68NW9209H47A

SYSTEM V/88 Release 3.2 Programmer's Reference Manual

(Part 1)

MOTOROLA

MOTORO

Motorola welcomes your

Manual Title ____________

Part Number ____________

Your Name ____________

Your Title ____________

Company ____________

Address ____________

____

____

General Information:

Do you read this manu

☐ Install the product

☐ Reference inforn

In general, how do you

☐ Index ☐ Table o

Completeness: ☐ Ex

What topic would you

____

____

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 2565 PHOENIX, ARIZONA

POSTAGE WILL BE PAID BY ADDRESSEE

MOTOROLA INC.

Computer Group, Microcomputer Division, DW164

2900 South Diablo Way

Tempe, AZ 85282-9741

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

Presentation: ☐ Excellent ☐ Very Good ☐ Good ☐ Fair ☐ Poor

What features of the manual are most useful (tables, figures, appendixes, index, etc.)?

Is the information easy to understand? ☐ Yes ☐ No If you checked no, please explain:

Is the information easy to find? ☐ Yes ☐ No If you checked no, please explain:

Technical Accuracy: ☐ Excellent ☐ Very Good ☐ Good ☐ Fair ☐ Poor

If you have found technical or typographical errors, please list them here.

Page Number	Description of Error

SYSTEM V/88 Release 3.2

Programmer's

Reference Manual

Part 1

(68NW9209H47A)

PREFACE

The *SYSTEM V/88 Release 3.2 Programmer's Reference Manual, Part 1* is for programmers and technical personnel who have a general familiarity with the SYSTEM V/88 operating system.

SYSTEM V/88 Release 3.2 is based on the AT&T UNIX System V Release 3.2.
SYSTEM V/68 Release 3 is based on the AT&T UNIX System V Release 3.

First Edition February 1990

CONTENTS (Part 1)

2. System Calls

3. Subroutines

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 returned value. This is almost always –1 or the NULL pointer; the individual descriptions specify the details. An error number is also made available in the external variable *errno*. *Errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

Each system call description attempts to list all possible error numbers. The following is a complete list of the error numbers and their names as defined in **<errno.h>**.

1 EPERM Not owner

Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or superuser. It is also returned for attempts by ordinary users to do things allowed only to the superuser.

2 ENOENT No such file or directory

This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process

No process can be found corresponding to that specified by *pid* in *kill*(2) or *ptrace*(2).

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 has occurred. This error may in some cases occur on a call following the one to which it actually applies.

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

7 E2BIG Arg list too long
: An argument list longer than 5,120 bytes is presented to a member of the *exec*(2) family.

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

9 EBADF Bad file number
: Either a file descriptor refers to no open file, or a *read*(2) [respectively, *write*(2)] request is made to a file which is open only for writing (respectively, reading).

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

11 EAGAIN No more processes
: A *fork* failed because the system's process table is full or the user is not allowed to create any more processes. Or a system call failed because of insufficient memory or swap space.

12 ENOMEM Not enough space
: During an *exec*(2), *brk*(2), or *sbrk*(2), a program asks for more space than the system is able to supply. This may not be a temporary condition; the maximum space size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a *fork*(2). If this error occurs on a resource associated with Remote File Sharing (RFS), it indicates a memory depletion wich may be temporary, dependent on system activity at the time the call was invoked.

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

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

15 ENOTBLK Block device required
: A non-block file was mentioned where a block device was required, e.g., in *mount*(2).

16 EBUSY Device or resource busy
: An attempt was made to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable.

17 EEXIST File exists
: An existing file was mentioned in an inappropriate context, e.g., *link*(2).

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

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

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

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

22 EINVAL Invalid argument
: Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal*(2) or *kill*(2); reading or writing a file for which *lseek*(2) has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.

23 ENFILE File table overflow
: The system file table is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
: No process may have more than NOFILES (default 20) descriptors open at a time.

25 ENOTTY Not a character device (or) Not a typewriter
: An attempt was made to *ioctl*(2) a file that is not a special character device.

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

27 EFBIG File too large
: The size of a file exceeded the maximum file size or ULIMIT (see *ulimit*(2)).

28 ENOSPC No space left on device
: During a *write*(2) to an ordinary file, there is no free space left on the device. In *fcntl*(2), the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

29 ESPIPE Illegal seek
: An *lseek*(2) was issued to a pipe.

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

31 EMLINK Too many links
: An attempt to make more than the maximum number of links (1024) to a file.

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

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

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

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

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

37-44 Reserved numbers

45 EDEADLK Deadlock
: A deadlock situation was detected and avoided. This error pertains to file and record locking.

46 ENOLCK No lock
: In *fcntl*(2) the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

60 ENOSTR Not a stream
: A *putmsg*(2) or *getmsg*(2) system call was attempted on a file descriptor that is not a STREAMS device.

62 ETIME Stream ioctl timeout
: The timer set for a STREAMS *ioctl*(2) call has expired. The cause of this error is device specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the *ioctl*(2) operation is indeterminate.

63 ENOSR No stream resources
: During a STREAMS *open*(2), either no STREAMS queues or no STREAMS head data structures were available.

64 ENONET Machine is not on the network
: This error is RFS specific. It occurs when users try to advertise, unadvertise, mount, or unmount remote resources while the machine has not done the proper startup to connect to the network.

65 ENOPKG No package
: This error occurs when users attempt to use a system call from a package which has not been installed.

66 EREMOTE Resource is remote
: This error is RFS specific. It occurs when users try to advertise a resource which is not on the local machine, or try to mount/unmount a device (or pathname) that is on a remote machine.

67 ENOLINK Virtual circuit is gone
: This error is RFS specific. It occurs when the link (virtual circuit) connecting to a remote machine is gone.

68 EADV Advertise error
: This error is RFS specific. It occurs when users try to advertise a resource which has been advertised already, or try to stop the RFS while there are resources still advertised, or try to force unmount a resource when it is still advertised.

69 ESRMNT Srmount error
: This error is RFS specific. It occurs when users try to stop RFS while there are resources still mounted by remote machines.

70 ECOMM Communication error
: This error is RFS specific. It occurs when trying to send messages to remote machines but no virtual circuit can be found.

71 EPROTO Protocol error
: Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.

74 EMULTIHOP Multihop attempted
: This error is RFS specific. It occurs when users try to access remote resources which are not directly accessible.

77 EBADMSG Bad message
: During a *read*(2), *getmsg*(2), or *ioctl*(2) I_RECVFD system call to a STREAMS device, something has come to the head of the queue that can't be processed. That something depends on the system call:
  read(2) - control information or a passed file descriptor.
  getmsg(2) - passed file descriptor.
  ioctl(2) - control or data information.

78 ENAMETOOLONG
: A component of a pathname exceeded NAME_MAX characters or an entire pathname exceeded PATH_MAX characters.

83 ELIBACC Cannot access a needed shared library
: Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and the shared library doesn't exist or the user doesn't have permission to use it.

84 ELIBBAD Accessing a corrupted shared library
: Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and *exec*(2) could not load the shared library. The shared library is probably corrupted.

85 ELIBSCN .lib section in *a.out* corrupted
: Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and there was erroneous data in the .lib section of the *a.out*. The .lib section tells *exec*(2) what shared libraries are needed. The *a.out* is probably corrupted.

86 ELIBMAX Attempting to link in more shared libraries than system limit
: Trying to *exec*(2) an *a.out* that requires more shared libraries (to be linked in) than is allowed on the current configuration of the system. See the System Administrator's Guide.

87 ELIBEXEC Cannot exec a shared library directly
: Trying to *exec*(2) a shared library directly. This is not allowed.

89 ENOSYS System call not supported by the system.

90 ELOOP Unable to parse a symbolic link.

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 1 to 30,000.

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

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

Session ID Each active process is a member of a group which is identified by a positive integer called the Session ID. This ID is the process ID of the session leader. This grouping may be used to terminate or continue certain processes upon termination of one of the processes in the group. See *exit*(2), *signals*(2), and *setsid*(2).

Controlling Process A controlling process is a session leader that has opened a terminal.

Orphaned Process Group An orphaned process group is a process group which has no member whose parent is a member of another process group in the same session. A member of such a group is not allowed to be in a stopped state.

Real User ID and Real Group ID Each user allowed on the system is identified by a positive integer (0 to 65535) called a real user ID.

Each user is also a member of at least one group. A group is identified by a positive integer called a group ID.

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

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

Superuser A process is recognized as a *superuser* process and is granted special privileges, such as immunity from file permissions, if its effective user ID is 0.

Special Processes The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

proc0 is the scheduler. *proc1* is the initialization process (*init*). *proc1* is the ancestor of every other process in the system and is used to control the process structure.

File Descriptor A file descriptor is a small integer used to do I/O on a file. The value of a file descriptor is from 0 to (NOFILES - 1). A process may have no more than NOFILES file descriptors open simultaneously. A file descriptor is returned by system calls, e.g., *open*(2), or *pipe*(2). The file descriptor is used as an argument by calls, e.g., *read*(2), *write*(2), *ioctl*(2), and *close*(2).

Filename Names consisting of 1 to 14 characters may be used to name an ordinary file, special file or directory.

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

Note that it is generally unwise to use *****, **?**, **[**, or **]** as part of filenames because of the special meaning attached to these characters by the shell (see *sh*(1)). Although permitted, the use of unprintable characters in filenames should be avoided.

