to set up the host ID and hostname, and to decide which daemons to start up.

Typically **netconfig** is invoked with no options. The program will then prompt the user with questions to set the attributes mentioned above. When it asks for a hostname, the user should enter no more than 32 characters taken from the following set:

a-z, A-Z, 0-9, -, _

The first character must not be a number. Remember that the name assigned to your workstation should be unique throughout the network.

Netconfig then asks if the Distributed File System (DFS) should be enabled. The users response will be recorded in *network.conf(5)* as the string *"dfs_enabled"* or *"dfs_disabled"*. Enabling DFS means that the DFS daemon *(dfsd(8n))* will be started on subsequent rebootings.

The first time **netconfig** is run on the workstation, the Internet address will not be set (actually it is set to the invalid address 0.0.0.0). The user will then be prompted to supply the network number of the local area network to which the workstation is attached. There are three classes of networks, A, B, and C. Since an Internet address has a network component and a host component the difference between the classes is where in the 4 byte Internet address is drawn the boundary between the two components. Class C addresses allow 256 hosts on the network, Class B addresses allow 65536 hosts on a network and Class A allow a few million.

When **netconfig** prompts for a network number the user selects which class to use by entering the data as follows:

User enters:	Class assumed:	Where x,y, and z =
x	A	0 < x < 128
x.y	B	127 < x < 192, 0 <= y < 256
x.y.z	C	191 < x < 224, 0 <= y < 256, 0 <= z < 256

In the above, *x, y,* and *z* are decimal integers.

Once the network number has been entered (and assuming that an Internet address has not already been assigned to this workstation) **netconfig** will *suggest* an Internet address. This suggested address is based on the network number already supplied plus a host number derived from the Ethernet address. Since the Ethernet address is guaranteed to be unique, and the host number of the Internet address must be unique on the local network, by basing the Internet address on the Ethernet address we increase the likelihood that the suggested address is unique. If you have selected class A or B addressing, and have only 6130 workstations on your network, then the user can feel confident that the suggested address is unique. If other vendors equipment also appears on the network then before using the suggested Internet address, verify that no other equipment uses that address.

If the suggested address is not appropriate, enter the host number component of the Internet address as follows:

Class	User enters:	Where x,y, and z =
A	x.y.z	0 <= x,y,z < 256
B	x.y	0 <= x,y < 256
C	x	0 <= x < 256

In the above, x, y, and z are decimal integers.

Next **netconfig** asks whether to enable the regular network daemons. These daemons include those that handle remote logins *(rlogind(8n))* and remote command execution *(rshd(8n))*. See the file */etc/rc.net* for what daemons will be started. Also see *tcp_servers(5n), udpd(8n), and tcpd(8n).*

If the DFS or regular network utilities are enabled and any of the interfaces have not yet been set, **netconfig** will prompt for the Internet address for each network interface.

OPTIONS

Typically the user would invoke **netconfig** without any switches or with the **—P** switch. The full list of capabilities follows.

—a address
Set the internet address from the command line. This option must be used in conjunction with the —**i** option. No prompting provided unless the "address" portion is omitted. Then **netconfig** will prompt for the address of each interface on the workstation.

—d dfs
Netconfig writes the *"dfs_disabled"* string into the *network.conf(5n)* file. No prompting occurs.

—d net
Netconfig writes the *"net_disabled"* string into the *network.conf(5n)* file. No prompting occurs.

—e dfs
This option causes the *"dfs_enabled"* string to be written into the *network.conf(5n)* file.

—e net
This option causes the *"net_enabled"* string to be written into the *network.conf(5n)* file. Prompting may occur if the internet address is not set for the interface(s).

—h hostname
Set the *hostname*; no prompting is provided.

—i interfacename
Indicates the interface name to act upon using the —**a** option.

—**q** This option is for when **netconfig** is invoked from *rc(8)* at boot time. It causes **netconfig** to prompt only for attributes for which we have no known previous value.

—P
Print out the internet and ethernet address for each interface on the workstation.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[0]	Indicates to *rc.net(5n)* to not enable networking or the Distributed File System.
[1]	Indicates to *rc.net(5n)* to enable networking but not the Distributed File System.
[2]	Indicates to *rc.net(5n)* to enable the Distributed File System, but not networking.

[3] Indicates to *rc.net(5n)* to enable both the Distributed File System and networking.

CAVEATS

DO NOT CHANGE THE INTERNET ADDRESS TOO OFTEN!!
Netconfig stores the internet address in a non-volatile random access memory in 6000 series products. This memory will fail after about 500-1000 writes. The memory will not be written unless a **y** is the response to the confirmation question to change the internet address.

Input validation for command line invocation is minimal.

SEE ALSO

hostid(1n), *hostname(1n)*, *gethostname(2)*, *inet(3n)*, *hosts(5n)*, *network.conf(5n)*.

NAME

nettest — network diagnostic test utility

SYNOPSIS

/etc/nettest [**—p** *proto*] [**—r** *node* [*paddr*]] [**—t** *interface*] [*count*]

DESCRIPTION

The **nettest** command provides the user with the capability to conduct a loopback test of the local host's network functions and to perform a time domain reflectometry (TDR) check of a network medium.

The loopback test allows the user to verify the operation of the networking software and hardware. Also, the user can test the data link between the local host and a selected remote host. Normally, this test sends its test data to an "echo server" via the datagram (UDP) protocol and waits for a reply. The data, if returned, are compared to the original data to determine if any transmission errors occurred.

Alternatively, the user may elect to use the stream (TCP) or control message (ICMP) protocols for data transmission. The ICMP protocol handles the echoing of data directly, so no echo server is required in this case.

Faults in a physical medium may be located through use of the LAN chip set's inherent TDR capability. Time domain reflectometry is a technique for locating discontinuities in a transmission line by injecting a signal into the cable and measuring the time interval between the incident signal and any reflection of that signal caused by an open or short in the transmission cable. In the LAN environment, signal reflections result in collisions, and since the point of the open or short is fixed, the time to collision is constant. Thus, it is possible to gauge the approximate distance to the cable fault by asserting the carrier signal and measuring the time until a collision occurs. In the case of the LAN chip set in use, the approximation is accurate to within 11.7 meters ($\simeq$ 38.4 feet), depending upon the LAN configuration in use.

If an iteration *count* is specified, **nettest** displays a "+" for each successful packet loopback, a "?" for each missing packet, and a "—" for each incorrect packet (e.g., CRC error, alignment error, or bad comparison). If the iteration count is specified for a TDR test, **nettest** reports the total number and type of each fault which may have occurred.

OPTIONS

—p *proto*

Use protocol *proto* when performing the loopback test. *Proto* may be TCP, UDP (default), or ICMP.

—r *node*

Perform loopback test via the specified remote *node*. *Node* may be an explicit host name as specified in the */etc/hosts* database, or it may be an Internet address written in "." notation.

—r *node paddr*

Perform loopback test via the specified remote *node* at physical address *paddr*. The physical address is written as six hex bytes separated by colons (e.g., 08:00:11:00:8c:22). Typically, this form is used to direct the loopback to a newly–installed or otherwise unknown node, in which case *node* is specified as an Internet address in order to update the information in the ARP tables.

—t *interface*

Perform a TDR test of the network physical medium via the network interface named *interface*. (Interface names and network numbers may be found by using the *netstat* utility.)

count

Perform the specified test *count* times.

EXAMPLES

nettest –t lna0

Perform a TDR test of the network physical medium attached to network interface "*lna0*".

nettest –p icmp –r 8.10.21.234 00:01:02:03:04:05

Perform a loopback test via the remote host at physical address "*0 1 2 3 4 5*" using the Internet Control Message Protocol (ICMP).

FILES

/etc/hosts	Data base for host names and addresses.

DIAGNOSTICS

Nettest displays error messages describing any system or data errors which might occur.

RETURN VALUE

[0]	The test passed.
[1]	The test failed.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Use of the UDP or TCP protocols during the loopback test requires that an echo server (see *udpd(8n)* and *tcpd(8n)*) be running on the target node in order for the test to pass.

During the TDR test, hardware anomalies may cause some hosts to falsely identify faults in the network medium when none actually exist. Therefore, confirm any indicated faults either by performing the **nettest** TDR on at least one other host on the network or by testing the network medium with a TDR cable tester.

Since raw sockets are privileged, only the super–user may use the ICMP loopback test.

Since entry of address information into the ARP tables is a restricted operation, only the super–user may specify the Internet and physical address of an unknown node.

SEE ALSO

netstat(1n), sh(1sh), inet(3n), syslog(3c), arp(4n), hosts(5n), arp(8n), tcpd(8n), udpd(8n).

NAME

newfs — construct a new file system

SYNOPSIS

/etc/newfs [**—v**] [**—F**] [**mkfs-options**] *special*

DESCRIPTION

Newfs is a "friendly" front-end to the *mkfs(8)* program. **Newfs** will use the argument *special* to obtain information about the disk for calculating the appropriate parameters to use in calling **mkfs**, then build the file system by forking **mkfs.**

Special is the special file (device) on which the file system is to be mounted.

Newfs must be run as super-user, since it must write to the file system.

OPTIONS

—v **Newfs** will print out its actions, including the parameters passed to **mkfs**.

—F **Newfs** will only print warnings, rather than exit, if it fails to identify the system, or if it discovers the special file is a swap device.

Mkfs-options which may be used to override default parameters passed to **mkfs** are:

—b *block-size*
The block size of the file system in bytes.

—c *#cylinders/group*
The number of cylinders per cylinder group in a file system. The default value used is 16.

—f *frag-size*
The fragment size of the file system in bytes.

—i *bytes/inode*
This specifies the density of inodes in the file system. The default is to create an inode for each 2048 bytes of data space. If fewer inodes are desired, a larger number should be used; to create more inodes a smaller number should be given.

—m *freespace%*
The percentage of space reserved from normal users; the minimum free space threshhold. The default value used is 10%.

—r *revolutions/minute*
The speed of the disk in revolutions per minute (normally 3600).

—S *sector-size*
The size of a sector in bytes (almost never anything but 512).

—s *size*
The size of the file system in sectors.

—t *#tracks/cylinder*
The number of tracks per cylinder.

FILES

/etc/mkfs	to actually build the file system
/usr/mdec	for boot–strapping programs

RETURN VALUE

[NO_ERRS]	Command completed without error.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

SEE ALSO

fs(5), format(8), fsck(8), mkfs(8), tunefs(8).

NAME

lpserver, plpserver, rawlpserver — line printer servers for MDQS

SYNOPSIS

lpserver [**—b baud**] [**—c**] [**—f flagging**] [**—h size**] [**—p parity**] [**—s**] [**—C columns**] [**—H headers**] [**—T trailers**]

DESCRIPTION

Lpserver, **plpserver** and **rawlpserver** are all lineprinter severs for **MDQS**.

Lpserver is the general lineprinter server. This server optimizes for overstriking and converts all control characters to their ^⟨**letter**⟩ patterns. Since this server chatches all control characters it cannot be used to control printers with escape sequences.

Plpserver is designed to be used with Printronix P150/P300/P600 printers. This server knows about these printers special modes such as **Plotmode**.

Rawlpserver is used when one wants the data to go out the port completely unaltered. This server is especially good for outputting graphical data and escape sequences to control printer modes.

OPTIONS

—b baud
Sets the baud rate for the tty port. The default baud rate is 9600.

—c Sets the tty port in **CRMOD** which causes all **LF** characters to be output as **CR–LF**. This option implies a parity of **ODD** unless the parity is explicitly set. If the parity is explicitly set to **NONE**, **ODD** parity is still produced with the **–c** option.

—f flagging
Sets the flagging method used by the tty driver. The two values for flagging are **HW** and **SW** representing hardware and software respectively. Software flagging is the default.

—h size
Sets the amount of information presented on the banner page of the printout. The values are **LARGE**, **SMALL** and **NONE**. The default is **LARGE**.

—p parity
Sets the parity to be used by the tty driver. The values are **EVEN**, **ODD**, **SPACE** and **NONE**. The value of **NONE** allows 8–bit data transmission. **NONE** in conjunction with **rawlpserver** is useful in outputting graphical data or special control commands to printers. The default setting for parity is **NONE**.

—s Supress form–feed characters that appear at the end of each file and banner page.

—C columns
Sets the maximum number of columns to be printed on for the banner page. This is particularly useful when the paper in the printer is only 80 columns wide and you do not want filenames on the

banner page to print past the end of the paper. The default is 132.

—H headers
Sets the number of banner pages to print at the start of each file. The default is 1.

—T trailers
Sets the number of trailing banner pages to print at the end of each request. The default is 0.

EXAMPLES

The following example is for a Printronix printer running at 2400 baud and one trailing banner page.

```
/usr/lib/mdqs/plpserver -b 2400 -T 1
```

The following example is for a Centronix printer using hardware flagging.

```
/usr/lib/mdqs/lpserver -f HW
```

The following example is for a color hardcopy unit with a small banner page.

```
/usr/lib/mdqs/rawlpserver -h SMALL
```

DIAGNOSTICS

Diagnostics are passed back to the **MDQS** daemon and reported in the **MDQS** console log specified in the **qconf** file.

CAVEATS

These commands are **NEVER** called directly by the user. These commands are specified in the file **/etc/qconf** and are called by the **MDQS** daemon.

SEE ALSO

lpr(1mdqs), qconf(5mdqs), mdqsd(8mdqs), sysadmin(8), tty(4).

NAME

pstat — print system facts

SYNOPSIS

/etc/pstat —fisTtx —p[a] [**—u** *ubase*] [*system*] [corefile]

DESCRIPTION

Pstat interprets the contents of certain system tables. If *corefile* is given, the tables are sought there, otherwise in */dev/kmem*. Kernel symbols are taken from the cvt table (see *cvt(4)*) unless *system* is specified. If *system* is given, kernel symbols are obtained from the namelist in *system*.

OPTIONS

—f Print the open file table with these headings:

LOC The core location of this table entry.
TYPE The type of object the file table entry points to.
FLG Miscellaneous state variables encoded thus:
- R open for reading
- W open for writing
- A open for appending

CNT Number of processes that know this open file.
INO The location of the inode table entry for this file.
OFFS/SOCK The file offset (see *lseek(2)*), or the core address of the associated socket structure.

—i Print the inode table with the these headings:

LOC The core location of this table entry.
FLAGS Miscellaneous state variables encoded thus:
- L locked
- U update time (*fs(5)*) must be corrected
- A access time must be corrected
- M file system is mounted here
- W wanted by another process (L flag is on)
- T contains a text file
- C changed time must be corrected
- S shared lock applied
- E exclusive lock applied
- Z someone waiting for an exclusive lock

CNT Number of open file table entries for this inode.
DEV Major and minor device number of file system in which this inode resides.
RDC Reference count of shared locks on the inode.
WRC Reference count of exclusive locks on the inode (this may be > 1 if, for example, a file descriptor is inherited across a fork).
INO I-number within the device.
MODE Mode bits, see *chmod(2)*.
NLK Number of links to this inode.

UID User ID of owner.

SIZ/DEV
Number of bytes in an ordinary file, or major and minor device of special file.

—p[a]

Print process table for active processes with the following headings. If **a** is specified all processes, rather than just active ones, are described.

LOC The core location of this table entry.

S Run state encoded thus:
- 0 no process
- 1 waiting for some event
- 3 runnable
- 4 being created
- 5 being terminated
- 6 stopped under trace

F Miscellaneous state variables, or–ed together (hexadecimal):
- 000001 loaded
- 000002 the scheduler process
- 000004 locked for swap out
- 000008 swapped out
- 000010 traced
- 000020 used in tracing
- 000080 in page–wait
- 000100 prevented from swapping during *fork(2)*
- 000200 gathering pages for raw i/o
- 000400 exiting
- 001000 process resulted from a *vfork(2)* which is not yet complete
- 002000 another flag for *vfork(2)*
- 004000 process has no virtual memory, as it is a parent in the context of *vfork(2)*
- 008000 process is demand paging data pages from its text inode.
- 010000 process has advised of anomalous behavior with **vadvise**.
- 020000 process has advised of sequential behavior with **vadvise**.
- 040000 process is in a sleep which will timeout.
- 080000 a parent of this process has exited and this process is now considered detached.
- 100000 process used 4.1BSD compatibility mode signal primitives, no system calls will restart.
- 200000 process is owed a profiling tick.

POIP number of pages currently being pushed out from this process.

PRI Scheduling priority, see *setpriority(2)*.

SIGNAL
: Signals received (signals 1–32 coded in bits 0–31),

UID
: Real user ID.

SLP
: Amount of time process has been blocked.

TIM
: Time resident in seconds; times over 127 coded as 127.

CPU
: Weighted integral of CPU time, for scheduler.

NI
: Nice level, see *setpriority(2)*.

PGRP
: Process number of root of process group (the opener of the controlling terminal).

PID
: The process ID number.

PPID
: The process ID of parent process.

ADDR
: If in core, the page frame number of the first page of the 'u-area' of the process. If swapped out, the position in the swap area measured in multiples of 512 bytes.

RSS
: Resident set size — the number of physical page frames allocated to this process.

SRSS
: RSS at last swap (0 if never swapped).

SIZE
: Virtual size of process image (data + stack) in multiples of 512 bytes.

WCHAN
: Wait channel number of a waiting process.

LINK
: Link pointer in list of runnable processes.

TEXTP
: If text is pure, pointer to location of text table entry.

CLKT
: Countdown for real interval timer, *setitimer(2)* measured in clock ticks (10 milliseconds).

—s Print information about swap space usage: the number of (1k byte) pages used and free is given as well as the number of used pages which belong to text images.

—T Print the number of used and free slots in the several system tables. This option is useful for checking to see how full system tables have become if the system is under heavy load.

—t Print table for terminals with these headings:

RAW
: Number of characters in raw input queue.

CAN
: Number of characters in canonicalized input queue.

OUT
: Number of characters in output queue.

MODE
: See *tty(4)*.

ADDR
: Physical device address.

DEL
: Number of delimiters (newlines) in canonicalized input queue.

COL
: Calculated column position of terminal.

STAT
: Miscellaneous state variables encoded thus:

  W waiting for open to complete
  O open
  S has special (output) start routine
  C carrier is on
  B busy doing output
  A process is awaiting output
  X open for exclusive use

H hangup on close
PGRP Process group for which this is controlling terminal.
DISC Line discipline; blank is old tty OTTYDISC or "ntty" for NTTYDISC or "net" for NETLDISC (see *bk(4)*).

—u*ubase*
Print information about a user process; *ubase* is its address as given by *ps(1)*. The process must be in main memory, or the file used can be a core image and the address 0.

—x Print the text table with these headings:

LOC The core location of this table entry.
FLAGS Miscellaneous state variables encoded thus:
T *ptrace(2)* in effect
W text not yet written on swap device
L loading in progress
K locked
w wanted (L flag is on)
P resulted from demand–page–from–inode exec format (see *execve(2)*)
DADDR Disk address in swap, measured in multiples of 512 bytes.
CADDR Head of a linked list of loaded processes using this text segment.
SIZE Size of text segment, measured in multiples of 512 bytes.
IPTR Core location of corresponding inode.
CNT Number of processes using this text segment.
CCNT Number of processes in core using this text segment.

FILES

/dev/cvt	default source for kernel symbols
/dev/kmem	default source of tables

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.

CAVEATS

It would be very useful if the system recorded "maximum occupancy" on the tables reported by **—T**; even more useful if these tables were dynamically allocated.

SEE ALSO

ps(1), stat(2), cvt(4), fs(5).

NAME

vipw, pwck — edit and/or check the password file

SYNOPSIS

vipw [**—c** [*filename* ...]]
pwck [**—c**] [*filename* ...]

DESCRIPTION

With no arguments, **vipw** locks the file */etc/passwd,* copies its contents to a temporary file, and invokes the editor (default = **vi**) on the temporary file. After the editor is exited, the modified data is checked as described below. If no problems are found, the temporary file replaces the old password file, which is unlocked. If only problems in the *Warning* category are found, the user may re–edit the file, quit without updating, or update the password file, ignoring the warnings. If any problems in the *ERROR* category are found, the user may re–edit the file or quit without updating.

If called as **pwck** or with the **—c** option, **vipw** will check the named files as described below. If no names are given, */etc/passwd* is checked.

File Checking

The following problems are considered *Warnings*. When they occur, a message is printed giving the line number and a description of the problem.

1 User name longer than 8 characters

2 User name begins with non–alphabetic character

3 User name contains characters other than a–z, A–Z, 0–9, –, and _

4 The home directory does not exist or is not a directory

These problems are considered *ERRORs*. When they occur, a message is printed giving the line number and a description of the problem.

1 Not enough fields.

2 Too many fields.

3 Entry longer than 1024 characters.

4 Non–numeric or empty user id field.

5 Non–numeric or empty group id field.

6 The shell program does not exist or is not executable.

OPTIONS

—c Check the named files (default is */etc/passwd*). Do not edit the password file.

FILES

/etc/passwd	The password file to edit or the default file to check.
/etc/ptmp	The temporary edit file.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[1]	The file(s) contained errors.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

VARIABLES

EDIT	The editor to be used instead of **vi**.

CAVEATS

Vipw can only be used by root to edit the password file (any user can use **pwck** and **vipw** with the **—c** option). Its use should be restricted to major modifications and corrections. Make normal changes with *chfn(1), chsh(1), passwd(1), and* other utilities.

The earlier versions of this utility were shell scripts. It is imperative that the earlier versions not be used in this system. Locking is now handled by the kernel, whereas the shell scripts used a lock file.

SEE ALSO

chfn(1), chsh(1), passwd(1), passwd(5),

NAME

qdev — display and modify MDQS local device status

SYNOPSIS

/etc/qdev [**—d**] [**—e**] [**—f**] [**—l** *form*] [**—r**] [**—s**] [device ...]

DESCRIPTION

The **qdev** command is used to display or change the status of local MDQS devices. Only one option can be specified at a time. If invoked without an option, **qdev** will display the status of all local mdqs devices.

Any user may restart or flush a device if his job is active. Any user may display device status. Only the superuser, the mdqs user, or a member of the systems group may enable or disable a device, or load forms into a device, or flush or restart a device with an active request which is not his own.

OPTIONS

—d disables the specified devices. Any requests being processed by those devices are signaled to restart, causing them to be requeued for later processing. If a device has been marked as "Failed", disabling the device will clear the "Failed" flag. Thus, a "Failed" device can be restarted by first disabling and then re-enabling it.

—e enables the specified devices for processing requests

—f flushes the current request from the specified devices. The request is removed from the queueing system and a message is sent to the user who made the request indicating that the request was forcibly flushed from the device.

—l *form*
changes the queueing systems idea of the current form on a given device. The form argument must reference a valid form in the formsfile if the formsfile exists. The same form will apply to all listed devices. Forms may be used to direct requests from a single queue to appropriate devices so that, for instance, all print requests can be submitted to one queue, but requests within that queue are directed to either a wide or a narrow printer.

—r restarts the current request in each of the specified devices. The request is requeued.

—s causes the status of each of the specified devices to be printed. If no devices are specified, the status of all local devices will be given. The status information will always contain the name of the device and the currently loaded forms. If the device is disabled or flagged as having failed too many times, an appropriate message will be displayed. If there is a request being processed by that device, the request name and process ID of the filecontrol process will also be displayed.

EXAMPLES

`/etc/qdev`
displays the status of all devices

`/etc/qdev -l fanfold vp0 vp1`
loads the devices vp0 and vp1 with the forms "fanfold"

`/etc/qdev -d batch1`
disables device batch1 and restarts the request that may have been running on that device.

`/etc/qdev -f net`
flushes the current request being sent over the "net" device. A letter will be generated informing the requestor that the request was flushed.

FILES

/etc/mdqsd	MDQS daemon
/etc/qconf	configuration information for MDQS
/usr/lib/mdqs/forms	list of available forms
/usr/spool/q	top of spooling directory tree

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Things can change while **qdev** is running; the picture it gives is only a close approximation of reality. For instance, **qdev** may produce false error messages if it cannot find a particular file or if a data structure it is looking at changes underneath it.

SEE ALSO

qstat(1mdqs), qmod(1mdqs), forms(5mdqs), mdqsd(8mdqs).

NAME

quot — summarize file system ownership

SYNOPSIS

/etc/quot [**—c**] [**—f**] [**—n**] [**—v**] *filesystem* ...

DESCRIPTION

Quot prints the number of blocks in the named *filesystem* currently owned by each user. The following options are available:

—c Print three columns giving file size in blocks, number of files of that size, and cumulative total of blocks in that size or smaller file.

—f Print count of number of files as well as space owned by each user.

—n Cause the pipeline **ncheck filesystem ¦ sort +0n ¦ quot —n filesystem** to produce a list of all files and their owners.

—v Prints the following five columns of output; the total number of blocks, the users name, and three columns consisting of the number of blocks not accessed in the last 30, 60, and 90 days respectively.

EXAMPLES

The following example will print respectively the file size in blocks, the number of files of that size, and the cumulative total of blocks in that size or smaller file for the filesystem */dev/hp4a*.

```
/etc/quot -c /dev/hp4a
```

FILES

Default file system varies with system.

/etc/passwd to get user names

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Holes in files are counted as if they actually occupied space.

SEE ALSO

df(1), du(1), ls(1).

NAME

lpserver, plpserver, rawlpserver — line printer servers for MDQS

SYNOPSIS

lpserver [**—b baud**] [**—c**] [**—f flagging**] [**—h size**]
[**—p parity**] [**—s**] [**—C columns**] [**—H headers**] [**—T trailers**]

DESCRIPTION

Lpserver, **plpserver** and **rawlpserver** are all lineprinter severs for **MDQS**.

Lpserver is the general lineprinter server. This server optimizes for overstriking and converts all control characters to their ^⟨**letter**⟩ patterns. Since this server chatches all control characters it cannot be used to control printers with escape sequences.

Plpserver is designed to be used with Printronix P150/P300/P600 printers. This server knows about these printers special modes such as **Plotmode**.

Rawlpserver is used when one wants the data to go out the port completely unaltered. This server is especially good for outputting graphical data and escape sequences to control printer modes.

OPTIONS

—b baud
Sets the baud rate for the tty port. The default baud rate is 9600.

—c Sets the tty port in **CRMOD** which causes all **LF** characters to be output as **CR–LF**. This option implies a parity of **ODD** unless the parity is explicitly set. If the parity is explicitly set to **NONE**, **ODD** parity is still produced with the **–c** option.