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 filename.

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.

Unless specifically stated otherwise, the NULL pathname is treated as if it named a nonexistent file.

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

Root Directory and Current Working Directory Each process has associated with it a concept of a **root** directory and a current working directory for the purpose of resolving pathname searches. The **root** directory of a process need not be the **root** directory of the **root** file system.

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

The effective user ID of the process is superuser.

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

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process matches the group of the file and the appropriate access bit of the "group" portion (0070) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the "other" portion (0007) of the file mode is set.

Otherwise, the corresponding permissions are denied.

Message Queue Identifier A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct   ipc_perm msg_perm;
struct msg   *msg_first;
struct msg   *msg_last;
char     msg_pad1[2];
ushort   msg_cbytes;
char     msg_pad2[2];
ushort   msg_qnum;
char     msg_pad3[2];
ushort   msg_qbytes;
pid_t    msg_lspid;
pid_t    msg_lrpid;
time_t   msg_stime;
time_t   msg_susec;
time_t   msg_rtime;
time_t   msg_rusec;
time_t   msg_ctime;
time_t   msg_cusec;
```

msg_perm is an ipc_perm structure that specifies the message operation permission (see below). This structure includes the following members:

```
uid_t  uid;          /* user id */
gid_t  gid;          /* group id */
uid_t  cuid;         /* creator user id */
```

```
    gid_t   cgid;           /* creator group id */
    mode_t  mode;           /* r/w permission */
    char    ipc_pad[2];     /* padding */
    ushort  seq;            /* slot usage sequence # */
    key_t   key;            /* key */
```

msg *msg_first
is a pointer to the first message on the queue.

msg *msg_last
is a pointer to the last message on the queue.

msg_cbytes
is the current number of bytes on the queue.

msg_qnum
is the number of messages currently on the queue.

msg_qbytes
is the maximum number of bytes allowed on the queue.

msg_lspid
is the process id of the last process that performed a *msgsnd* operation.

msg_lrpid
is the process id of the last process that performed a *msgrcv* operation. *

msg_stime
and **msg_susec** are the seconds and microseconds, respectively, of the time of the last *msgsnd* operation.

msg_rtime
and **msg_rusec** are the seconds and microseconds, respectively, of the time of the last *msgrcv* operation.

msg_ctime
and **msg_cusec** are the seconds and microseconds, respectively, of the time of the last *msgctl*(2) operation that changed a member of the above structure.

Message Operation Permissions In the *msgop*(2) and *msgctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed, interpreted as follows:

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others
00002	Write by others

Read and write permissions on a msqid are granted to a process if one or more of the following are true:

The effective user ID of the process is superuser.

The effective user ID of the process matches **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid* and the appropriate bit of the "user" portion (0600) of **msg_perm.mode** is set.

The effective group ID of the process matches **msg_perm.cgid** or **msg_perm.gid** and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.

The appropriate bit of the "other" portion (006) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Semaphore Identifier A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct  ipc_perm sem_perm; /* operation permission struct */
ulong   sem_pad1
ushort  sem_pad2
struct  sem *sem_base;     /* ptr to first semaphore in set */
ushort  sem_nsems;         /* number of sems in set */
time_t  sem_otime;         /* last operation time */
time_t  sem_ousec;
time_t  sem_ctime;              /* last change time */
                                /* Times measured in secs since */
                                /* 00:00:00 GMT, Jan. 1, 1970 */
time_t  sem_cusec;
```

sem_perm is an ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
uid_t   uid;        /* user id */
gid_t   gid;        /* group id */
uid_t   cuid;       /* creator user id */
gid_d   cgid;       /* creator group id */
mode_t  mode;       /* r/a permission */
char    ipc_pad[2]; /* padding */
ushort  seq;        /* slot usage sequence number */
key_t   key;        /* key */
```

sem_nsems
is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1.

sem_otime
and **sem_ousec** are the seconds and microseconds, respectively, of the time of the last *semop*(2) operation.

sem_ctime
and **sem_cusec** are the seconds and microseconds, respectively, of the time of the last *semctl*(2) operation that changed a member of the above structure.

A semaphore is a data structure called *sem* that contains the following members:

```
ushort  semval;     /* semaphore value */
short   sempid;     /* pid of last operation  */
ushort  semncnt;    /* # awaiting semval > cval */
ushort  semzcnt;    /* # awaiting semval = 0 */]
```

semval
is a non-negative integer which is the actual value of the semphore.

sempid
is equal to the process ID of the last process that performed a semaphore operation on this semaphore.

semncnt
is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value.

semzcnt
is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Alter by user
00040	Read by group
00020	Alter by group
00004	Read by others
00002	Alter by others

Read and alter permissions on a semid are granted to a process if one or more of the following are true:

The effective user ID of the process is superuser.

The effective user ID of the process matches **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid* and the appropriate bit of the "user" portion (0600) of **sem_perm.mode** is set.

The effective group ID of the process matches **sem_perm.cgid** or **sem_perm.gid** and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.

The appropriate bit of the "other" portion (006) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Shared Memory Identifier A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. (Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed.) The data structure is referred to as *shmid_ds* and contains the following members:

```
struct  ipc_perm shm_perm; /* operation permission struct */
int     shm_segsz;         /* size of segment */
ulong   shm_pad1;          /* padding */
pid_t   shm_lpid;          /* pid of last operation */
pid_t   shm_cpid;          /* creator pid */
ushort  shm_nattch;        /* number of current attaches */
ushort  shm_cnattch
time_t  shm_atime;         /* last attach time */
time_t  shm_ausec;
time_t  shm_dtime;         /* last detach time */
time_t  shm_dusec;
time_t  shm_ctime;         /* last change time */
time_t  shm_cusec;         /* Times measured in secs since */
                           /* 00:00:00 GMT, Jan. 1, 1970 */
```

shm_perm is an ipc_perm structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
uid_t   cuid;          /* creator user id */
gid_t   cgid;          /* creator group id */
uid_t   uid;           /* user id */
gid_t   gid;           /* group id */
mode_t  mode;          /* r/w permission */
char    ipc_pad[2];    /* padding */
ushort  seq;           /* slot usage sequence # */
key_t   key;           /* key */
```

shm_segsz
specifies the size of the shared memory segment in bytes.

shm_cpid
is the process id of the process that created the shared memory identifier.

shm_lpid
is the process id of the last process that performed a *shmop*(2) operation.

shm_nattch
is the number of processes that currently have this segment attached.

shm_atime
and **shm_ausec** are the seconds and microseconds, respectively, of the time of the last *shmat*(2) operation,

shm_dtime
and **shm_dusec** are the seconds and microseconds, respectively, of the time of the last *shmdt*(2) operation.

shm_ctime
and **shm_ausec** are the seconds and microseconds, respectively, of the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

Shared Memory Operation Permissions In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others
00002	Write by others

Read and write permissions on a shmid are granted to a process if one or more of the following are true:

The effective user ID of the process is superuser.

The effective user ID of the process matches **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.

The effective group ID of the process matches **shm_perm.cgid** or **shm_perm.gid** and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.

The appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

STREAMS A set of kernel mechanisms that support the development of network services and data communication *drivers*. It defines interface standards for character input/output within the kernel and between the kernel and user level processes. The STREAMS mechanism is composed of utility routines, kernel facilities, and a set of data structures.

Stream A stream is a full-duplex data path within the kernel between a user process and driver routines. The primary components are a *stream head*, a *driver* and zero or more *modules* between the *stream head* and *driver*. A *stream* is analogous to a Shell pipeline except that data flow and processing are bidirectional.

Stream Head In a *stream*, the *stream head* is the end of the *stream* that provides the interface between the *stream* and a user process. The principle functions of the *stream head* are processing STREAMS-related system calls, and passing data and information between a user process and the *stream*.

Driver In a *stream*, the *driver* provides the interface between peripheral hardware and the *stream*. A *driver* can also be a pseudo-*driver*, e.g., a *multiplexor* or log *driver* (see *log*(7)), which is not associated with a hardware device.

Module A module is an entity containing processing routines for input and output data. It always exists in the middle of a *stream*, between the stream's head and a *driver*. A *module* is the STREAMS counterpart to the commands in a Shell pipeline except that a module contains a pair of functions which allow independent bidirectional (*downstream* and *upstream*) data flow and processing.

Downstream In a *stream*, the direction from *stream head* to *driver*.

Upstream In a *stream*, the direction from *driver* to *stream head*.

Message In a *stream*, one or more blocks of data or information, with associated STREAMS control structures. *Messages* can be of several defined types, which identify the *message* contents. *Messages* are the only means of transferring data and communicating within a *stream*.

Message Queue In a *stream*, a linked list of *messages* awaiting processing by a *module* or *driver*.

Read Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *upstream*.

Write Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *downstream*.

Multiplexor A multiplexor is a driver that allows *streams* associated with several user processes to be connected to a single *driver*, or several *drivers* to be connected to a single user process. STREAMS does not provide a general multiplexing *driver*, but does provide the facilities for constructing them, and for connecting multiplexed configurations of *streams*.