—f flagging
Sets the flagging method used by the tty driver. The two values for flagging are **HW** and **SW** representing hardware and software respectively. Software flagging is the default.

—h size
Sets the amount of information presented on the banner page of the printout. The values are **LARGE**, **SMALL** and **NONE**. The default is **LARGE**.

—p parity
Sets the parity to be used by the tty driver. The values are **EVEN**, **ODD**, **SPACE** and **NONE**. The value of **NONE** allows 8–bit data transmission. **NONE** in conjunction with **rawlpserver** is useful in outputting graphical data or special control commands to printers. The default setting for parity is **NONE**.

—s Supress form–feed characters that appear at the end of each file and banner page.

—C columns
Sets the maximum number of columns to be printed on for the banner page. This is particularly useful when the paper in the printer is only 80 columns wide and you do not want filenames on the banner page to print past the end of the paper. The default is 132.

—H headers
Sets the number of banner pages to print at the start of each file. The default is 1.

—T trailers
Sets the number of trailing banner pages to print at the end of each request. The default is 0.

EXAMPLES

The following example is for a Printronix printer running at 2400 baud and one trailing banner page.

```
/usr/lib/mdqs/plpserver -b 2400 -T 1
```

The following example is for a Centronix printer using hardware flagging.

```
/usr/lib/mdqs/lpserver -f HW
```

The following example is for a color hardcopy unit with a small banner page.

```
/usr/lib/mdqs/rawlpserver -h SMALL
```

DIAGNOSTICS

Diagnostics are passed back to the **MDQS** daemon and reported in the **MDQS** console log specified in the **qconf** file.

CAVEATS

These commands are **NEVER** called directly by the user. These commands are specified in the file **/etc/qconf** and are called by the **MDQS** daemon.

SEE ALSO

lpr(1mdqs), qconf(5mdqs), mdqsd(8mdqs), sysadmin(8), tty(4).

NAME

rc — command script for auto–reboot and daemons

SYNOPSIS

/etc/rc
/etc/rc.local

DESCRIPTION

Rc is the command script which controls the automatic reboot and **rc.local** is the script holding commands which are pertinent only to a specific site.

When an automatic reboot is in progress, **rc** is invoked with the argument *autoboot*. If the file */fastboot* exists, it indicates that the disk check (see *fsck(8)*) run at the last system shutdown (see *shutdown(8)*) was successful and **fsck** need not be run again. If */fastboot* does not exist, **rc** runs **fsck** with option **—p** to "preen" all the disks of minor inconsistencies resulting from the last system shutdown and to check for serious inconsistencies caused by hardware or software failure. If this disk check and repair succeeds, then the second part of **rc** is run.

This second part of **rc**, which is run after an auto–reboot succeeds and also if **rc** is invoked when a single user shell terminates (see *init(8)*), starts the standard daemons on the system and performs other housekeeping tasks such as preserving editor files, clearing the scratch directory */tmp*, and saving a core image of the kernel if one was made (see *savecore(8)*). **Rc.local** and any other **rc** files that may exist, such as **rc.net** for starting network daemons and **rc.mdqs** for starting spooling daemons, are then executed.

FILES

/fastboot	indicates whether **fsck** should be run

SEE ALSO

init(8), reboot(8), savecore(8), shutdown(8).

NAME

rc.net — command script for network auto-reboot and daemons

SYNOPSIS

/etc/rc.net

DESCRIPTION

Rc.net is the command script which controls the automatic reboot for networking.

When an automatic reboot is in progress, **rc.net** is invoked by */etc/rc(8)*. *Rc.net* runs *netconfig(8n)* and uses the return code to determine the network configuration. *Rc.net* then starts the appropriate network daemon processes before terminating and returning control back to */etc/rc(8)*.

There are four possible network configurations:

1. Enable no networking whatsoever. Start no network daemons.
2. Enable Distributed File System (DFS) only. Start only the daemons that are required by the DFS, such as *syslog(8)*, *nameserver(8n)*, and *dfsd(8n)*.
3. Enable Networking but no DFS. Start *syslog(8)*, *nameserver(8n)*, and *udpd(8n)*. *Tcpd(8n)* and *udpd(8n)*, will serve as the daemons for *rlogin(1n) rsh(1n), uptime(1n), ftp(1n), telnet(1n),* and any other services that rely on the *tcp(4n), ip(4n),* and *udp(4n),* network protocols.
4. Enable Networking and DFS. Start all of the DFS and network daemons.

FILES

network.conf(5n)	This file is used by *netconfig(8n)* to remember from one boot to the next what the network configuration is.

SEE ALSO

netconfig(8n).

NAME

rdump — file system dump across the network

SYNOPSIS

/etc/rdump [*key* [*argument* ...] *filesystem*]

DESCRIPTION

Rdump copies to magnetic tape all files changed in the *filesystem* after a certain date. The command is identical in operation to *dump(8)* except that the **f** key should be specified and the file supplied should be of the form *machine:device*.

Rdump uses rexec to creates a remote server, */etc/rmt*, on the client machine to access the tape device.

FILES

~/.netrc

DIAGNOSTICS

Same as *dump(8)* with some additional network related messages.

RETURN VALUE

[0] *Rdump* was successful.

[1] *Rdump* was unsuccessful.

[NP_WARN] An error warranting a warning message occurred. Execution continues.

SEE ALSO

dump(8), rmt(8n).

NAME

reboot — UTek bootstrapping procedures

SYNOPSIS

/etc/reboot [**—n**] [**—q**]

DESCRIPTION

UTek is started by placing it in memory transferring to it. Since the system is not reenterable, it is necessary to read it in from disk or tape each time it is to be bootstrapped.

Rebooting a running system. When UTek is running and a **reboot** is desired, *shutdown(8)* is normally used. If there are no users then **/etc/reboot** can be used. **Reboot** causes the disks to be synched, and a multi–user **reboot** (as described below) is initiated. This causes a system to be booted and an automatic disk check to be performed. If all this succeeds without incident, the system is then brought up multi–user.

Power fail and crash recovery. Normally, the system will reboot itself at power–up or after crashes. An automatic consistency check of the file systems will be performed and unless this fails the system will resume multi–user operation.

OPTIONS

—n Avoids the sync.

—q Reboots quickly and ungracefully, without shutting down running processes first.

FILES

/vmunix	system code

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.

SEE ALSO

fsck(8), halt(8), init(8), newfs(8), rc(8), shutdown(8).

NAME

restore — incremental file system restore

SYNOPSIS

/etc/restore *key* [*argument...*] [*filename...*]

DESCRIPTION

Restore reads tapes dumped with the *dump(8)* command. Its actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying the files that are to be restored. Unless the **h** key is specified (see below), the appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

OPTIONS

The function portion of the key is specified by one of the following letters:

r The tape is read and loaded into the current directory. This should not be done lightly; the **r** key should only be used to restore a complete dump tape onto a clear file system or to restore an incremental dump tape after a full level zero restore. Thus

```
/etc/newfs /dev/rrp0g eagle
/etc/mount /dev/rp0g /mnt
cd /mnt
/etc/restore rf /dev/rmt1
```

is a typical sequence to restore a complete dump. Another **restore** can be done to get an incremental dump in on top of this. Note that **restore** leaves a file *restoresymtable* in the root directory to pass information between incremental restore passes. This file should be removed when the last incremental tape has been restored.
A *dump(8)* followed by a *newfs(8)* and a **restore** is used to change the size of a file system.

R **Restore** requests a particular volume of a multi volume set on which to restart a full restore (see the **r** key above). This allows **restore** to be interrupted and then restarted.

x The named files are extracted from the tape. If the named file matches a directory whose contents had been written onto the tape, and the **h** key is not specified, the directory is recursively extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, then the root directory is extracted, which results in the entire content of the tape being extracted, unless the **h** key has been specified.

t The names of the specified files are listed if they occur on the tape. If no file argument is given, then the root directory is listed, which results in the entire content of the tape being listed, unless the **h** key has been specified. Note that the **t** key replaces the function of the old **dumpdir** program.

i This mode allows interactive restoration of files from a dump tape. After reading in the directory information from the tape, **restore** provides a shell like interface that allows the user to move around the directory tree selecting files to be extracted. The available commands are given below; for those commands that require an argument, the default is the current directory.

ls [*arg*] — List the current or specified directory. Entries that are directories are appended with a /. Entries that have been marked for extraction are prepended with a *. If the verbose key is set the inode number of each entry is also listed.

cd [*arg*] — Change the current working directory to the specified argument.

pwd — Print the full pathname of the current working directory.

add [*arg*] — The current directory or specified argument is added to the list of files to be extracted. If a directory is specified, then it and all its descendents are added to the extraction list (unless the **h** key is specified on the command line). Files that are on the extraction list are prepended with a * when they are listed by **ls**.

delete [*arg*]— The current directory or specified argument is deleted from the list of files to be extracted. If a directory is specified, then it and all its descendents are deleted from the extraction list (unless the **h** key is specified on the command line). The most expedient way to extract most of the files from a directory is to add the directory to the extraction list and then delete those files that are not needed.

extract — All the files that are on the extraction list are extracted from the dump tape. **Restore** will ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

verbose — The sense of the **v** key is toggled. When set, the verbose key causes the **ls** command to list the inode numbers of all entries. It also causes **restore** to print out information about each file as it is extracted.

help — List a summary of the available commands.

quit — Restore immediately exits, even if the extraction list is not empty.

The following characters may be used in addition to the letter that selects the function desired.

b The next argument to **restore** indicates buffer size for reading dump media. The argument is interpreted as number of 1k bytes, and it is intended as a means of speeding up media reads (specifying the S option for cartridge streaming tape causes the buffer size to be set to 128 for a buffer size of 128k). The default is 10k. The number specified should agree with that used by dump to produce the media. This option should not be used when reading flexible disk media.

F This indicates flexible disk media is being used (default is 9 track tape).

S This indicates that cartridge streaming tape is being used (default is 9 track tape).

v Normally **restore** does its work silently. The **v** (verbose) key causes it to type the name of each file it treats preceded by its file type.

f The next argument to **restore** is the path name of the device to use instead of the default. The path name can specify a flexible disk, a cartridge tape, a 9–track tape, or a disk file, and the location of the device can be local or remote. If the name of the file is "—", **restore** reads from standard input. Thus, *dump(8)* and **restore** can be used in a pipeline to dump and restore a file system with the command

```
/etc/dump 0f - /usr ¦ (cd /mnt; /etc/restore xf -)
```

y **Restore** will not ask whether it should abort the restore if it gets a tape error. It will always try to skip over the bad tape block(s) and continue as best it can.

m **Restore** will extract by inode numbers rather than by file name. This is useful if only a few files are being extracted, and one wants to avoid regenerating the complete pathname to the file.

h **Restore** extracts the actual directory, rather than the files that it references. This prevents hierarchical restoration of complete subtrees from the tape.

EXAMPLES

cd /fs (where fs is the file system to which you are restoring, e.g., /ab)

`/etc/restore if /dev/tc`
(use restore interactively from the cartridge streaming tape on */dev/tc)*

`/etc/restore tf /dev/rdf`
(this will show all files involved in this flexible disk dump. The information will be extracted from the volume)

`/etc/restore tf /dev/rdf ./joe`
(will show all files in subtree joe, where joe is a child of the dumped file system */cd/joe*).

`/etc/restore tf /dev/rdf joe`
(same function as above)

`cd /fs` (fs is same as above)

`/etc/restore xf /dev/rdf ./joe/thisdir`
(will restore *thisdir* and everything dumped below *thisdir*. This may not be all of *thisdir*, since dump only grabs files that have changed.)

Note that syntax is relative to the current directory. For example, if **restore** is preceded by **cd /ab**, then *./joe*, *./joe/thisdir*, etc., will be created as subtrees of */ab*.

FILES

/dev/tc	the default tape drive (cartridge streamer)
*/tmp/rstdir**	file containing directories on the tape.
*/tmp/rstmode**	owner, mode, and time stamps for directories.
./restoresymtable	symtab information passed between incremental restores.

DIAGNOSTICS

Complaints about bad key characters.

Complaints if it gets a read error. If **y** has been specified, or the user responds **y**, **restore** will attempt to continue the **restore**.

If the dump extends over more than one volume, **restore** will ask the user to change tapes. If the **x** or **i** key has been specified, **restore** will also ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

There are numerous consistency checks that can be listed by **restore**. Most checks are self-explanatory or can "never happen". Common errors are given below.

Converting to new file system format.
A dump tape created from the old file system has been loaded. It is automatically converted to the new file system format.

⟨*filename*⟩: not found on tape
The specified file name was listed in the tape directory, but was not found on the tape. This is caused by tape read errors while looking for the file, and from using a dump tape created on an active file system.

expected next file ⟨*inumber*⟩, got ⟨*inumber*⟩
A file that was not listed in the directory showed up. This can occur when using a dump tape created on an active file system.

Incremental tape too low
: When doing incremental restore, a tape that was written before the previous incremental tape, or that has too low an incremental level has been loaded.

Incremental tape too high
: When doing incremental restore, a tape that does not begin its coverage where the previous incremental tape left off, or that has too high an incremental level has been loaded.