SEE ALSO

intro(3)

NAME

access – determine accessibility of a file

SYNOPSIS

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

DESCRIPTION

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

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

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

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

[ENOENT]
Read, write, or execute (search) permission is requested for a NULL pathname.

[ENOENT]
The named file does not exist.

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

[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.

[EFAULT]
path points outside the allocated address space for the process.

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

[ENOLINK]
path points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP]
Components of *path* require hopping to multiple remote machines.

[EINVAL]
An invalid value was specified for *amode*.

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

SEE ALSO

chmod(2), stat(2).

DIAGNOSTICS

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

NAME

acct – enable or disable process accounting

SYNOPSIS

int acct (*path*)
char **path;*

DESCRIPTION

acct is used to enable or disable the system process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal (see *exit*(2) and *signal*(2)). The effective user ID of the calling process must be superuser to use this call.

path points to a pathname naming the accounting file. The accounting file format is given in *acct*(4).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

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

[EPERM]	The effective user of the calling process is not superuser.
[EBUSY]	An attempt is being made to enable accounting when it is already enabled.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	One or more components of the accounting file pathname do not exist.
[EACCES]	The file named by *path* is not an ordinary file.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points to an illegal address.

SEE ALSO

exit(2), signal(2), acct(4)

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.

NAME

advfs – advertise a directory for remote access

SYNOPSIS

int advfs (*dir*, *resource*, *rwflag*)
char **dir;*
char **resource;*
int *rwflag;*

DESCRIPTION

advfs advertises *dir*, which points to the pathname of a local directory, under the symbolic name *resource*. *resource* will be used by remote machines to identify this directory when mounting it on those machines.

The low-order bit of *rwflag* controls write permission by remote machines on the advertised directory. If 1, writing is forbidden, otherwise writing is permitted according to the access permissions on individual files. If the local file system that contains *dir* is mounted read-only, it must be advertised read-only.

advfs may be invoked only by the superuser.

ERRORS

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

[ENONET]	The Shared Resource environment has not been started (see *rfstart*(1M)).
[EPERM]	The effective user ID is not superuser.
[ENOENT]	*dir* does not exist.
[ENOTDIR]	A component of *dir* is not a directory.
[EFAULT]	*resource* or *dir* points outside the allocated address space of the process.
[EADV]	Directory *dir* is already advertised.
[EBUSY]	*resource* is the name of a currently advertised resource.
[EREMOTE]	*dir* is a remote directory.
[ENOSPC]	There are no more entries in the advertise table.
[EROFS]	Attempt to advertise a read-only file system as read-write.

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

rmount(2) rumount(2) unadvfs(2)

NAME

alarm – set a process alarm clock

SYNOPSIS

unsigned alarm (*sec*)
unsigned *sec;*

DESCRIPTION

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

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

If *sec* is 0, any previously made alarm request is canceled.

SEE ALSO

pause(2), signal(2), sigpause(2), sigset(2)

DIAGNOSTICS

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

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

int brk (*endds*)
char **endds;*

char *sbrk (*incr*)
int *incr;*

DESCRIPTION

brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment (see *exec*(2)). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

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

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

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

[ENOMEM]	Such a change would result in more space being allocated than is allowed by the system-imposed maximum process size (see *ulimit*(2)).
[EAGAIN]	Total amount of system memory available for a read during physical IO is temporarily insufficient (see *shmop*(2)). This may occur even though the space requested was less than the system-imposed maximum process size (see *ulimit*(2)).

SEE ALSO

exec(2), shmop(2), ulimit(2), end(3C)

DIAGNOSTICS

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

NAME

chdir – change working directory

SYNOPSIS

int chdir (*path*)
char **path;*

DESCRIPTION

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

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

[ENOTDIR]	A component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chdir* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chroot(2)

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.

NAME

chmod – change mode of file

SYNOPSIS

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

DESCRIPTION

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

Access permission bits are interpreted as:

04000	Set user ID on execution.
020#0	Set group ID on execution if # is **7**, **5**, **3**, or **1**
	Enable mandatory file/record locking if # is **6**, **4**, **2**, or **0**
01000	Save text image after execution.
00400	Read by owner.
00200	Write by owner.
00100	Execute (search if a directory) by owner.
00070	Read, write, execute (search) by group.
00007	Read, write, execute (search) by others.

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

If the effective user ID of the process is not superuser and the file is not a directory, mode bit 01000 (save text image on execution) is cleared.

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

If an executable file has the sticky bit set, the operating system does not delete the program text from memory when the last user process terminates. In this case, if the sticky bit is set the text is already available when the next user of the file executes it, thus making execution faster.

If the executing process is not owned by the superuser, *chmod* masks the sticky bit but does not return an error.

If a directory is writable and has the sticky bit set, files within that directory can be removed only if one or more of the following is true (see *unlink*(2)):

the user owns the file
the user owns the directory
the file is writable by the user
the user is the superuser

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record locking exists on a regular file. This may effect future calls to *open*(2), *creat*(2), *read*(2), and *write*(2) on this file.

chmod fails and the file mode is unchanged if one or more of the following are true:

[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.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not superuser.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chmod* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chown(2), creat(2), fcntl(2), mknod(2), open(2), read(2), write(2).
chmod(1) in the *User's Reference Manual*.

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.

NAME

chown – change owner and group of a file

SYNOPSIS

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

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

DESCRIPTION

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

Only the owner of a file (or the superuser) may change the owner or group of that file. If POSIX_CHOWN_RESTRICTED is in effect (system configuration option, off by default), only the superuser may change the owner of the file, and, to change the group, a user must belong to the specified group and be the owner of the file (or be the superuser).

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

chown always follows symbolic links, i.e., if a file is a symbolic link, *chown* changes the ownership of the file to which the link points. The *lchown* call works in the same way as *chown* except that it does *not* follow the symbolic link. *lchown* changes the ownership of the symbolic link (the target of *ln -s*) without affecting the ownership of the original file.

chown or *lchown* fails and the owner and group of the named file remains unchanged if one or more of the following are true:

[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.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not superuser. If POSIX_CHOWN_RESTRICTED is in effect, then the effective user ID is not superuser.

[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chown* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chmod(2)
chown(1) in the *User's Reference Manual*.
ln(1) in the *User's Reference Manual*.

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.

NAME

chroot – change root directory

SYNOPSIS

int chroot (*path*)
char **path;*

DESCRIPTION

path points to a path name naming a directory. *chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /. The user's working directory is unaffected by the *chroot* system call.

The effective user ID of the process must be superuser to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. cannot be used to access files outside the subtree rooted at the root directory.

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

[ENOTDIR]	Any component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EPERM]	The effective user ID is not superuser.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chroot* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chdir(2)

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.

NAME

close – close a file descriptor

SYNOPSIS

int close (*fildes*)
int *fildes;*

DESCRIPTION

fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *close* closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

If a STREAMS (see *intro*(2)) file is closed, and the calling process had previously registered to receive a SIGPOLL signal (see *signal*(2) and *sigset*(2)) for events associated with that file (see I_SETSIG in *streamio*(7)), the calling process will be unregistered for events associated with the file. The last *close* for a *stream* causes the *stream* associated with *fildes* to be dismantled. If O_NDELAY is not set and there have been no signals posted for the *stream*, *close* waits up to 15 seconds, for each module and driver, for any output to drain before dismantling the *stream*. If the O_NDELAY flag is set or if there are any pending signals, *close* does not wait for output to drain, and dismantles the *stream* immediately.

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

[EBADF]	*fildes* is not a valid open file descriptor.
[EINTR]	A signal was caught during the *close* system call.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer ctive.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), intro(2), open(2), pipe(2), signal(2), sigset(2)
streamio(7) in the *System Administrator's Reference Manual.*

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.

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

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

DESCRIPTION

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

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

> All bits set in the process's file mode creation mask are cleared (see *umask*(2))
>
> The "save text image after execution bit" of the mode is cleared (see *chmod*(2)).

Upon successful completion, a write-only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls (see *fcntl*(2)). No process may have more than 128 files open simultaneously, see NOFILES (defined in /usr/include/sys/param.h). A new file may be created with a mode that forbids writing.

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

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[ENOENT]	The path name is null.
[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.
[EROFS]	The named file resides or would reside on a read-only file system.

[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EACCES]	The file exists and write permission is denied.
[EISDIR]	The named file is an existing directory.
[EMFILE]	NOFILES file descriptors are currently open.
[EFAULT]	*path* points outside the allocated address space of the process.
[ENFILE]	The system file table is full.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see *chmod*(2)).
[EINTR]	A signal was caught during the *creat* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENOSPC]	The file system is out of inodes.

SEE ALSO

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

DIAGNOSTICS

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

NAME

dup – duplicate an open file descriptor

SYNCPSIS

int dup (*fildes*)
int *fildes;*

DESCRIPTION

fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

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

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

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

The file descriptor returned is the lowest one available.

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

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

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

[EMFILE]
NOFILES file descriptors are currently open.

[ENOLINK]
fildes is on a remote machine and the link to that machine is no longer active.

SEE ALSO

close(2), creat(2), exec(2), fcntl(2), open(2), pipe(2), lockf(3C)

DIAGNOSTICS

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

NAME

exec: execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

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

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

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

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

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

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

For signals set by *sigset*(2), *exec* will ensure that the new process has the same system signal action for each signal type whose action is SIG_DFL, SIG_IGN, or SIG_HOLD as the calling process. However, if the action is to catch the signal, then the action will be reset to SIG_DFL, and any pending signal for this type will be held.