Tape read error while restoring ⟨*filename*⟩
Tape read error while skipping over inode ⟨ *inumber* ⟩
Tape read error while trying to resynchronize
: A tape read error has occurred. If a file name is specified, then its contents are probably partially wrong. If an inode is being skipped or the tape is trying to resynchronize, then no extracted files have been corrupted, though files may not be found on the tape.

resync restore, skipped ⟨*num*⟩ blocks
: After a tape read error, **restore** may have to resynchronize itself. This message lists the number of blocks that were skipped over.

RETURN VALUE

[USAGE] Incorrect command line syntax. Execution terminated.

[NP_ERR] An error occurred that was not a system error. Execution terminated.

This is the return code used when restore has found that one or more files it was about to restore already exist (the files are left alone). In this case the system error variable, errno, will contain EEXIST (17).

[P_ERR] A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

[NP_WARN] An error warranting a warning message occurred. Execution continues.

Return values greater than 124 indicate a fatal condition from which restore aborted.

CAVEATS

Restore can get confused when doing incremental **restores** from dump tapes that were made on active file systems.

A level zero dump must be done after a full **restore**. Because **restore** runs in user code, it has no control over inode allocation; thus a full **restore** must be done to get a new set of directories reflecting the new inode numbering, even though the contents of the files is unchanged.

SEE ALSO

rrestore(8), dump(8), rdump(8), mkfs(8), mount(8), newfs(8).

NAME

rexecd — remote execution server

SYNOPSIS

/etc/tcp_services/rexecd

DESCRIPTION

Rexecd is the server for the *rexec(3n)* routine. The server provides remote execution facilities with authentication based on user names and encrypted passwords.

Rexecd is started by *tcpd(8n)* when a service requests at the port indicated in the "exec" service specification; see *services(5n)*. When a service request is received the following protocol is initiated:

1) The server reads characters from the socket up to a null ('\0') byte. The resultant string is interpreted as an ASCII number, base 10.

2) If the number received in step 1 is non–zero, it is interpreted as the port number of a secondary stream to be used for the **stderr**. A second connection is then created to the specified port on the client's machine.

3) A null terminated user name of at most 16 characters is retrieved on the initial socket.

4) A null terminated, encrypted, password of at most 16 characters is retrieved on the initial socket.

5) A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.

6) **Rexecd** then validates the user as is done at login time and, if the authentication was successful, changes to the user's home directory, and establishes the user and group protections of the user. If any of these steps fail the connection is aborted with a diagnostic message returned.

7) A null byte is returned on the connection associated with the **stderr** and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by **rexecd**.

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of 1 (0 is returned in step 7 above upon successful completion of all the steps prior to the command execution).

username too long
The name is longer than 16 characters.

password too long
The password is longer than 16 characters.

command too long
The command line passed exceeds the size of the argument list (as configured into the system).

Login incorrect.
No password file entry for the user name existed or the wrong password was supplied.

No remote directory.
The *chdir* command to the home directory failed.

Try again.
A *fork* by the server failed.

/bin/sh: . . .
The user's login shell could not be started.

RETURN VALUE

[0] **Rexecd** is running.

[1] **Rexecd** is not running.

[USAGE]	Incorrect command line syntax. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.

SEE ALSO

rexec(3n), tcpd(8n).

NAME

rlogind — remote login server

SYNOPSIS

/etc/tcp_services/rlogind [**—d**]

DESCRIPTION

Rlogind is the server for the *rlogin(1n)* program. The server provides a remote login facility with authentication based on privileged port numbers.

Rlogind is run by *tcpd(8n)* when a connection is made on the "login" service specification; see *services(5n)*. When a service request is received the following protocol is initiated:

1) The server checks the client's source port. If the port is not in the range 0–1023, the server aborts the connection.

2) The server checks the client's source address. If the address is associated with a host for which no corresponding entry exists in the host name data base (see *hosts(5n)*), the server aborts the connection.

Once the source port and address have been checked, *rlogind* allocates a pseudo terminal and manipulates file descriptors so that the slave half of the pseudo terminal becomes the **stdin , stdout ,** and **stderr** for a login process. The login process is an instance of the *login(1)* program, invoked with the **—r** option. The login process then proceeds with the authentication process as described in *rshd(8n)*, but if automatic authentication fails, it reprompts the user to login as one finds on a standard terminal line.

The parent of the login process manipulates the master side of the pseduo terminal, operating as an intermediary between the login process and the client instance of the *rlogin* program. The login process propagates the client terminal's baud rate and terminal type, as found in the environment variable, "TERM"; see *environ(7)* .

OPTIONS

—d Turn on socket debugging for use with **trpt**.

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of 1.

Hostname for your address unknown.
No entry in the host name database existed for the client's machine.

Try again.
A **fork** by the server failed.

/bin/sh: ...
The user's login shell could not be started.

RETURN VALUE

[0] **Rlogind** is running.

[1] **Rlogind** is not running.

[USAGE]	Incorrect command line syntax. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.

CAVEATS

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an "open" environment.

SEE ALSO

rlogin(1n), rsh(1n), tcpd(8n).

NAME

rmt — remote magtape protocol module

SYNOPSIS

/etc/rmt

DESCRIPTION

Rmt is a program used by the remote dump and restore programs in manipulating a magnetic tape drive through an interprocess communication connection. **Rmt** is normally started up with an *rexec(3n)* or *rcmd(3n)* call.

The **rmt** program accepts requests specific to the manipulation of magnetic tapes, performs the commands, then responds with a status indication. All responses are in ASCII and in one of two forms. Successful commands have responses of

A*number*\n

where *number* is an ASCII representation of a decimal number. Unsuccessful commands are responded to with

E*error-number*\n*error-message*\n,

where *error-number* is one of the possible error numbers described in *intro(2)* and *error-message* is the corresponding error string as printed from a call to *perror(3c)*. The protocol is comprised of the following commands (a space is present between each token).

O *device mode*
Open the specified *device* using the indicated *mode*. *Device* is a full pathname and *mode* is an ASCII representation of a decimal number suitable for passing to *open(2)*. If a device had already been opened, it is closed before a new open is performed.

C *device*
Close the currently open device. The *device* specified is ignored.

L *whence offset*
Perform an *lseek(2)* operation using the specified parameters. The response value is that returned from the **lseek** call.

W *count*
Write data onto the open device. **Rmt** reads *count* bytes from the connection, aborting if a premature end-of-file is encountered. The response value is that returned from the *write(2)* call.

R *count*
Read *count* bytes of data from the open device. If *count* exceeds the size of the data buffer (10 kilobytes), it is truncated to the data buffer size. *Rmt* then performs the requested *read(2)* and responds with **A***count-read*\n if the read was successful; otherwise an error in the standard format is returned. If the read was successful, the data read is then sent.

I *operation count*
Perform a MTIOCOP *ioctl(2)* command using the specified parameters. The parameters are interpreted as the ASCII representations of the decimal values to place in the *mt_op* and *mt_count* fields of the structure used in the *ioctl* call. The return value is the *count* parameter when the operation is successful.

S Return the status of the open device, as obtained with a MTIOCGET *ioctl* call. If the operation was successful, an *ack* is sent with the size of the status buffer, then the status buffer is sent (in binary).

Any other command causes **rmt** to exit.

DIAGNOSTICS

All responses are of the form described above.

RETURN VALUE

[0] **Rmt** was successful.

[1] **Rmt** was unsuccessful.

CAVEATS

People tempted to use this for a remote file access protocol are discouraged.

SEE ALSO

rcmd(3n), rexec(3n), rdump(8n), rrestore(8n).

NAME

route — manually manipulate the routing tables

SYNOPSIS

/etc/route [**—f**] [*command args*]

DESCRIPTION

Route is a program used to manually manipulate the network routing tables. It normally is not needed, as the system routing table management daemon, *routed(8n)*, should tend to this task.

Route accepts three commands: **add**, to add a route; **delete**, to delete a route; and **change**, to modify an existing route.

All commands have the following syntax:

```
/etc/route command destination gateway [ metric ]
```

where *destination* is a host or network for which the route is "to", *gateway* is the gateway to which packets should be addressed, and *metric* is an optional count indicating the number of hops to the *destination*. If no metric is specified, **route** assumes a value of 0. Routes to a particular host are distinguished from those to a network by interpreting the Internet address associated with *destination*. If the *destination* has a "local address part" of INADDR_ANY, then the route is assumed to be to a network; otherwise, it is presumed to be a route to a host. If the route is to a destination connected via a gateway, the *metric* should be greater than 0. All symbolic names specified for a *destination* or *gateway* are looked up first in the host name database, *hosts(5n)*. If this lookup fails, the name is then looked for in the network name database, *networks(5n)*.

Route uses a raw socket and the SIOCADDRT and SIOCDELRT *ioctl*'s to do its work. As such, only the super–user may modify the routing tables.

OPTIONS

—f **Route** will "flush" the routing tables of all gateway entries. If this is used in conjunction with one of the commands described above, the tables are flushed prior to the command's application.

command args
See above for description.

DIAGNOSTICS

add %s: gateway %s flags %x
The specified route is being added to the tables. The values printed are from the routing table entry supplied in the *ioctl* call.

delete %s: gateway %s flags %x
As above, but when deleting an entry.

%s %s done
When the **—f** flag is specified, each routing table entry deleted is indicated with a message of this form.

not in table
A delete operation was attempted for an entry which wasn't present in the tables.

routing table overflow
An add operation was attempted, but the system was low on resources and was unable to allocate memory to create the new entry.

RETURN VALUE

[0]	No errors occurred.
[1]	Errors occurred.

CAVEATS

The change operation is not implemented, one should add the new route, then delete the old one.

SEE ALSO

intro(4n), routed(8n).

NAME

routed — network routing daemon

SYNOPSIS

/etc/routed [**—s**] [**—q**] [**—t**] [*logfile*]

DESCRIPTION

Routed is invoked at boot time to manage the network routing tables. The routing daemon uses a variant of the Xerox NS Routing Information Protocol in maintaining up to date kernel routing table entries.

In normal operation **routed** listens on *udp(4n)* socket 520 (decimal) for routing information packets. If the host is an internetwork router, it periodically supplies copies of its routing tables to any directly connected hosts and networks.

When **routed** is started, it uses the SIOCGIFCONF *ioctl* to find those directly connected interfaces configured into the system and marked "up" (the software loopback interface is ignored). If multiple interfaces are present, it is assumed the host will forward packets between networks. **Routed** then transmits a **request** packet on each interface (using a broadcast packet if the interface supports it) and enters a loop, listening for **request** and **response** packets from other hosts.

When a **request** packet is received, **routed** formulates a reply based on the information maintained in its internal tables. The **response** packet generated contains a list of known routes, each marked with a "hop count" metric (a count of 16, or greater, is considered "infinite"). The metric associated with each route returned provides a metric *relative to the sender.*

Response packets received by **routed** are used to update the routing tables if one of the following conditions is satisfied:

(1) No routing table entry exists for the destination network or host, and the metric indicates the destination is "reachable" (i.e. the hop count is not infinite).

(2) The source host of the packet is the same as the router in the existing routing table entry. That is, updated information is being received from the very internetwork router through which packets for the destination are being routed.

(3) The existing entry in the routing table has not been updated for some time (defined to be 90 seconds) and the route is at least as cost effective as the current route.

(4) The new route describes a shorter route to the destination than the one currently stored in the routing tables; the metric of the new route is compared against the one stored in the table to decide this.

When an update is applied, **routed** records the change in its internal tables and generates a **response** packet to all directly connected hosts and networks. **Routed** waits a short period of time (no more than 30

seconds) before modifying the kernel's routing tables to allow possible unstable situations to settle.

In addition to processing incoming packets, **routed** also periodically checks the routing table entries. If an entry has not been updated for 3 minutes, the entry's metric is set to infinity and marked for deletion. Deletions are delayed an additional 60 seconds to insure the invalidation is propagated throughout the internet.

Hosts acting as internetwork routers gratuitously supply their routing tables every 30 seconds to all directly connected hosts and networks.

In addition to the facilities described above, **routed** supports the notion of "distant" *passive* and *active* gateways. When **routed** is started up, it reads the file */etc/gateways* to find gateways which may not be identified using the SIOGIFCONF *ioctl*. Gateways specified in this manner should be marked passive if they are not expected to exchange routing information, while gateways marked active should be willing to exchange routing information (i.e. they should have a **routed** process running on the machine). Passive gateways are maintained in the routing tables forever and information regarding their existence is included in any routing information transmitted. Active gateways are treated equally to network interfaces. Routing information is distributed to the gateway and if no routing information is received for a period of the time, the associated route is deleted.

The */etc/gateways* is comprised of a series of lines, each in the following format:

< **net** ¦ **host** > *name1* **gateway** *name2* **metric** *value* < **passive** ¦ **active** >

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 an Internet address specified in "dot" notation; see *inet(3n)*.

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.

The keyword **passive** or **active** indicates if the gateway should be treated as *passive* or *active* (as described above).

OPTIONS

—s This option forces **routed** to supply routing information whether it is acting as an internetwork router or not.

—q This option is the opposite of the **—s** option.

—t This option causes, all packets sent or received to be printed on the standard output. In addition, **routed** will not divorce itself from the controlling terminal so that interrupts from the keyboard will kill the process.

logfile
Any other argument supplied is interpreted as the name of the file in which **routed**'s actions should be logged. This log contains information about any changes to the routing tables and a history of recent messages sent and received which are related to the changed route.

FILES

/etc/gateways	for distant gateways

RETURN VALUE

[0]	**Routed** is running.
[1]	**Routed** is not running.
[-1]	No action taken.

CAVEATS

The kernel's routing tables may not correspond to those of **routed** for short periods of time while processes utilizing existing routes exit; the only remedy for this is to place the routing process in the kernel.

SEE ALSO

udp(4n).

NAME

rrestore — restore a file system dump across the network

SYNOPSIS

/etc/rrestore [*key*] [*filename* ...]

DESCRIPTION

Rrestore obtains from magnetic tape files saved by a previous *dump(8)*. The command is identical in operation to *restore(8)* except that the *f* key should be specified and the file supplied should be of the form *machine:device*.

Rrestore creates a remote server */etc/rmt* on the client machine to access the tape device.

DIAGNOSTICS

Same as *restore(8)* with a few extra related to the network.

RETURN VALUE

[0] *Rrestore* was successful.

[1] *Rrestore* was unsuccessful.

[NP_WARN] An error warranting a warning message occurred. Execution continues.

SEE ALSO

restore(8), rmt(8n).

NAME

rshd — remote shell server

SYNOPSIS

/etc/tcp_services/rshd

DESCRIPTION

Rshd is the server for the *rcmd(3n)* routine and, consequently, for the *rsh(1n)* program. The server provides remote execution facilities with authentication based on privileged port numbers.

Tcpd spawns an **Rshd** process for service requests at the port indicated in the *cmd* service specification; see *services(5n)*. When a service request is received the following protocol is initiated:

1) The server checks the client's source port. If the port is not in the range 0–1023, the server aborts the connection.

2) The server reads characters from the socket up to a null ('\0') byte. The resultant string is interpreted as an ASCII number, base 10.

3) If the number received in step 1 is non–zero, it is interpreted as the port number of a secondary stream to be used for the **stderr**. A second connection is then created to the specified port on the client's machine. The source port of this second connection is also in the range 0–1023.

4) The server checks the client's source address. If the address is associated with a host for which no corresponding entry exists in the host name data base (see *hosts(5n)*), the server aborts the connection.

5) A null terminated user name of at most 16 characters is retrieved on the initial socket. This user name is interpreted as a user identity to use on the **server**'s machine.

6) A null terminated user name of at most 16 characters is retrieved on the initial socket. This user name is interpreted as the user identity on the **client**'s machine.

7) A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.

8) **Rshd** then validates the user according to the following steps. The remote user name is looked up in the password file and a **chdir** is performed to the user's home directory. If either the lookup or **chdir** fail, the connection is terminated. If the user is not the super–user, (user ID 0), the file */etc/hosts.equiv* is consulted for a list of hosts considered "equivalent". If the client's host name is present in this file, the authentication is considered successful. If the lookup fails, or the user is the super–user, then the file *.rhosts* in the home directory of the remote user is checked for the machine name and identity of the

user on the client's machine. If this lookup fails, the connection is terminated.

9) A null byte is returned on the connection associated with the **stderr** and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by **rshd**.

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of 1 (0 is returned in step 9 above upon successful completion of all the steps prior to the command execution).

locuser too long
The name of the user on the client's machine is longer than 16 characters.

remuser too long
The name of the user on the remote machine is longer than 16 characters.

command too long
The command line passed exceeds the size of the argument list (as configured into the system).

Hostname for your address unknown.
No entry in the host name database existed for the client's machine.

Login incorrect.
No password file entry for the user name existed.

No remote directory.
The *chdir* command to the home directory failed.

Permission denied.
The authentication procedure described above failed.

Can't make pipe.
The pipe needed for the **stderr**, wasn't created.

Try again.
A *fork* by the server failed.

/bin/sh: . . .
The user's login shell could not be started.

RETURN VALUE

[0] **Rshd** is running.

[1] **Rshd** is not running.

[USAGE] Incorrect command line syntax. Execution terminated.

[P_ERR] A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

[NP_WARN] An error warranting a warning message occurred. Execution continues.

[NP_ERR] An error occurred that was not a system error. Execution terminated.

CAVEATS

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an "open" environment.

SEE ALSO

rsh(1n), rcmd(3n), tcpd(8n).

NAME

rwhod — system status server

SYNOPSIS

/etc/rwhod [—t*scanrate*] [—f*logfile*]

DESCRIPTION

Rwhod is the server which maintains the database used by the **rwho** and **uptime** programs. Its operation is predicated on the ability to *broadcast* messages on a network.

Rwhod operates as both a producer and consumer of status information. As a producer of information it periodically queries the state of the system and constructs status messages which are broadcast on a network. As a consumer of information, it listens for other **rwhod** servers' status messages, validating them, then recording them in a collection of files located in the directory */usr/spool/rwho*.

The **rwho** server transmits and receives messages at the port indicated in the **rwho** service specification, see *services(5n)*. The messages sent and received, are of the form:

```
struct  outmp {
        char    out_line[8];/* tty name */
        char    out_name[8];/* user id */
        long    out_time;/* time on */
};

struct  whod {
        char    wd_vers;
        char    wd_type;
        char    wd_fill[2];
        int     wd_sendtime;
        int     wd_recvtime;
        char    wd_hostname[32];
        int     wd_loadav[3];
        int     wd_boottime;
        struct  whoent {
                struct  outmp we_utmp;
                int     we_idle;
        } wd_we[1024 / sizeof (struct whoent)];
};
```

All fields are converted to network byte order prior to transmission. The load averages are as calculated by the *w(1)* program, and represent load averages over the 5, 10, and 15 minute intervals prior to a server's transmission. The host name included is that returned by the *gethostname(2)* system call. The array at the end of the message contains information about the users logged in to the sending machine. This information includes the contents of the *utmp(5)* entry for each non-idle terminal line and a value indicating the time since a character was last received on the terminal line.

Messages received by the **rwho** server are discarded unless they originated at a **rwho** server's port. In addition, if the host's name, as specified in the message, contains any unprintable ASCII characters, the message is discarded. Valid messages received by **rwhod** are placed in files named *whod.hostname* in the directory */usr/spool/rwho*. These files contain only the most recent message, in the format described above.

Status messages are generated approximately once every 60 seconds. **Rwhod** performs an *nlist(3c)* on */vmunix* every 10 minutes to guard against the possibility that this file is not the system image currently operating.

OPTIONS

—t*scanrate*

Scanrate is the number of seconds between each status broadcast [default 2 minutes]. Note: *ruptime(1n)* considers a host down if no packet has arrived in 5 minutes.

—f*logfile*

Put debugging/trace information into *logfile*.

RETURN VALUE

[0] **Rwhod** is running.

[1] **Rwhod** is not running.

CAVEATS

As the number of hosts on the network increases, the number of **rwho** packets increases as the square of the number of hosts. This can have a detrimental effect on workstation performance, so for that reason, **rwhod** is not normally enabled. If you do wish to install and run it, see **ruptime(1n)**.

SEE ALSO

ruptime(1n), uptime(1n), rwho(1n).

NAME

savecore — save a core dump of the operating system

SYNOPSIS

/etc/savecore *dirname* [**—d** *dumpdev*] [**—s** *system*]

DESCRIPTION

Savecore is meant to be called near the end of the */etc/rc* file. Its function is to save the core dump of the system (assuming one was made) and to write a reboot message in the shutdown log.

Savecore checks the core dump to be certain it corresponds with the current running system. If it does it saves the core image in the file *dirname/vmcore.n* and its brother, the namelist, in *dirname/vmunix.n* The trailing *.n* in the pathnames is replaced by a number which grows every time **savecore** is run in that directory. This number is read from the second line of the file *dirname/savecore.bounds*, and is incremented whenever a core image is saved.

Before **savecore** writes out a core image, it reads an additional number from the first line of the file *dirname/savecore.bounds*. If there are fewer free blocks on the filesystem which contains *dirname* than this number obtained from the *savecore.bounds* file, the core dump is not done. If the *savecore.bounds* file does not exist, **savecore** always writes out the core file (assuming that a core dump was taken).

Savecore also writes a reboot message in the shut down log. If the system crashed as a result of a panic, **savecore** records the panic string in the shut down log too.

OPTIONS

—d *dumpdev*
The name of the device containing the dump may be supplied as *dumpdev*. Otherwise, the dump device is read from the kernel.

—s *system*
If the core dump was from a system other than the default, the name of that system may be supplied as *system*.

FILES

/usr/adm/shutdownlog	shut down log
dirname/savecore.bounds	
/dev/cvt	table of kernel symbols

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.

[P_WARN] A system error occurred. Execution continues. See *intro(2)* for more information on system errors.

[P_ERR] A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Savecore can be fooled into thinking a core dump is the wrong size.

SEE ALSO

cvt(4).

NAME

sendmail, mailq — send mail over the internet

SYNOPSIS

/usr/lib/sendmail [*flags*] [*address ...*]

newaliases

mailq

DESCRIPTION

Sendmail sends a message to one or more people, routing the message over whatever networks are necessary. **Sendmail** does internetwork forwarding as necessary to deliver the message to the correct place.

If invoked as **newaliases**, **sendmail** will rebuild the alias database. If invoked as **mailq**, **sendmail** will print the contents of the mail queue.

Sendmail is not intended as a user interface routine; other programs provide user–friendly front ends; **sendmail** is used only to deliver pre–formatted messages.

With no flags, **sendmail** reads its standard input up to a control–D or a line with a single dot and sends a copy of the letter found there to all of the addresses listed. It determines the network to use based on the syntax and contents of the addresses.

Local addresses are looked up in a file and aliased appropriately. Aliasing can be prevented by preceding the address with a backslash. Normally the sender is not included in any alias expansions, e.g., if 'john' sends to 'group', and 'group' includes 'john' in the expansion, then the letter will not be delivered to 'john'.

If the first character of the user name is a vertical bar, the rest of the user name is used as the name of a program to pipe the mail to. It may be necessary to quote the name of the user to keep **sendmail** from suppressing the blanks from between arguments.

OPTIONS

—ba Go into ARPANET mode. All input lines must end with a CR–LF, and all messages will be generated with a CR–LF at the end. Also, the "From:" and "Sender:" fields are examined for the name of the sender.

—bd Run as a daemon. This requires Berkeley IPC.

—bi Initialize the alias database.

—bm Deliver mail in the usual way (default).

—bp Print a listing of the queue.

—bs Use the SMTP protocol as described in RFC821. This flag implies all the operations of the **—ba** flag that are compatible with SMTP.

—bt Run in address test mode. This mode reads addresses and shows the steps in parsing; it is used for debugging configuration tables.

—bv Verify names only — do not try to collect or deliver a message. Verify mode is normally used for validating users or mailing lists.

—bz Create the configuration freeze file.

—C*filename*
Use alternate configuration file.

—d*X* Set debugging value to *X*.

—F*fullname*
Set the full name of the sender.

—f*name*
Sets the name of the "from" person (i.e., the sender of the mail). **—f** can only be used by the special users *root, daemon,* and *network,* or if the person you are trying to become is the same as the person you are.

—h*N* Set the hop count to *N*. The hop count is incremented every time the mail is processed. When it reaches a limit, the mail is returned with an error message, the victim of an aliasing loop.

—n Don't do aliasing.

—o*x value*
Set option *x* to the specified *value*. Options are described below.

—q[*time***]**
Processed saved messages in the queue at given intervals. If *time* is omitted, process the queue once. *Time* is given as a tagged number, with 's' being seconds, 'm' being minutes, 'h' being hours, 'd' being days, and 'w' being weeks. For example, "—q1h30m" or "—q90m" would both set the timeout to one hour thirty minutes.

—r*name*
An alternate and obsolete form of the **—f** flag.

—t Read message for recipients. To:, Cc:, and Bcc: lines will be scanned for people to send to. The Bcc: line will be deleted before transmission. Any addresses in the argument list will be suppressed.

—v Go into verbose mode. Alias expansions will be announced, etc.

There are also a number of processing options that may be set. Normally these will only be used by a system administrator. Options may be set either on the command line using the **—o** flag or in the configuration file. These are described in detail in the *Installation and Operation Guide.* The options are:

A*filename*
Use alternate alias file.

c On mailers that are considered "expensive" to connect to, don't initiate immediate connection. This requires queueing.

d*x* Set the delivery mode to *x*. Delivery modes are 'i' for interactive (synchronous) delivery, 'b' for background (asynchronous) delivery, and 'q' for queue only — i.e., actual delivery is done the next time the queue is run.

D Try to automatically rebuild the alias database if necessary.

e*x* Set error processing to mode *x*. Valid modes are 'm' to mail back the error message, 'w' to "write" back the error message (or mail it back if the sender is not logged in), 'p' to print the errors on the terminal (default), 'q' to throw away error messages (only exit status is returned), and 'e' to do special processing for the BerkNet. If the text of the message is not mailed back by modes 'm' or 'w' and if the sender is local to this machine, a copy of the message is appended to the file *dead.letter* in the sender's home directory.

F*mode* The mode to use when creating temporary files.

f Save UTek-style From lines at the front of messages.

g*N* The default group id to use when calling mailers.

H*file* The SMTP help file.

i Do not take dots on a line by themselves as a message terminator.