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

The shared memory segments attached to the calling process will not be attached to the new process (see *shmop*(2)).

Profiling is disabled for the new process; see *profil*(2).

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

nice value (see *nice*(2))
process ID
parent process ID
process group ID
semadj values (see *semop*(2))
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
time left until an alarm clock signal (see *alarm*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
utime, *stime*, *cutime*, and *cstime* (see *times*(2))
file-locks (see *fcntl*(2) and *lockf*(3C))

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

[ENOENT]	One or more components of the new process path name of the file do not exist.
[ENOTDIR]	A component of the new process path of the file prefix is not a directory.
[EACCES]	Search permission is denied for a directory listed in the new process file's path prefix.
[EACCES]	The new process file is not an ordinary file.
[EACCES]	The new process file mode denies execution permission.
[ENOEXEC]	The exec is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but an invalid magic number in its header.
[ETXTBSY]	The new process file is a pure procedure (shared text) file that is currently open for writing by some process.
[ENOMEM]	The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.
[E2BIG]	The number of bytes in the new process's argument list is greater than the system-imposed limit of 8192 bytes.

[EFAULT]	Required hardware is not present.
[EFAULT]	*path*, *argv*, or *envp* point to an illegal address.
[EAGAIN]	Not enough memory.
[ELIBACC]	Required shared library does not have execute permission.
[ELIBEXEC]	Trying to *exec*(2) a shared library directly.
[EINTR]	A signal was caught during the *exec* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

alarm(2), exit(2), fcntl(2), fork(2), nice(2), ptrace(2), semop(2), signal(2), sigset(2), times(2), ulimit(2), umask(2), lockf(3C), a.out(4), environ(5)
sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

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

NAME

exit, _exit – terminate process

YNOPSIS

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

DESCRIPTION

exit terminates the calling process with the following consequences:

All open file descriptors and directory streams in the calling process are closed.

If the parent process of the calling process is executing a *wait* or *waitpid*, it is notified of the calling process's termination and the low order 8 bits (i.e., bits 0377) of *status* are made available to it (see *wait*(2), *waitpid*(2)).

If the parent process of the calling process is not executing a *wait* or *waitpid* function, the exit status code is saved for return to the parent process whenever the parent process executes an appropriate subsequent *wait* or *waitpid*. A *zombie process* is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by *times*.

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

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a semadj value (see *semop*(2)), that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed (see *plock*(2)).

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

If the process is a controlling process (see *intro*(2)), the SIGHUP signal is sent to each process that has a process group ID equal to that of the calling process.

A death of child signal is sent to the parent.

If the *exit* of this process causes a process group to become an orphaned process group (see *intro*(2)) and any member of the newly orphaned process group is stopped, the SIGHUP signal, followed by the SIGCONT signal is sent to each member of the newly orphaned process group.

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

SEE ALSO

acct(2), intro(2), plock(2), semop(2), signal(2),
sigset(2), wait(2), waitpid(2).

WARNING

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

DIAGNOSTICS

None. There can be no return from an *exit* system call.

NAME

fcntl – file control

SYNOPSIS

#include <fcntl.h>

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

DESCRIPTION

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

The commands available are:

F_DUPFD	Return a new file descriptor: Lowest numbered available file descriptor greater than or equal to *arg*. Same open file (or pipe) as the original file. Same file pointer as the original file (i.e., both file descriptors share one file pointer). Same access mode (read, write or read/write). Same file status flags (i.e., both file descriptors share the same file status flags). The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.
F_GETFD	Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0** the file remains open across *exec*, otherwise, the file is closed upon execution of *exec*.
F_SETFD	Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (**0** or **1** as above).
F_FREESP	Free-up file space according to the variable of type *struct flock* pointed to by *arg*. The *l_whence*, *l_start* and *l_len* fields specify the area of the file to be freed. The *l_type* field is ignored (see *fcntl*(5)).
F_GETFL	Get *file* status flags.
F_CHKFL	Check *arg* for validity as file status flags.

F_SETFL	Set *file* status flags to *arg*. Only certain flags can be set (see *fcntl*(5)).
F_GETLK	Get the first lock that blocks the lock description given by the variable of type *struct flock* pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged except for the lock type that is set to F_UNLCK.
F_SETLK	Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg* (see *fcntl*(5)). The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set, *fcntl* returns immediately with an error value of –1.
F_SETLKW	This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process sleeps until the segment is free to be locked.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure *flock* describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), process id (*l_pid*), and RFS system id (*l_sysid*) of the segment of the file to be affected. The process id and system id fields are used only with the F_GETLK *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting *l_len* to zero (0). If such a lock also has *l_whence* and *l_start* set to zero (0), the whole file is locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When mandatory file and record locking is active on a file, (see *chmod*(2)), *read* and *write* system calls issued on the file is affected by the record locks in effect.

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

[EBADF]	*fildes* is not a valid open file descriptor.
[EINVAL]	*cmd* is F_DUPFD. *arg* is either negative, or greater than or equal to the configured value for the maximum number of open file descriptors allowed each user.
[EINVAL]	*cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.
[EACCES]	*cmd* is F_SETLK the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write (F_WRLCK) lock and the segment of a file to be locked is already read or write locked by another process.
[ENOLCK]	*cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.

[EDEADLK]	*cmd* is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling-process to sleep, waiting for that lock to become free, would cause a deadlock.
[EFAULT]	*cmd* is F_SETLK, *arg* points outside the program address space.
[EINTR]	A signal was caught during the *fcntl* system call.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.

EE ALSO

close(2), creat(2), dup(2), exec(2), fork(2), open(2), pipe(2), fcntl(5)

IAGNOSTICS

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

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

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

'ARNINGS

Because in the future the variable *errno* is set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

NAME

fork – create a new process

SYNOPSIS

int fork ()

DESCRIPTION

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

environment
close-on-exec flag (see *exec*(2))
signal handling settings (i.e., **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, function address)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
nice value (see *nice*(2))
all attached shared memory segments (see *shmop*(2))
process group ID
tty group ID (see *exit*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
process session ID

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

The child process has a unique process ID.

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

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

All semadj values are cleared (see *semop*(2)).

Process locks, text locks and data locks are not inherited by the child (see *plock*(2)).

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0. The time(s) left until an alarm clock signal is reset to 0. (See *setitimer*(2)).

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 would be exceeded.
[EAGAIN]	The system-imposed limit on the total number of processes under execution by a single user would be exceeded.
[EAGAIN]	Total amount of system memory available when reading via raw IO is temporarily insufficient.

SEE ALSO

exec(2), nice(2), plock(2), ptrace(2), semop(2), shmop(2), signal(2), sigset(2), times(2), ulimit(2), umask(2), wait(2), setitimer(2)

DIAGNOSTICS

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

NAME

getdents – read directory entries and put in a file system independent format

SYNOPSIS

#include <sys/dirent.h>

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

DESCRIPTION

fildes is a file descriptor obtained from an *open*(2) or *dup*(2) system call.

getdents attempts to read *nbyte* bytes from the directory associated with *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte*.

The file system independent directory entry is specified by the *dirent* structure. For a description of this see *dirent*(4).

On devices capable of seeking, *getdents* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *getdents*, the file pointer is incremented to point to the next directory entry.

This system call was developed in order to implement the *readdir*(3X) routine (for a description see *directory*(3X)), and should not be used for other purposes.

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

[EBADF]	*fildes* is not a valid file descriptor open for reading.
[EFAULT]	*buf* points outside the allocated address space.
[EINVAL]	*nbyte* is not large enough for one directory entry.
[ENOENT]	The current file pointer for the directory is not located at a valid entry.
[ENOLINK]	*fildes* points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	*fildes* is not a directory.

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

SEE ALSO

directory(3X), dirent(4)

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached. If the system call failed, a –1 is returned and *errno* is set to indicate the error.

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>** . The *getitimer* call returns the current value for the timer specified in *which* in the structure at *value* . The *setitimer* call sets a timer to the specified *value* (returning the previous value of the timer if *ovalue* is nonzero).

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

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

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

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.

RETURN VALUE

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

ERRORS

The possible errors are:

[EFAULT]	The *value* parameter specified a bad address.
[EINVAL]	A *value* parameter specified a time was too large to be handled.

SEE ALSO

sigaction(2), alarm(2)

NAME

getmsg – get next message off a stream

SYNOPSIS

#include <stropts.h>

int getmsg (*fd, ctlptr, dataptr, flags*)
int fd;
struct strbuf **ctlptr;*
struct strbuf **dataptr;*
int **flags;*

DESCRIPTION

getmsg retrieves the contents of a message (see *intro*(2)) located at the *stream head* read queue from a STREAMS file, and places the contents into user specified buffer(s). The message must contain either a data part, a control part or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the STREAMS module that generated the message.

fd specifies a file descriptor referencing an open *stream*. *ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;  /* maximum buffer length */
int len;     /* length of data        */
char *buf;   /* ptr to buffer    */
```

where *buf* points to a buffer in which the data or control information is to be placed, and *maxlen* indicates the maximum number of bytes this buffer can hold. On return, *len* contains the number of bytes of data or control information actually received, or is 0 if there is a zero-length control or data part, or is -1 if no data or control information is present in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

ctlptr is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctlptr* (or *dataptr*) is NULL or the *maxlen* field is -1, the control (or data) part of the message is not processed and is left on the *stream head* read queue and *len* is set to -1. If the *maxlen* field is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and *len* is set to If the *maxlen* field is set to 0 and there are more than zero bytes of control (or data) information, that information is left on the read queue and *len* is set to 0. If the *maxlen* field in *ctlptr* or *dataptr* is less than, respectively, the control or data part of the message, *maxlen* bytes are retrieved. In this case, the remainder of the message is left on the *stream head* read queue and a non-zero return value is provided, as described below under *DIAGNOSTI If information is retrieved from a priority* message, *flags* is set to RS_HIPRI on return.

By default, *getmsg* processes the first priority or non-priority message available on the *stream head* read queue. However, a user may choose to retrieve only priority messages by setting *flags* to RS_HIPRI. In this case, *getmsg* will only process the next message if it is a priority message.

If O_NDELAY has not been set, *getmsg* blocks until a message, of the type(s) specified by *flags* (priority or either), is available on the *strea If O_NDELAY has been set and a message of the specified type(s) is not present on the read queue, getmsg* fails and sets *errno* to EAGAIN.

If a hangup occurs on the *stream* from which messages are to be retrieved, *getmsg* will continue to operate normally, as described above, until the *stream head* read queue is empty. Thereafter, it will return 0 in the *len* fields of *ctlptr* and *dataptr*.

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

[EAGAIN]	The O_NDELAY flag is set, and no messages are available.
[EBADF]	*Fd* is not a valid file descriptor open for reading.
[EBADMSG]	Queued message to be read is not valid for *getmsg*.
[EFAULT]	*ctlptr*, *dataptr*, or *flags* points to a location outside the allocated address space.
[EINTR]	A signal was caught during the *getmsg* system call.
[EINVAL]	An illegal value was specified in *flags*, or the *stream* referenced by *fd* is linked under a multiplexor.

[ENOSTR] A *stream* is not associated with *fd*.

A *getmsg* can also fail if a STREAMS error message had been received at the *stream head* before the call to *getmsg*. The error returned is the value contained in the STREAMS error message.

SEE ALSO

intro(2), read(2), poll(2), putmsg(2), write(2)
STREAMS Primer
STREAMS Programmer's Guide

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of MORECTL|MOREDATA indicates that both types of information remain. Subsequent *getmsg* calls will retrieve the remainder of the message.

NAME

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

SYNOPSIS

int getpid ()

int getpgrp ()

int getppid ()

DESCRIPTION

getpid returns the process ID of the calling process.

getpgrp returns the process group ID of the calling process.

getppid returns the parent process ID of the calling process.

SEE ALSO

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

NAME

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

SYNOPSIS

unsigned short getuid ()

unsigned short geteuid ()

unsigned short getgid ()

unsigned short getegid ()

DESCRIPTION

getuid returns the real user ID of the calling process.

geteuid returns the effective user ID of the calling process.

getgid returns the real group ID of the calling process.

getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2)

NAME

ioctl – control device

SYNOPSIS

int ioctl (*fildes, request, arg*)
int *fildes, request;*

DESCRIPTION

ioctl performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are *device-specific* control functions. The arguments *request* and *arg* are passed to the file designated by *fildes* and are interpreted by the device driver. This control is infrequently used on non-STREAMS devices, with the basic input/output functions performed through the *read*(2) and *write*(2) system

For STREAMS files, specific functions are performed by the *ioctl* call as described in *streamio*(7).

fildes is an open file descriptor that refers to a device. *Request* selects the control function to be performed and will depend on the device being addressed. *Arg* represents additional information that is needed by this specific device to perform the requested function. The data type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and STREAMS functions, generic functions are provided by more than one device driver, for example, the general terminal interface (see *termio*(7)).

ioctl will fail for any type of file if one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[ENOTTY]	*fildes* is not associated with a device driver that accepts control functions.
[EINTR]	A signal was caught during the *ioctl* system call.

ioctl will also fail if the device driver detects an error. In this case, the error is passed through *ioctl* without change to the caller. A particular driver might not have all of the following error cases. Other requests to device drivers will fail if one or more of the following are true:

[EFAULT]	*request* requires a data transfer to or from a buffer pointed to by *arg*, but some part of the buffer is outside the process's allocated space.

[EINVAL]	*request* or *arg* is not valid for this device.
[EIO]	Some physical I/O error has occurred.
[ENXIO]	The *request* and *arg* are valid for this device driver, but the service requested can not be performed on this particular subdevice.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.

STREAMS errors are described in *streamio*(7).

SEE ALSO

streamio(7), termio(7) in the *System Administrator's Reference Manual*.

DIAGNOSTICS

Upon successful completion, the value returned depends upon the device control function, but must be a non-negative integer. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

[EINVAL]	*sig* is not a valid signal number.
[EINVAL]	*sig* is SIGKILL and *pid* is 1 (proc1).
[ESRCH]	No process can be found corresponding to that specified by *pid*.

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

SEE ALSO

getpid(2), setpgrp(2), signal(2), sigset(2), sigaction(2)
kill(1) in the *User's Reference Manual*.

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.

NAME

link – link to a file

SYNOPSIS

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

DESCRIPTION

path1 points to a path name naming an existing file. *path2* points to a path name naming the new directory entry to be created. *link* creates a new link (directory entry) for the existing file.

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

[ENOTDIR]	A component of either path prefix is not a directory.
[ENOENT]	A component of either path prefix does not exist.
[EACCES]	A component of either path prefix denies search permission.
[ENOENT]	The file named by *path1* does not exist.
[EEXIST]	The link named by *path2* exists.
[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 logical devices (file systems).
[ENOENT]	*path2* points to a null path name.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	*path* points outside the allocated address space of the process.
[EMLINK]	The maximum number of links to a file would be exceeded.
[EINTR]	A signal was caught during the *link* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

unlink(2)

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.

NAME

lseek – move read/write file pointer

SYNOPSIS

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

DESCRIPTION

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

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

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

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

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned. Note that if *fildes* is a remote file descriptor and *offset* is negative, *lseek* returns the file pointer even if it is negative.

lseek fails and the file pointer remains unchanged if one or more of the following are true:

[EBADF]	*fildes* is not an open file descriptor.
[ESPIPE]	*fildes* is associated with a pipe or fifo.
[EINVAL]	*whence* is not 0, 1, or 2. *
[EINVAL]	*fildes* is not a remote file descriptor, and the resulting file pointer would be negative.

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

SEE ALSO

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

DIAGNOSTICS

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

NAME

memctl – control write/execute attributes of memory

SYNOPSIS

#include <sys/m88kbcs.h>

int memctl (*start*, *length*, *mode*);