L*n* The log level.

m Send to "me" (the sender) also if I am in an alias expansion.

o If set, this message may have old style headers. If not set, this message is guaranteed to have new style headers (i.e., commas instead of spaces between addresses). If set, an adaptive algorithm is used that will correctly determine the header format in most cases.

Q*queuedir*
Select the directory in which to queue messages.

r*timeout*
The timeout on reads; if none is set, **sendmail** will wait forever for a mailer.

S*file* Save statistics in the named file.

s Always instantiate the queue file, even under circumstances where it is not strictly necessary.

T*time* Set the timeout on messages in the queue to the specified time. After sitting in the queue for this amount of time, they will be returned to the sender. The default is three days.

t*stz,dtz* Set the name of the time zone.

u*N* Set the default user ID for mailers.

x*la* If the load average is greater than *la* mail is queued (for later delivery) rather than processed immediately.

X*la* If the load average is greater than *la* remote smtp connections to the daemon are refused.

EXAMPLES

Given a file *testletter* like:

```
To: joe
Subject: Sample sendmail letter

This is the hard way to send mail
```

The command to mail it would be:

```
sendmail -t -i -v <testletter
```

The **—t** tells **sendmail** to read the addresses from the letter. The **—i** tells **sendmail** to deliver interactively (i.e. wait till delivered). The **—v** causes **sendmail** to give a short synopsis of what it is doing.

FILES

Except for */usr/lib/sendmail.cf* and *$HOME/.forward*, these pathnames are all specified in */usr/lib/sendmail.cf*. Thus, these values are only approximations.

$HOME/.forward	forwarding address
/usr/lib/aliases	raw data for alias names
/usr/lib/aliases.pag	
/usr/lib/aliases.dir	data base of alias names
/usr/lib/sendmail.cf	configuration file
/usr/lib/sendmail.fc	frozen configuration
/usr/lib/sendmail.hf	help file
/usr/lib/sendmail.st	collected statistics
/usr/bin/uux	to deliver uucp mail
/usr/lib/mail/mh_deliver	to deliver local mail
*/usr/spool/mqueue/**	temp files

DIAGNOSTICS

If there was an error in sending the letter, **sendmail** will either send mail back to the sender, write a message to the user, or exit with a status (depending on configuration and flags).

VARIABLES

HOME	The user's home directory. Used to find the .forward file.
NAME	Full name placed on outgoing mail

RETURN VALUE

Sendmail returns an exit status describing what it did. The codes are defined in ⟨*sysexits.h*⟩

EX_OK	Successful completion on all addresses.
EX_NOUSER	User name not recognized.
EX_UNAVAILABLE	Catchall meaning necessary resources were not available.
EX_SYNTAX	Syntax error in address.
EX_SOFTWARE	Internal software error, including bad arguments.
EX_OSERR	Temporary operating system error, such as "cannot fork".
EX_NOHOST	Host name not recognized.
EX_TEMPFAIL	Message could not be sent immediately, but was queued.

CAVEATS

Sendmail converts blanks in addresses to dots. This is incorrect according to the old ARPANET mail protocol RFC733 (NIC 41952), but is consistent with the new protocols (RFC822).

SEE ALSO

mail(1mh), mailaddr(7).

NAME

shutdown — close down the system at a given time

SYNOPSIS

/etc/shutdown [**—h**] [**—k**] [**—r**] *time* [*warning-message* ...]

DESCRIPTION

Shutdown provides an automated shutdown procedure which a super–user can use to notify users nicely when the system is shutting down.

The preferred way to bring the system down is to turn off the power switch. This will initiate **shutdown**. When **shutdown** exits, the system will go through an automated process of checking the file systems (see fsck(8)). If the file systems are healthy, a file named */fastboot* is created. The existence of this file signals the system at boot time that an additional file system check need not be done at boot time and the system will boot in much less time.

Time is the time at which **shutdown** will bring the system down and may be the word *now* (indicating an immediate shutdown) or specify a future time in one of two formats: +number and hour:min. The first form brings the system down in *number* minutes and the second brings the system down at the time of day indicated (as a 24—hour clock).

At intervals which get closer together as shutdown time approaches, *warning-messages* are displayed at the terminals of all users on the system. Five minutes before shutdown, or immediately if shutdown is in less than 5 minutes, logins are disabled by creating */etc/nologin* and writing a message there. If this file exists when a user attempts to log in, *login(1)* prints its contents and exits. The file is removed just before **shutdown** exits.

At shutdown time a message is written in the file */usr/adm/shutdownlog*, containing the time of shutdown, who ran shutdown and the reason. Then a terminate signal is sent at *init* to bring the system down to single–user state.

The time of the shutdown and the warning message are placed in */etc/nologin* and should be used to inform the users about when the system will be back up and why it is going down (or anything else).

OPTIONS

—h **Shutdown** will exec *halt(8)*.

—k **Shutdown** will not shut down the system (**—k** is to make users *think* the system is going down).

—r *Shutdown* will exec *reboot(8)*.

FILES

/etc/nologin	tells login not to let anyone log in
/usr/adm/shutdownlog	log file for successful shutdowns.
/fastboot	created during shutdown procedure after successful disk check

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Shutdown only allows you to kill the system between now and 23:59 if you use the absolute time for shutdown.

SEE ALSO

login(1), fsck(8), reboot(8).

NAME

swapon — specify additional device for paging and swapping

SYNOPSIS

/etc/swapon [-a] [-q]
/etc/swapon name ...

DESCRIPTION

Swapon is used to specify additional devices on which paging and swapping are to take place. The system begins by swapping and paging on only a single device so that only one disk is required at bootstrap time. Calls to **swapon** normally occur in the system multi-user initialization file */etc/rc* making all swap devices available, so that the paging and swapping activity is interleaved across several devices.

In the second form, *name* is individual block devices as given in the system swap configuration table. The call makes only this space available to the system for swap allocation.

OPTIONS

—a This option causes all devices marked as *sw* swap devices in */etc/fstab* to be made available.

—q This option prevents warning messages from being printed if a swap device in */etc/fstab* cannot be made available.

FILES

/etc/fstab

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

There is no way to stop paging and swapping on a device. It is therefore not possible to make use of devices which may be dismounted during system operation.

SEE ALSO

swapon(2), fstab(5), init(8).

NAME

sync — update the super block

SYNOPSIS

sync

DESCRIPTION

Sync executes the **sync** system primitive. **Sync** can be called to insure all disk writes have been completed before the processor is halted in a way not suitably done by *reboot(8)* or *halt(8)* .

See *sync(2)* for details on the system primitive.

SEE ALSO

sync(2), fsync(2), halt(8), reboot(8), update(8).

NAME

sysadmin — system administration interface

SYNOPSIS

/etc/sysadmin [*—n*]

DESCRIPTION

The command **sysadmin** executes the System Administration package found in */usr/sysadmin*. This package allows easy access to system configuration files and simple ways to perform many of the tasks required of the system administrator.

Without the **—n** option, the system administrator password is requested, since the user must become the superuser to do any real work.

With the **—n** option, the package is executed without a change to the user. This provides a way to view the menus without being able to make any changes.

Some universal commands are provided at menu choice. Documentation is provided by the package by typing **H**, **h**, or **?**. Typing (CNTRL–L) redraws the screen. Typing *!* followed by a command will execute the command. After execution, the screen may have to be redrawn. All other information is contained in the help mode provided by the package.

FILES

/usr/sysadmin	System Administrator's home directory

CAVEATS

The userid for sysadmin is 0, which is the same userid as root. It is very important to make sure that the sysadmin account has a password. You may choose to use the same password for root and sysadmin, a different password, or a '*'. Setting up the sysadmin account with a password of '*' will make it so that only the superuser can execute **sysadmin.**

Sysadmin will not work on a screen that is smaller than 23 lines by 79 columns.

Sysadmin uses *curses(3t),* so Tektronix 4025 terminals require the termcap entry 4025–cr.

SEE ALSO

vipw(8).

NAME

sysconf — system configuration interface

SYNOPSIS

sysconf [*—d devicename*] [*—s sysin*] [*—o sysout*]

DESCRIPTION

The System Configuration package is supplied as a separate package. The package must be installed to use; see *sysadmin(8)*. It consists of a menu program **sysconf** , pre–linked kernel objects, a device driver library, a directory of device description files, an assembler and a linker.

Sysconf is a menu–driven program which provides menus to set parameters and enable devices. The option to generate a kernel uses the selected driver information and parameter settings to generate an assembler file, *param.s*. A shell script *ld_kernel* is executed to assemble *param.s* and link the kernel with a pre–linked kernel object and device library. A system definition file, *sysdef(5),* is created showing the active device drivers supported and current parameter settings. A file, **MAKEDEV,** is created containing rules for making the special device files for active device drivers.

OPTIONS

—d devicename
Specifies a directory containing the device description files. The default directory is *./descrip*

—s sysin
Specifies a system definition file to use for default input. The default is to set all installed devices active and use default values for parameters.

—o sysout
Specifies an output file for the system definition file. The default is *./sysdef*.

System Configuration means device drivers and system parameters can be configured for a given application. System Configuration consists of two facets: device configuration and parameter tuning. An option also exists to specify an alternative pre–linked kernel object.

Each device which may be attached to a workstation needs a device driver to allow useage of the device. Drivers are normally linked into the kernel. Any driver linked into the kernel is considered active. If the driver is not active, the given device can not be used and the memory normally taken by the driver is available for use by user processes. Some drivers require static buffers which consume memory; this memory is also available if the driver is not active. A driver may provide support for more than one device; one device is considered the real device and other devices are aliases for the real device. Device configuration is based on drivers not on devices; if the driver is active, support is available for the real and alias devices. All device information is obtained from device description files(see *devdes(5)*). **Sysconf** depends on the these files for all

information dealing with device drivers. They should not be lightly modified. **Sysconf** provides a menu to allow the enabling and disabling of drivers. The default device configuration is obtained by checking the workstation for devices currently installed; drivers for all on–board devices and any installed devices are considered active.

Parameter tuning allows the sizes of Utek internal tables to be adjusted and allows the setting of timezone parameters. The default values for parameters are based on whether networking is to be used and the maximum load factor. The maximum load factor is average amount of entries needed in Utek tables to support work by x user processes. Parameter tuning is done in six areas: timezone setting; process limits; file I/O; general I/O; mass storage; and dynamically set parameters. Timezone setting consists of setting the minutes west of Greenwich and specifying the type of daylight saving's time to be used. Process limits are the maximum number of processes allowed, and the number of segments allowed. For the enhanced virtual memory kernel, the number of memory maps and text segments may be set. File I/O parameters consist of the number of change directories allowed using the Distributed File System; the number of inodes available and the number of total open files allowed. General I/O parameters are number of terminal character lists; number of message buffers; and number of entries in the timeout queue. Mass storage parameter tuning allows specification of the root, dump and argument devices. The dynamically set parameters deal with memory allocation for page buffers, the number of buffer headers and number of swap buffers.

FILES

/usr/sys/conf/sysconf	System Configuration program
/usr/sys/conf/descrip	Directory of Device Description files
/usr/sys/conf/lib6?00.a	Device Driver Library
/usr/sys/conf/.o*	pre–linked kernel object
sysdef	System Definition file
MAKEDEV	Shell script to make special devices for active drivers

SEE ALSO

devdes(5), sysdef(5), sysadmin(8), as(1), ld(1).

NAME

syslog — log systems messages

SYNOPSIS

/etc/syslog [**—d**] [**—f***filename*] [**—m***N*]

DESCRIPTION

Syslog reads a datagram socket and logs each line it reads into a set of files described by the configuration file */etc/syslog.conf*. **Syslog** configures when it starts up and whenever it receives a hangup signal.

Each message is one line. A message can contain a priority code, marked by a digit in angle braces at the beginning of the line. Priorities are defined in ⟨*syslog.h*⟩, as follows:

LOG_ALERT — Priority 1.
This priority should essentially never be used. It applies only to messages that are so important that every user should be aware of them, e.g., a serious hardware failure.

LOG_SALERT — Priority 2.
Messages of this priority should be issued only when immediate attention is needed by a qualified system person, e.g., when some valuable system resource dissappears. They get sent to a list of system people.

LOG_EMERG — Priority 3.
Emergency messages are not sent to users, but represent major conditions. An example might be hard disk failures. These could be logged in a separate file so that critical conditions could be easily scanned.

LOG_ERR — Priority 4.
These represent error conditions, such as soft disk failures, etc.

LOG_CRIT — Priority 5.
Such messages contain critical information, but which can not be classed as errors, for example, *su* attempts. Messages of this priority and higher are typically logged on the system console.

LOG_WARNING — Priority 6.
Issued when an abnormal condition has been detected, but recovery can take place.

LOG_NOTICE — Priority 7
Something that falls in the class of "important information"; this class is informational but important enough that you don't want to throw it away casually. Messages without any priority assigned to them are typically mapped into this priority.

LOG_INFO Priority 8.
Information level messages. These messages could be thrown away without problems, but should be included if you want to keep a close watch on your system.

LOG_DEBUG Priority 9.
It may be useful to log certain debugging information. Normally this will be thrown away.

It is expected that the kernel will not log anything below LOG_ERR priority.

The configuration file is in two sections separated by a blank line. The first section defines files that **syslog** will log into. Each line contains a single digit which defines the lowest priority (highest numbered priority) that this file will receive, an optional asterisk which guarantees that something gets output at least every 20 minutes, and a pathname. The second part of the file contains a list of users that will be informed on SALERT level messages.