char **start*;
int *length*;
int *mode*;

DESCRIPTION

memctl can change the access mode of a part of memory. It recognizes these modes:

MCT_TEXT	1	Readable and executable
MCT_DATA	2	Readable and writable
MCT_RONLY	3	Readable only

The text section of a process is initially in mode 1. The data, bss, and stack sections are initially in mode 2. A process should only access its memory in the ways supported by the current mode, or unpredictable problems will result. The main purpose of this facility is to allow code to be written in the data section of a process and then be executed. This will not work correctly unless *memctl*( ) is called to make the relevant part of the data section executable after it has been modified and before it has been executed. If the memory is shared by several processes, all the processes must follow this procedure. *start* and *length* must be specified in bytes, and must be multiples of 4k.

RETURN VALUE

If the call fails, –1 is returned, and the global errno set to reflect the error. Otherwise, **0** is returned.

ERRORS

[EINVAL] The *mode* is invalid, or the *start* or *length* are not multiples of 4k.

[EFAULT] The region of memory specified by the *start* and *length* parameters is not valid for the process.

NAME

mkdir – make a directory

SYNOPSIS

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

DESCRIPTION

The routine *mkdir* creates a new directory with the name *path*. The mode of the new directory is initialized from the *mode*. The protection part of the *mode* argument 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 the process's effective group ID. The newly created directory is empty with the possible exception of entries for "." and "..". *mkdir* will fail and no directory will be created if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[EACCES]	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created.
[ENOENT]	The path is longer than the maximum allowed.
[EEXIST]	The named file already exists.
[EROFS]	The path prefix resides on a read-only file system.
[EFAULT]	*path* points outside the allocated address space of the process.
[EMLINK]	The maximum number of links to the parent directory would be exceeded.
[EIO]	An I/O error has occurred while accessing the file system.

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.

NAME

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

SYNOPSIS

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

DESCRIPTION

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

0170000 file type; one of the following:

> 0010000 fifo special
> 0020000 character special
> 0040000 directory
> 0060000 block special
> 0100000 or 0000000 ordinary file

0004000 set user ID on execution
00020#0 set group ID on execution if # is **7**, **5**, **3**, or **1**
enable mandatory file/record locking if # is **6**, **4**, **2**, or **0**
0001000 save text image after execution
0000777 access permissions; constructed from the following:

> 0000400 read by owner
> 0000200 write by owner
> 0000100 execute (search on directory) by owner
> 0000070 read, write, execute (search) by group
> 0000007 read, write, execute (search) by others

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

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

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

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

[EPERM]	The effective user ID of the process is not superuser.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EROFS]	The directory in which the file is to be created is located on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	*path* points outside the allocated address space of the process.
[ENOSPC]	No space is available.
[EINTR]	A signal was caught during the *mknod* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chmod(2), exec(2), umask(2), fs(4)
mkdir(1) in the *User's Reference Manual.*

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.

WARNING

If **mknod** is used to create a device in a remote directory (Remote File Sharing), the major and minor device numbers are interpreted by the server.

NAME

mount – mount a file system

SYNOPSIS

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

int mount (*spec, dir, mflag, fstyp, dataptr, datalen*)
char **spec*, **dir;*
int *mflag, fstyp;*
char **dataptr;*
int *datalen;*

DESCRIPTION

mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *spec* and *dir* are pointers to pathnames. *fstyp* is the file system type number. The *sysfs*(2) system call can be used to determine the file system type number. Note that if both the MS_DATA and MS_FSS flag bits of *mflag* are off, the file system type defaults to the **root** file system type. Only if either flag is on, does *fstyp* indicate the file system type.

If the MS_DATA flag is set in *mflag*, the system expects the *dataptr* and *datalen* arguments to be present. Together, they describe a block of file-system specific data at address *dataptr* of length *datalen*. This is interpreted by file-system specific code within the operating system, and its format depends upon the file system type. A particular file system type may not require this data, in which case *dataptr* and *datalen* should both be zero. Note that MS_FSS is obsolete and is ignored if MS_DATA is also set, but if MS_FSS is set and MS_DATA is not, *dataptr* and *datalen* are both assumed to be zero.

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

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

mount may be invoked only by the superuser. It is intended for use only by the *mount*(1M) utility.

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

[EPERM]	The effective user ID is not superuser.

[ENOENT]	Any of the named files does not exist.
[ENOTDIR]	A component of a path prefix is not a directory.
[EREMOTE]	*spec* is remote and cannot be mounted.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENOTBLK]	*spec* is not a block special device.
[ENXIO]	The device associated with *spec* does not exist.
[ENOTDIR]	*dir* is not a directory.
[EFAULT]	*spec* or *dir* points outside the allocated address space of the process.
[EBUSY]	*dir* is currently mounted on, is someone's current working directory, or is otherwise busy.
[EBUSY]	The device associated with *spec* is currently mounted.
[EBUSY]	There are no more mount table entries.
[EROFS]	*spec* is write protected and *mflag* requests write permission.
[ENOSPC]	The file system state in the super-block is not FsOKAY (or FsBITOKAY) and *mflag* requests write permission.
[EINVAL]	The super block has an invalid magic number, the *fstyp* is invalid, or *mflag* is not valid.
[ENOMEM]	An attempt to allocate space for the bit map of a high-performance file system failed.

SEE ALSO

sysfs(2), umount(2), fs(4).
mkfs(1M), mount(1M) in the *System Administrator's Reference Manual*.

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.

NAME

msgctl – message control operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (*msqid, cmd, buf*)
int *msqid, cmd;*
struct msqid_ds **buf;*

DESCRIPTION

msgctl provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*. Only super user can raise the value of **msg_qbytes**.

IPC_RMID Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

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

[EINVAL]	*msqid* is not a valid message queue identifier.
[EINVAL]	*cmd* is not a valid command.
[EACCES]	*cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process [see *intro*(2)].
[EPERM]	*cmd* is equal to **IPC_RMID** or **IPC_SET**. The effective user ID of the calling process is not equal to that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.
[EPERM]	*cmd* is equal to **IPC_SET,** an attempt is being made to increase to the value of **msg_qbytes,** and the effective user ID of the calling process is not equal to that of super user.
[EFAULT]	*buf* points to an illegal address.

SEE ALSO

intro(2), msgget(2), msgop(2)

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.

NAME

msgget – get message queue

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (*key*, *msgflg*)
key_t *key*;
int *msgflg*;

DESCRIPTION

msgget returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure (see *intro*(2)) are created for *key* if one of the following are true:

> *key* is equal to **IPC_PRIVATE**.
>
> *key* does not already have a message queue identifier associated with it, and (*msgflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

> **Msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low-order 9 bits of **msg_perm.mode** are set equal to the low-order 9 bits of *msgflg*.
>
> **Msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, **msg_susec**, **msg_rtime** , and are set equal to 0.
>
> **Msg_ctime and msg_cusec** are set equal to the current time.
>
> **Msg_qbytes** is set equal to the system limit.

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

[EACCES]	A message queue identifier exists for *key*, but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *msgflg* would not be granted.
[ENOENT]	A message queue identifier does not exist for *key* and (*msgflg* & **IPC_CREAT**) is "false".

[ENOSPC]	A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.
[EEXIST]	A message queue identifier exists for *key* but ((*msgflg* & **IPC_CREAT**) & (*msgflg* & **IPC_EXCL**)) is "true".

SEE ALSO

intro(2), msgctl(2), msgop(2)

DIAGNOSTICS

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

NAME

msgop – message operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (*msqid, msgp, msgsz, msgflg)*
int *msqid;*
struct *msgbuf* ******msgp;*
int *msgsz, msgflg;*

int msgrcv (*msqid, msgp, msgsz, msgtyp, msgflg*)
int *msqid;*
struct *msgbuf* ******msgp;*
int *msgsz;*
long *msgtyp;*
int *msgflg;*

DESCRIPTION

msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*. *msgp* points to a structure containing the message. This structure is composed of the following members: *