To bring **syslog** down, it should be sent a terminate signal. It logs that it is going down and then waits approximately 30 seconds for any additional messages to come in.

There are some special messages that cause control functions. ⟨*⟩N sets the default message priority to *N*. ⟨$⟩ causes **syslog** to reconfigure (equivalent to a hangup signal). This can be used in a shell file run automatically early in the morning to truncate the log.

Syslog creates the file */etc/syslog.pid*, containing a single line with its process ID. This can be used to kill or reconfigure **syslog**.

OPTIONS

—d Turn on debugging (if compiled in).

—f*name*
Specify an alternate configuration file.

—m*N*
Set the mark interval to *N* (default 20 minutes).

EXAMPLES

The configuration file:

```
5*/dev/console
8/usr/adm/syslog
3/usr/adm/critical

eric
kridle
kalash
```

logs all messages of priority 5 or higher onto the system console, including timing marks every 20 minutes; all messages of priority 8 or higher into the file */usr/adm/syslog*; and all messages of priority 3 or higher into */usr/adm/critical*. The users "eric", "kridle", and "kalash" will be informed on any subalert messages.

FILES

/etc/syslog.conf	the configuration file
/etc/syslog.pid	the process ID

CAVEATS

LOG_ALERT and LOG_SUBALERT messages should only be allowed to privileged programs.

Actually, **syslog** is not clever enough to deal with kernel error messages in the current implementation.

SEE ALSO

syslog(3c).

NAME

tcpd — Tcp master server

SYNOPSIS

/etc/tcpd [**—v**] [**—f** *confile*] [**—p***serverdir*]

DESCRIPTION

Tcpd is the master network server. This server acts as a surrogate for network server programs. It opens sockets for each service and waits for a connection to arrive on one of them. The appropriate server is then started with the new connection on the standard input and output. The standard error is connected to *syslog(8)*.

Tcpd is used instead of having each server be a daemon, because this reduces the number of processes (which improves performance).

The list of network servers/services is given in the configuration file; default is */etc/tcp_servers* (see *tcp_servers(5n)* for description of format.

Tcpd checks to see if the configuration file has changed (every minute), and reconfigures itself if it has. Sending it a HUP signal will cause it to reconfigure immediately.

OPTIONS

—p*serverdir*
The daemon changes directory to *serverdir* starting up; [default is */usr/lib/tcp_servers*]

—f*confile*
Use file *confile* instead of the usual configuration file */etc/tcp_servers*.

—v Print out the name of all services which are provided when starting up. (Used by */etc/rc.net*).

RETURN VALUE

[USAGE]	Incorrect command line syntax. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.

CAVEATS

The number of servers is limited to 26. The max number of arguments to a server is limited to 64.

There is no way to set options before the accept. If the server wants an option (such as **SO_KEEPALIVE**), it should do it itself.

SEE ALSO

services(5n), ftpd(8n), rexecd(8n), rlogind(8n), rshd(8n), syslog(8), telnetd(8n).

NAME

telnetd — DARPA TELNET protocol server

SYNOPSIS

/etc/tcp_services/telnetd [**—d**] [*port*]

DESCRIPTION

Telnetd is a server which supports the DARPA standard TELNET virtual terminal protocol. The TELNET server process is started (by *tcpd(8n)*) when a connection is made at the port indicated in the TELNET service description; see *services(5n)*. This port number may be overridden (for debugging purposes) by specifying a port number on the command line.

Telnetd operates by allocating a pseudo–terminal device for a client, then creating a login process which has the slave side of the pseudo–terminal as *stdin*, *stdout*, and *stderr*. **Telnetd** manipulates the master side of the pseudo terminal, implementing the TELNET protocol and passing characters between the client and login process.

When a TELNET session is started up, **telnetd** sends a TELNET option to the client side indicating a willingness to do "remote echo" of characters. The pseudo terminal allocated to the client is configured to operate in "cooked" mode, and with XTABS and CRMOD enabled (see *tty(4)*). Aside from this initial setup, the only mode changes **telnetd** will carry out are those required for echoing characters at the client side of the connection.

Telnetd supports binary mode, and most of the common TELNET options, but does not, for instance, support timing marks. Consult the source code for an exact list of which options are not implemented.

OPTIONS

—d Causes each socket created by **telnetd** to have debugging enabled (see SO_DEBUG in *socket(2)*).

RETURN VALUE

[0] **Telnetd** is running.

[1] **Telnetd** is not running.

SEE ALSO

telnet(1n), tcpd(8n).

NAME

tftpd — DARPA Trivial File Transfer Protocol server

SYNOPSIS

/etc/tftpd [**—d**] [*port*]

DESCRIPTION

Tftpd is a server which supports the DARPA Trivial File Transfer Protocol. The TFTP server operates at the port indicated in the TFTP service description; see *services(5n)*. This port number may be overridden (for debugging purposes) by specifying a port number on the command line. If the **—d** option is specified, each socket created by **tftpd** will have debugging enabled (see SO_DEBUG in *socket(2)*).

The use of **tftp** does not require an account or password on the remote system. Due to the lack of authentication information, **tftpd** will allow only publicly readable files to be accessed. Note that this extends the concept of "public" to include all users on all hosts that can be reached through the network; this may not be appropriate on all systems, and its implications should be considered before enabling **tftp** service.

OPTIONS

—d Causes each socket created by **tftpd** to have debugging enabled (see SO_DEBUG in *socket(2)*).

port
Use this port instead of the one in *services(5n)*.

RETURN VALUE

[0] **Tftpd** is running.

[1] **Tftpd** is not running.

[3] System socket error

CAVEATS

This server is known only to be self consistent (i.e. it operates with the user TFTP program, **tftp**).

The search permissions of the directories leading to the files accessed are not checked.

SEE ALSO

services(5n).

NAME

timed — time daemon

SYNOPSIS

/etc/timed [**—a**] [**—b**] [**—d**] [**—f** *logfile*] [**—t** *minutes*] *remotehost*

DESCRIPTION

Timed is a network server that implements a simple time synchronization service.

The local host's time is compared to the time of the *remotehost* every 60 minutes. If the times differ, the local host's time will be synchronized to the time of the *remotehost.*

OPTIONS

—a Get the remote host's time by using the **udp** service **time** instead of the default service of **utime.**

—b Boottime; invoke **timed** as part of the booting process.

—d Debugging; don't spawn a new process and don't disconnect the tty.

—f *logfile*
Instead of writing error messages to *syslog(8)*, write them to *logfile.*

—t *minutes*
Synchronize clocks every *minutes* minutes instead of the default of every 60 minutes.

RETURN VALUE

[USAGE]	Incorrect command line syntax. Execution terminated.
[NO_ERRS]	Command completed without error.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

SEE ALSO

adjtime(2), udpd(8n).

NAME

tunefs — tune up an existing file system

SYNOPSIS

/etc/tunefs *tuneup-options special* ¦ *filesystem*

DESCRIPTION

Tunefs is designed to change the dynamic parameters of a file system which affect the layout policies. The parameters which are to be changed are indicated by the options given below.

The file system may be specified by giving *special*, the name of the special file (device) on which the file system is mounted, or by specifying *filesystem*, the name of the root directory of the file system.

OPTIONS

Tuneup *options* are:

—a *maxcontig*
This specifies the maximum number of contiguous blocks that will be laid out before forcing a rotational delay (see **—d** below). The default value is one, since most device drivers require an interrupt per disk transfer. Device drivers that can chain several buffers together in a single transfer should set this to the maximum chain length.

—d *rotdelay*
This specifies the expected time (in milliseconds) to service a transfer completion interrupt and initiate a new transfer on the same disk. It is used to decide how much rotational spacing to place between successive blocks in a file.

—e *maxbpg*
This indicates the maximum number of blocks any single file can allocate out of a cylinder group before it is forced to begin allocating blocks from another cylinder group. Typically this value is set to about one quarter of the total blocks in a cylinder group. The intent is to prevent any single file from using up all the blocks in a single cylinder group, thus degrading access times for all files subsequently allocated in that cylinder group. The effect of this limit is to cause big files to do long seeks more frequently than if they were allowed to allocate all the blocks in a cylinder group before seeking elsewhere. For file systems with exclusively large files, this parameter should be set higher.

—m *minfree*
This value specifies the percentage of space held back from normal users; the minimum free space threshold. The default value used is 10%. This value can be set to zero, however up to a factor of three in throughput will be lost over the performance obtained at a 10% threshold. Note that if the value is raised above the current usage level, users will be unable to allocate files until enough files have been deleted to get under the higher threshold.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

This program should work on mounted and active file systems. Because the super-block is not kept in the buffer cache, the program will only take effect if it is run on dismounted file systems. (If run on the root file system, the system must be rebooted.)

SEE ALSO

fs(5), newfs(8), mkfs(8).

NAME

udpd — udp misc server

SYNOPSIS

/etc/udpd [**—d**] [**—f**conFile]

DESCRIPTION

Udpd is a network server which implements several simple network services. The services are:

echo
Echo data back to the sender's address.

discard
Drop the packet.

time
Send back the time of day in seconds since 1 Jan 1900 midnight. This is different from other UTek time functions which return time since 1 Jan 1970. This is a 32-bit number in network byte order.

daytime
Send back the time of day in human readable form.

chargen
Send back a random length character pattern.

systat
Send back a list of user's currently on the system (maybe more than one packet).

tekup
Send back a string similar to the output of *uptime(1n)* and *w(1)*. This function is used by the ECS version of the **uptime** command.

OPTIONS

—d Debugging; don't spawn a new process and don't disconnect the tty.

—f*logfile*
Instead of writing error messages to *syslog(8)*, write them to *logfile*.

RETURN VALUE

0 – daemon started okay

1 – errors prevented starting daemon

SEE ALSO

uptime(1n), services(5n), syslog(8).

NAME

mount, umount — mount and dismount file system

SYNOPSIS

/etc/mount [**—f**] [**—r**] [**—v**] [*special*] *name*

/etc/mount —a

/etc/umount [**—v**] *special*

/etc/umount —a

DESCRIPTION

Mount announces to the system that a removable file system is present on the device *special*. The file *name* must exist already; it must be a directory (unless the root of the mounted file system is not a directory). It becomes the name of the newly mounted root.

If only *name* is given without *special*, *name* must be an entry in the file */etc/fstab* (see *fstab(5)*).

Umount announces to the system that the removable file system previously mounted on device *special* is to be removed.

These commands maintain and update a table of mounted devices in */etc/mtab*. If invoked without an argument, **mount** prints the table.

Mount and **umount** must be run by the super–user.

OPTIONS

—a All of the file systems described in */etc/fstab* are attempted to be mounted or unmounted. In this case, *special* and *name* are taken from */etc/fstab*. The *special* file name from */etc/fstab* is the block special name.

—f If invoked with this option, **mount** will not actually mount any file systems, but */etc/mtab* will be updated as if it had.

—r Indicates to **mount** that the file system is to be mounted read–only.

—v This option will cause **mount** or **umount** to print its actions as it executes.

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.

FILES

/etc/mtab	mount table
/etc/fstab	file system table

DIAGNOSTICS

Not owner
The caller is not the super–user.

Permission denied
The file */etc/mtab* could not be updated.

RETURN VALUE

For both **mount** and **umount**:

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

For *umount*:

[NP_WARN]	An error warranting a warning message occurred. Execution continues.

CAVEATS

Mounting file systems full of garbage will crash the system.

Mounting a root directory on a non–directory makes some apparently good pathnames invalid.

SEE ALSO

mount(2), fstab(5), mtab(5).

NAME

update — periodically update the super block

SYNOPSIS

/etc/update

DESCRIPTION

Update is a program that executes the *sync(2)* primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file.

SEE ALSO

sync(2), init(8), rc(8), sync(8).

NAME

uuclean — uucp spool directory clean–up

SYNOPSIS

uuclean [*option*] ...

DESCRIPTION

Uuclean will scan the spool directory for files with the specified prefix and delete all those which are older than the specified number of hours.

This program will typically be started by *cron(8)*.

OPTIONS

—p*pre*
Scan for files with *pre* as the file prefix. Up to 10 **—p** arguments may be specified. A **—p** without any *pre* following will cause all files older than the specified time to be deleted.

—n*time*
Files whose age is more than *time* hours will be deleted if the prefix test is satisfied. (default time is 72 hours)

—m
Send mail to the owner of the file when it is deleted.

FILES

/usr/lib/uucp	directory with commands used by uuclean internally
/usr/lib/uucp/spool	spool directory

RETURN VALUE

[0] No errors were encountered.

[nonzero]
Errors were encountered.

SEE ALSO

uucp(1n), uux(1n).

NAME

uusnap — show snapshot of the UUCP system

SYNOPSIS

uusnap

DESCRIPTION

Uusnap displays in tabular format a synopsis of the current UUCP situation. The format of each line is as follows:

```
site    N Cmds    N Data    N Xqts    Message
```

Where *site* is the name of the site with work, *N* is a count of each of the three possible types of work (*command, data,* or *remote execute*), and *Message* is the current status message for that site as found in the STST file.

Included in *Message* may be the time left before UUCP can re-try the call, and the count of the number of times that UUCP has tried to reach the site.

RETURN VALUE

[0] No errors encountered.

[nonzero]
Errors encountered.

SEE ALSO

uucp(1n).

NAME

vidfs — edit and/or check the dfs access permissions file

SYNOPSIS

vidfs [**—c** [*filename* ...]]

DESCRIPTION

With no arguments, **vidfs** locks the file */etc/hosts.dfs.access,* copies its contents to a temporary file, and invokes the editor (default = **vi**) on the temporary file. After the editor is exited, the modified data is checked as described below. If no problems are found, the temporary file replaces the old access permissions file, which is unlocked. If only problems in the *Warning* category are found, the user may re-edit the file, quit without updating, or update the access permissions file, ignoring the warnings. If any problems in the *ERROR* category are found, the user may re-edit the file or quit without updating.

If called with the **—c** option, **vidfs** will check the named files as described below. If no names are given, */etc/hosts.dfs.access* is checked.

File Checking

The following problems are considered *Warnings*. When they occur, a message is printed giving the line number and a description of the problem.

1 Duplicate entries

2 Host does not respond

3 Access by root permitted

The following problems are considered *ERRORs*. When they occur, a message is printed giving the line number and a description of the problem.

1 User not found in the password file

2 Ambiguous entry (an incoming request is to be assigned rights of more than one local user)

OPTIONS

—c	Check the named files (default is */etc/hosts.dfs.access*). Do not edit the access permissions file.

FILES

/etc/hosts.dfs.access	The access permissions file to edit or the default file to check.
/etc/dfstmp	The temporary edit file.

VARIABLES

EDIT	The editor to be used instead of **vi**.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.

[1]	The file(s) contained errors.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

vidfs can only be used by root to edit the access permissions file (any user can use **vidfs** with the **—c** option).

SEE ALSO

hosts.dfs.access(5n),

NAME

vipw, pwck — edit and/or check the password file

SYNOPSIS

vipw [**—c** [*filename* ...]]
pwck [**—c**] [*filename* ...]

DESCRIPTION

With no arguments, **vipw** locks the file */etc/passwd,* copies its contents to a temporary file, and invokes the editor (default = **vi**) on the temporary file. After the editor is exited, the modified data is checked as described below. If no problems are found, the temporary file replaces the old password file, which is unlocked. If only problems in the *Warning* category are found, the user may re–edit the file, quit without updating, or update the password file, ignoring the warnings. If any problems in the *ERROR* category are found, the user may re–edit the file or quit without updating.

If called as **pwck** or with the **—c** option, **vipw** will check the named files as described below. If no names are given, */etc/passwd* is checked.

File Checking

The following problems are considered *Warnings*. When they occur, a message is printed giving the line number and a description of the problem.

1 User name longer than 8 characters

2 User name begins with non–alphabetic character

3 User name contains characters other than a–z, A–Z, 0–9, –, and _

4 The home directory does not exist or is not a directory

These problems are considered *ERRORs*. When they occur, a message is printed giving the line number and a description of the problem.

1 Not enough fields.

2 Too many fields.

3 Entry longer than 1024 characters.

4 Non–numeric or empty user id field.

5 Non–numeric or empty group id field.

6 The shell program does not exist or is not executable.

OPTIONS

—c Check the named files (default is */etc/passwd*). Do not edit the password file.

FILES

/etc/passwd	The password file to edit or the default file to check.
/etc/ptmp	The temporary edit file.

VARIABLES

EDIT	The editor to be used instead of **vi**.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[1]	The file(s) contained errors.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Vipw can only be used by root to edit the password file (any user can use **pwck** and **vipw** with the **—c** option). Its use should be restricted to major modifications and corrections. Make normal changes with *chfn(1), chsh(1), passwd(1), and* other utilities.

The earlier versions of this utility were shell scripts. It is imperative that the earlier versions not be used in this system. Locking is now handled by the kernel, whereas the shell scripts used a lock file.

SEE ALSO

chfn(1), chsh(1), passwd(1), passwd(5).

NAME

viwsb — edit and/or check the dumptable file

SYNOPSIS

/etc/viwsb [**—c** [*filename ...*]]

DESCRIPTION

With no arguments, **viwsb** locks the file */etc/dumptable,* copies its contents to a temporary file, and invokes the editor (default = **vi**) on the temporary file. After the editor is exited, the modified data is checked as described below. If no problems are found, the temporary file replaces the old dumptable file, which is unlocked. If only problems in the *Warning* category are found, the user may re-edit the file, quit without updating, or update the dumptable file, ignoring the warnings. If any problems in the *ERROR* category are found, the user may re-edit the file or quit without updating.

If called with the **—c** option, **viwsb** will check the named files as described below. If no names are given, */etc/dumptable* is checked.

File Checking

The following problems are considered *Warnings*. When they occur, a message is printed giving the line number and a description of the problem.

1 Workstation name longer than 32 characters

2 Workstation name begins with non-alphabetic character

3 Workstation name contains characters other than a-z, A-Z, 0-9, -, and _

4 Dump level mask longer than 10 digits

5 More than one group designator

The following problems are considered *ERRORs*. When they occur, a message is printed giving the line number and a description of the problem.

1 Incorrect number of fields.

2 Entry longer than 1024 characters.

3 Empty file system field.

4 Non-numeric or empty dump level field.

5 Non-numeric or empty frequency field.

6 Non-numeric or empty group load field.

7 Non-alphabetic or empty group field.

OPTIONS

—c Check the named files (default is */etc/dumptable*). Do not edit the dumptable file.

FILES

/etc/dumptable	The dumptable file to edit or the default file to check.
/etc/wsbtmp	The temporary edit file.

VARIABLES

EDIT	The editor to be used instead of **vi**.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[1]	The file(s) contained errors.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

CAVEATS

Viwsb can only be used by root to edit the dumptable file (any user can use **viwsb** with the **—c** option).

SEE ALSO

wsdumptable(5n), wsdump(8n), wsrestore(8n)

NAME

wsass — wsdump assign utility

SYNOPSIS

/etc/wsass [**—a**] [**—d**] [**—r**] *device*

DESCRIPTION

Wsass is a utility used by the workstation dump and restore programs in assigning and rewinding a magnetic tape drive. **Wsass** calls *assign(1), deassign(1)* and *mt(1)* to assign, deassign and rewind the device. **Wsass** is needed to locally handle the device which is owned by the remote process running under *wsdump(8n).*

OPTIONS

—a Assign the device.

—d Deassign the device.

—r Rewind the device.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

SEE ALSO

assign(1), deassign(1), mt(1), wsdump(8n), wsrestore(8n).

NAME

wsdump — multi-workstation dump utility

SYNOPSIS

/etc/wsdump [**—l** *level*] [**—g** *group*] [**—w** *workstation* [**—f** *filesystem*]] [**—s** *size*] [**—d** *density*] [**—F**] [**—S**] *target-media*

DESCRIPTION

Wsdump is a multi-workstation dump which runs in conjunction with *rdump(8n).*

OPTIONS

—l *level* This option specifies the 'dump level'. All files modified since the last date stored in the file */etc/dumpdates* on the workstation being dumped for the same filesystem at lesser levels will be dumped. The **—l** option must be specified if the **—g** or **—w** option is specified.

—g *group*
This option specifies the group of workstations to be dumped. Workstations are placed into groups ranging from *A* to *Z* when they are entered in the file */etc/wsdumptable.*

—w *workstation*
This option specifies the *workstation* to be dumped. If the **—w** option is specified and the **—f** option is not specified, all filesystems that reside on *workstation* and found in the file */etc/wsdumptable* will be dumped.

—f *filesystem*
This option specifies the *filesystem* to be dumped. This option must be used with the **—w** option.

—s *size* The size of the dump media is specified by *size.* When the specified size is reached, **dump** will wait for media to be changed. The default size is 2000 feet for 9 track tapes 360k for floppy disk and 400 feet for cartridge tape.

—d *density*
The density of the tape, expressed in BPI, is taken from *density.* This is used in calculating the amount of tape used per volume. The default is 1600 for 9-track tape, and 8000 for cartridge.

—F Specifies floppy disk backup media (9-track tape is default).

—S Specifies streamer cartridge tape backup media (9-track tape is default).

If no options are given, **wsdump** scans through the file */etc/wsdumptable* and determines which filesystems to dump at which levels. To execute **wsdump** one must be logged in as *dumpopr.*

EXAMPLES

`/etc/wsdump -l 0 -w kokomo -f /dev/dw10a -S /dev/tc`
(dump entire filesystem "/dev/dw10a" on "kokomo" to local tape cartridge device "/dev/tc")

/etc/wsdump -l 0 -w nodename /dev/rmt1
(dump all filesystems on "nodename" found in the file /etc/wsdumptable to the tape device "/dev/rmt1")

/etc/wsdump -l 0 -g A /dev/rmt2
(dump all filesystems in group "A" to the local tape device "/dev/rmt2")

/etc/wsdump /dev/rmt8
(dump all filesystems on all workstations specified in /etc/wsdumptable to the local tape device "/dev/rmt8")

FILES

/etc/wsdumpdates	Dump date record: output file
/etc/wsdumptable	Dump table input file
~/.netrc	Network access file.
/usr/adm/wsdumplast	Dump table for group dumps

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_ERR]	A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.

SEE ALSO

netrc(5n), wsdumptable(5n), rdump(8n), rrestore(8n), wsass(8n), wsrestore(8n).

NAME

wsrestore — multi-workstation restore utility

SYNOPSIS

/etc/wsrestore [**—w** *workstation*] [**—f** *filesystem*] [**—t** *restore device*] [**—m** *modification date*] [**—l** *last access date*] [**—d** *dump date and time*] [**—r** *restore directory*] [**—o** *restore options*] [**—F**] [**—S**]

DESCRIPTION

Wsrestore is a multi-workstation restore that runs in conjunction with *rrestore(8n)*. **Wsrestore** executes *rrestore* on a remote workstation, allowing that workstation to restore from a dump media on the host server to the workstation's storage device.

OPTIONS

—w *workstation*

This option specifies the *workstation* to be restored. This option is required. If it is not entered on the command line, then **wsrestore** will prompt for it.

—f *filesystem*

This option specifies the *filesystem* to be restored. This option is required. If it is not entered on the command line, then **wsrestore** will prompt for it.

—t *restore device*

This option specifies the *restore device* to be used in restoring archived files. The path must be a complete path specifying the device as a floppy, a cartridge tape, a 9 track tape, or a disk file. The raw device should be specified and no rewind should be used for 9 track or cartridge tape (e.g. /dev/nrmt8). Default devices are used depending on the media specified (see **—F** and **—S**).

—m *modification date*

This option specifies the *modification date* of the file to be restored. This date is used to scan the */etc/wsdumpdates* file to locate the proper volume to mount for the restore. This option must be entered along with **—l** unless the **—d** option has been used. If it is not entered on the command line, then **wsrestore** will prompt for it.

—l *last access date*

This option specifies the *last access date* of the file to be restored. This date is used to scan the */etc/wsdumpdates* file to locate the proper volume to mount for the restore. This option must be entered along with **—m** unless the **—d** option has been used. If it is not entered on the command line, then **wsrestore** will prompt for it.

—d *dump date*

This option specifies the *dump date* of the volume to be restored. The dump date must be entered as date and time exactly to

match the volume required as listed in the */etc/wsdumpdates* file. If this option is used, —**m** and —l are unnecessary.

—**r** *restore directory*
This option specifies the *restore directory* on the workstation where files are to be restored. The default directory is */usr/dumpopr/restore*.

—**o** *restore options*
This option specifies the additional *restore options* to be used. The default option is —*i* for interactive restoration.

—**F** Specifies floppy disk backup media (9–track tape is default).

—**S** Specifies streamer cartridge tape backup media (9–track tape is default).

If no options are given, **wsrestore** will prompt for all information to perform the restore. You must be logged in as *dumpopr* to execute **wsrestore**.

EXAMPLES

`/etc/wsrestore -w dolphin -f /dev/dw00a -t /dev/nrmt7`
(interactive restore of filesystem "/dev/dw00a" on "dolphin" from 9–track tape device "/dev/nrmt7". Extracted files are placed in /usr/dumpopr/restore on "dolphin".)

`/etc/wsrestore -S -t /dev/ntc -r /tmp`
(interactive restore from local tape cartridge device "/dev/ntc". Workstation and filesystem are prompted for by **wsrestore**. Extracted files are placed in /tmp on specified workstation.)

`/etc/wsrestore -w duke -d "08/22/85 03:23:34" -o iv`
(verbose interactive restore of workstation "duke" for dump done on August 22, 1985 at 3:23:34. Restore device and filesystem are prompted for by **wsrestore**.)

FILES

/etc/wsdumpdates	Dump date record – scanned to find volume to mount for restore.
~/.netrc	Network access file.

RETURN VALUE

[NO_ERRS]	Command completed without error.
[USAGE]	Incorrect command line syntax. Execution terminated.
[NP_WARN]	An error warranting a warning message occurred. Execution continues.
[NP_ERR]	An error occurred that was not a system error. Execution terminated.
[P_WARN]	A system error occurred. Execution continues. See *intro(2)* for more information on system errors.

[P_ERR] A system error occurred. Execution terminated. See *intro(2)* for more information on system errors.

SEE ALSO

netrc(5n), wsdumptable(5n), rdump(8n), rrestore(8n), wsdump(8n).