```
        long   mtype;    /* message type */
        char   mtext[];  /* message text */
```

mtype is a positive integer that can be used by the receiving process for message selection (see *msgrcv* below). *mtext* is any text of length *msgsz* bytes. *msgsz* can range from 0 to a system-imposed maximum.

msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to **msg_qbytes** (see *intro*(2)).

The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are:

If (*msgflg* **& IPC_NOWAIT**) is "true", the message is not sent and the calling process returns immediately.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process suspends execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

msqid is removed from the system (see *msgctl*(2)). When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(2).

msgsnd fails and no message are sent if one or more of the following are true:

[EINVAL]	*msqid* is not a valid message queue identifier.
[EACCES]	Operation permission is denied to the calling process (see *intro*(2)).
[EINVAL]	*mtype* is less than 1.
[EAGAIN]	The message cannot be sent for one of the reasons cited above and (*msgflg* & **IPC_NOWAIT**) is "true".
[EINVAL]	*msgsz* is less than zero or greater than the system-imposed limit.
[EFAULT]	*msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see *intro*(2)).

Msg_qnum is incremented by 1.

Msg_lspid is set equal to the process ID of the calling process.

Msg_stime and **msg_cusec** are set equal to the current time.

msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. **This structure is composed of the following members:**

```
long   mtype;    /* message type */
char   mtext[];  /* message text */
```

mtype is the received message's type as specified by the sending process. *mtext* is the text of the message. *msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & MSG_NOERROR) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

msgtyp specifies the type of message requested as follows:

If *msgtyp* is equal to 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the calling process returns immediately with a return value of –1 and *errno* set to ENOMSG.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process suspends execution until one of the following occurs:

A message of the desired type is placed on the queue.

msqid is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(2).

msgrcv fails and no message is received if one or more of the following are true:

[EINVAL]	*msqid* is not a valid message queue identifier.
[EACCES]	Operation permission is denied to the calling process.
[EINVAL]	*msgsz* is less than 0.
[E2BIG]	*mtext* is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "false".
[ENOMSG]	The queue does not contain a message of the desired type and (*msgtyp* & **IPC_NOWAIT**) is "true".
[EFAULT]	*msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see *intro*(2)).

Msg_qnum is decremented by 1.

Msg_lrpid is set equal to the process ID of the calling process.

Msg_rtime and **msg_rusec** are set equal to the current time.

SEE ALSO

intro(2), msgctl(2), msgget(2), signal(2).

DIAGNOSTICS

If *msgsnd* or *msgrcv* return due to the receipt of a signal, a value of –1 is returned to the calling process and *errno* is set to EINTR. If they return due to removal of *msqid* from the system, a value of –1 is returned and *errno* is set to EIDRM.

Upon successful completion, the return value is as follows:

msgsnd returns a value of 0.

msgrcv returns a value equal to the number of bytes actually placed into *mtext*.

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

NAME

nice – change priority of a process

SYNOPSIS

int *nice* (*incr*)
int *incr;*

DESCRIPTION

nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a non-negative number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. (The default nice value is 20.) Requests for values above or below these limits result in the nice value being set to the corresponding limit.

[EPERM]	*nice* will fail and not change the nice value if *incr* is negative or greater than 39 and the effective user ID of the calling process is not super-user.

SEE ALSO

exec(2)
nice(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NAME

open – open for reading or writing

SYNOPSIS

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

DESCRIPTION

path points to a path name naming a file. *open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. For non-STREAMS (see *intro*(2)) files, *oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NONBLOCK

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

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NONBLOCK is set:

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

If O_NONBLOCK is clear:

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

When opening a file associated with a communication line:

If O_NONBLOCK is set:

The open will return without waiting for carrier.

If O_NONBLOCK is clear:

The open will block until carrier is present.

O_APPEND	If set, the file pointer will be set to the end of the file prior to each write.
O_SYNC	When opening a regular file, this flag affects subsequent writes. If set, each *write*(2) will wait for both the file data and file status to be physically updated.
O_CREAT	If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the effective group ID of the process, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)): All bits set in the file mode creation mask of the process are cleared (see *umask*(2)). The "save text image after execution bit" of the mode is cleared (see *chmod*(2)).
O_TRUNC	If the file exists, its length is truncated to 0 and the mode and owner are unchanged.
O_EXCL	If O_EXCL and O_CREAT are set, *open* will fail if the file exists.
O_NOCTTY	If *path* identifies a terminal device, the terminal device will not become the controlling terminal for the process.

When opening a STREAMS file, *oflag* may be constructed from O_NDELAY or-ed with either O_RDONLY, O_WRONLY or O_RDWR. Other flag values are not applicable to STREAMS devices and have no effect on them. The value of O_NDELAY affects the operation of STREAMS drivers and certain system calls (see *read*(2), *getmsg*(2), *putmsg*(2) and *write*(2)). For drivers, the implementation of O_NDELAY is device-specific. Each STREAMS device driver may treat this option differently.

Certain flag values can be set following *open* as described in *fcntl*(2).

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

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

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

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

[EACCES]	*oflag* permission is denied for the named file.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see *chmod* (2)).
[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *open* system call.
[EIO]	A hangup or error occurred during a STREAMS *open*.
[EISDIR]	The named file is a directory and *oflag* is write or read/write.
[EMFILE]	NOFILES file descriptors are currently open.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENFILE]	The system file table is full.
[ENOENT]	O_CREAT is not set and the named file does not exist.
[ENOLINK]	*path* points to a remote machine, and the link to that machine is no longer active.
[ENOMEM]	The system is unable to allocate a send descriptor.
[ENOSPC]	O_CREAT and O_EXCL are set, and the file system is out of inodes.
[ENOSR]	Unable to allocate a *stream*.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[ENXIO]	O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
[ENXIO]	A STREAMS module or driver open routine failed.
[EROFS]	The named file resides on a read-only file system and *oflag* is write or read/write.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), intro(2), lseek(2), read(2), getmsg(2), putmsg(2), umask(2), write(2)

DIAGNOSTICS

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

NAME

pathconf, fpathconf – get configurable pathname variables

SYNOPSIS

#include <unistd.h>

long pathconf (*path*,*name*)
char **path*;
int *name*;

long fpathconf (*fields*,*name*)
int *fields*, *name*;

DESCRIPTION

pathconf and *fpathconf* provide a method for an application to determine the current value of a configurable limit or option that is associated with a file or directory.

For *fpathconf*, *path* points to a pathname of a file or directory. For *fpathconf*, *fields* is an open file descriptor. *name* is the variable to be queried relative to the file or directory. The following variables can be queried:

_PC_LINK_MAX
_PC_MAX_CANON
_PC_MAX_INPUT
_PC_NAME_MAX
_PC_PATH_MAX
_PC_PIPE_BUF
_PC_CHOWN_RESTRICTED
_PC_NO_TRUNC
_PC_BLKSIZE
_PC_VDISABLE

RETURN VALUE

If *name* is not a valid variable name, or if the variable cannot be associated with the specified file or directory, or if the process does not have permission to query the file specified by *path*, or *path* does not exist, *pathconf* will return -1, and *errno* will be set to indicate the error. If the named variable is not defined on the system, a value of -1 will be returned and *errno* will remain unchanged.

Otherwise, *pathconf* and *fpathconf* will return the current value associated with the variable for the file or directory.

ERRORS

pathconf and *fpathconf* will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	A pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX.
[ELOOP]	Too many symbolic links were encountered in translating a pathname.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*path* points to an invalid address.

fpathconf will also fail if the following condition occurs:

[EBADF]	*fields* is not a valid open file descriptor.
[EINVAL]	Name is not equal to one of the allowable values above or name cannot be associated with the specified file or directory.

SEE ALSO

sysconf(3P)

NAME

pause – suspend process until signal

SYNOPSIS

pause —

DESCRIPTION

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

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

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

SEE ALSO

alarm(2), kill(2), signal(2), sigpause(2), sigsuspend(2), wait(2)

NAME

pipe – create an interprocess channel

SYNOPSIS

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

DESCRIPTION

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

Up to 8192 bytes of data are buffered by the pipe before the writing process is blocked. A read only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis.

pipe will fail if:

[EMFILE]	NOFILES file descriptors are currently open.
[ENFILE]	The system file table is full.

SEE ALSO

read(2), write(2)
sh(1) in the *User's Reference Manual.*

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.

NAME

plock – lock process, text, or data in memory

SYNOPSIS

#include <sys/lock.h>

int plock (*op*)
int *op;*

DESCRIPTION

plock allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *plock* also allows these segments to be unlocked. The effective user ID of the calling process must be super-user to use this call. *Op* specifies the following:

PROCLOCK	lock text and data segments into memory (process lock)
TXTLOCK	lock text segment into memory (text lock)
DATLOCK	lock data segment into memory (data lock)
UNLOCK	remove locks

plock will fail and not perform the requested operation if one or more of the following are true:

[EPERM]	The effective user ID of the calling process is not super-user.
[EINVAL]	*Op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.
[EINVAL]	*Op* is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process.
[EINVAL]	*Op* is equal to **DATLOCK** and a data lock, or a process lock already exists on the calling process.
[EINVAL]	*Op* is equal to **UNLOCK** and no type of lock exists on the calling process.
[EAGAIN]	Not enough memory.

SEE ALSO

exec(2), exit(2), fork(2)

DIAGNOSTICS

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

NAME

poll – STREAMS input/output multiplexing

SYNOPSIS

#include <stropts.h>
#include <poll.h>

int poll (*fds, nfds, timeout*)
struct *pollfd fds*[];
unsigned *long nfds;*
int *timeout;*

DESCRIPTION

poll provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open *streams* (see *intro*(2)). *poll* identifies those *streams* on which a user can send or receive messages, or on which certain events have occurred. A user can receive messages using *read*(2) or *getmsg*(2) and can send messages using *write*(2) and *putmsg*(2). Certain *ioctl*(2) calls, such as I_RECVFD and I_SENDFD (see *streamio*(7)), can also be used to receive and send messages.

fds specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one element for each open file descriptor of interest. The array's elements are *pollfd* structures which contain the following members:

```
int fd;                 /* file descriptor */
short events;           /* requested events */
short revents;          /* returned events */
```

where *fd* specifies an open file descriptor and *events* and *revents* are bitmasks constructed by or-ing any combination of the following event flags:

POLLIN	A non-priority or file descriptor passing message (see I_RECVFD) is present on the *stream head* read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with POLLPRI.
POLLPRI	A priority message is present on the *stream head* read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with POLLIN.
POLLOUT	The first downstream write queue in the *stream* is not full. Priority control messages can be sent (see *putmsg*) at any time.

POLLERR	An error message has arrived at the *stream head*. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.
POLLHUP	A hangup has occurred on the *stream*. This event and POLLOUT are mutually exclusive; a *stream* can never be writable if a hangup has occurred. However, this event and POLLIN or POLLPRI are not mutually exclusive. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.
POLLNVAL	The specified *fd* value does not belong to an open *stream*. This flag is only valid in the *revents* field; it is not used in the *events* field.

For each element of the array pointed to by *fds*, *poll* examines the given file descriptor for the event(s) specified in *events*. The number of file descriptors to be examined is specified by *nfds*. If *nfds* exceeds NOFILES, the system limit of open files (see *ulimit*(2)), *poll* will fail.

If the value *fd* is less than zero, *events* is ignored and *revents* is set to 0 in that entry on return from *poll*.

The results of the *poll* query are stored in the *revents* field in the *pollfd* structure. Bits are set in the *revents* bitmask to indicate which of the requested events are true. If none are true, none of the specified bits is set in *revents* when the *poll* call returns. The event flags POLLHUP, POLLERR and POLLNVAL are always set in *revents* if the conditions they indicate are true; this occurs even though these flags were not present in *events*.

If none of the defined events have occurred on any selected file descriptor, *poll* waits at least *timeout* msec for an event to occur on any of the selected file descriptors. On a computer where millisecond timing accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. If the value *timeout* is 0, *poll* returns immediately. If the value of *timeout* is -1, *poll* blocks until a requested event occurs or until the call is interrupted. *poll* is not affected by the O_NDELAY flag.

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

[EAGAIN]	Allocation of internal data structures failed but request should be attempted again.
[EFAULT]	Some argument points outside the allocated address space.

[EINTR]	A signal was caught during the *poll* system call.
[EINVAL]	The argument *nfds* is less than zero, or *nfds* is greater than NOFILES.

SEE ALSO

intro(2), read(2), getmsg(2), putmsg(2), write(2)
streamio(7) in the *System Administrator's Reference Manual.*
STREAMS Primer.
STREAMS Programmer's Guide.

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A positive value indicates the total number of file descriptors that has been selected (i.e., file descriptors for which the *revents* field is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, a value of -1 is returned and *errno* is set to indicate the error.

NAME

profil – execution time profile

SYNOPSIS

void profil (*buff, bufsiz, offset, scale*)
char **buff;*
int *bufsiz, offset, scale;*

DESCRIPTION

buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick. Then the value of *offset* is subtracted from it, and the remainder multiplied by *scale*. If the resulting number corresponds to an entry inside *buff*, that entry is incremented. An entry is defined as a series of bytes with length *sizeof(short)*.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1-1 mapping of pc's to entries in *buff*; 077777 (octal) maps each pair of instruction entries together. 02(octal) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

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

SEE ALSO

prof(1), times(2), monitor(3C)

DIAGNOSTICS

Not defined.

NAME

ptrace – process trace

SYNOPSIS

#include <sys/types.h>
#include <sys/ptrace.h>
int 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. Its primary use is for the implementation of breakpoint debugging (see *sdb*(1)). The child process behaves normally until it encounters a signal (see *signal*(2) for the list), at which time it enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

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

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

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

3 With this request, information about the debuggee stored in the kernel address space is made available to the debugging process. This information is referenced by *addr* which is interpreted as a word offset into a synthetic structure **struct ptrace_user**, defined in <**sys/ptrace.h**>.

The request will fail if *addr* is not a relative word offset within **struct ptrace_user**, (i.e., if:

addr < 0

or

addr ≥ sizeof(ptrace_user)

or if *addr* is not a multiple of four). Upon failure, a value of -1 is returned to the debugger process and the debugger's *errno* is set to [EIO] .

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

6 **For 88k systems:** With this request, information about the debuggee stored in the kernel may be changed. This is similar to request 3, but only the following entries of the **ptrace_user** structure may be changed:

pt_r0 - pt_r31
General purpose registers.

pt_psr Processor status register. Only the byte order, carry, misaligned access, and serialize bits may be changed.

pt_fpsr Floating point user status register.

pt_fpcr Floating point user control register.

pt_sigtbl
Signal table entries.

pt_sigframe
Signal Interface Data Structure.

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

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

9 This request sets the trace bit in the Processor Status Word of the child and then executes the same steps as listed above for request 7. The trace bit causes an interrupt upon completion of one machine instruction. This effectively allows single stepping of the child.

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

DIAGNOSTICS

ptrace will in general fail if the following is true:

[EINVAL]	*Request* is not supported by this system.

SEE ALSO

sdb(1), exec(2), signal(2), wait(2)

NAME

putmsg – send a message on a stream

SYNOPSIS

#include <stropts.h>

int putmsg (*fd, ctlptr, dataptr, flags*)
int *fd;*
struct *strbuf *ctlptr;*
struct *strbuf *dataptr;*
int *flags;*

DESCRIPTION

putmsg creates a message (see *intro*(2)) from user specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part, or both. The data and control parts to be sent are distinguished by placement in separate buffers, described below. The semantics of each part is defined by the STREAMS module that receives the message.

fd specifies a file descriptor referencing an open *stream. ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;    /* not used  */
int len;       /* length of data  */
char *buf;     /* ptr to buffer  */
```

ctlptr points to the structure describing the control part, if any, to be included in the message. The *buf* field in the *strbuf* structure points to the buffer where the control information resides, and the *len* field indicates the number of bytes to be sent. The *maxlen* field is not used in *putmsg* (see *getmsg*(2)). In a similar manner, *dataptr* specifies the data, if any, to be included in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

To send the data part of a message, *dataptr* must be non-NULL and the *len* field of *dataptr* must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part will be sent if either *dataptr* (*ctlptr*) is NULL or the *len* field of *dataptr* (*ctlptr*) is set to -1.

If a control part is specified, and *flags* is set to RS_HIPRI, a *priority* message is sent. If *flags* is set to 0, a non-priority message is sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg* fails and sets *errno* to EINVAL. If no control part and no data part are specified, and *flags* is set to 0, no message is sent, and 0 is returned.

For non-priority messages, *putmsg* will block if the *stream* write queue is full due to internal flow control conditions. For priority messages, *putmsg* does not block on this condition. For non-priority messages, *putmsg* does not block when the write queue is full and O_NDELAY is set. Instead, it fails and sets *errno* to EAGAIN.

putmsg also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the *stream*, regardless of priority or whether O_NDELAY has been specified. No partial message is sent.

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

[EAGAIN]	A non-priority message was specified, the O_NDELAY flag is set and the *stream* write queue is full due to internal flow control conditions.
[EAGAIN]	Buffers could not be allocated for the message that was to be created.
[EBADF]	*fd* is not a valid file descriptor open for writing.
[EFAULT]	*ctlptr* or *dataptr* points outside the allocated address space.
[EINTR]	A signal was caught during the *putmsg* system call.
[EINVAL]	An undefined value was specified in *flags*, or *flags* is set to RS_HIPRI and no control part was supplied.
[EINVAL]	The *stream* referenced by *fd* is linked below a multiplexor.
[ENOSTR]	A *stream* is not associated with *fd*.
[ENXIO]	A hangup condition was generated downstream for the specified *stream*.
[ERANGE]	The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost *stream* module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

A *putmsg* also fails if a STREAMS error message had been processed by the *stream* head before the call to *putmsg*. The error returned is the value contained in the STREAMS error message.

SEE ALSO

intro(2), read(2), getmsg(2), poll(2), write(2)
STREAMS Primer.
STREAMS Programmer's Guide.

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.

NAME

read, readv – read from file

SYNOPSIS

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

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

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

DESCRIPTION

fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*. *readv* performs the same action, but scatters the input data into the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt−1].

For *readv*, the *iovec* structure is defined as

```
        struct iovec {
                caddr_t iov_base;
                int     iov_len;
        };
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. *readv* will always fill an area completely before proceeding to the next.

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

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

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

A *read* from a STREAMS (see *intro*(2)) file can operate in three different modes: "byte-stream" mode, "message-nondiscard" mode, and "message-discard" mode. The default is byte-stream mode. This can be changed using the I_SRDOPT *ioctl* request (see *streamio*(7)), and can be tested with the I_GRDOPT *ioctl*. In byte-stream mode, *read* will retrieve data from the *stream* until it has retrieved *nbyte* bytes, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read* retrieves data until it has read *nbyte* bytes, or until it reaches a message boundary. If the *read* does not retrieve all the data in a message, the remaining data are replaced on the *stream*, and can be retrieved by the next *read* or *getmsg*(2) call. Message-discard mode also retrieves data until it has retrieved *nbyte* bytes, or it reaches a message boundary. However, unread data remaining in a message after the *read* returns are discarded, and are not available for a subsequent *read* or *getmsg*.

When attempting to read from a regular file with mandatory file/record locking set (see *chmod*(2)), and there is a blocking (i.e. owned by another process) write lock on the segment of the file to be read:

> If O_NDELAY is set, the read will return a -1 and set errno to EAGAIN.
>
> If O_NDELAY is clear, the read will sleep until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data is written to the file or the file is no longer open for writing.

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

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data becomes available.

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

> If O_NDELAY is set, the read will return a -1 and set errno to EAGAIN.
>
> If O_NDELAY is clear, the read will block until data becomes available.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, *read* accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. *read* then returns the number of bytes read, and places the zero-byte message back on the *stream* to be retrieved by the next *read* or *getmsg.* In the two other modes, a zero-byte message returns a value of 0 and the message is removed from the *stream.* When a zero-byte message is read as the first message on a *stream,* a value of 0 is returned regardless of the read mode.

A *read* from a STREAMS file can only process data messages. It cannot process any type of protocol message and will fail if a protocol message is encountered at the *stream head.*

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

[EAGAIN]	Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.
[EAGAIN]	Total amount of system memory available when reading via raw IO is temporarily insufficient.
[EAGAIN]	No message waiting to be read on a *stream* and O_NDELAY flag set.
[EBADF]	*fildes* is not a valid file descriptor open for reading.
[EBADMSG]	Message waiting to be read on a *stream* is not a data message.
[EDEADLK]	The read was going to go to sleep and cause a deadlock situation to occur.
[EFAULT]	*Buf* points outside the allocated address space.
[EINTR]	A signal was caught during the *read* system call.
[EINVAL]	Attempted to read from a *stream* linked to a multiplexor.

[ENOLCK]	The system record lock table was full, so the read could not go to sleep until the blocking record lock was removed.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.

A *read* from a STREAMS file will also fail if an error message is received at the *stream head*. In this case, *errno* is set to the value returned in the error message. If a hangup occurs on the *stream* being read, *read* will continue to operate normally until the *stream head* read queue is empty. Thereafter, it will return 0.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2),intro(2), open(2), pipe(2), select(2), getmsg(2)
streamio(7), termio(7) in the *System Administrator's Reference Manual*.

DIAGNOSTICS

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

NAME

readlink – read value of a symbolic link

SYNOPSIS

int readlink (*path, buf, bufsiz*)
char **path,* **buf;*
int *bufsiz;*

DESCRIPTION

readlink places the contents of the symbolic link *name* in the buffer *buf* which has size *bufsiz*. The contents of the link are not null terminated when returned.

RETURN VALUE

The call returns the count of characters placed in the buffer if it succeeds, or a –1 if an error occurs, placing the error code in the global variable *errno*.

ERRORS

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

[EPERM]	The *path* argument contained a byte with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX *characters,* or an entire pathname exceeded PATH_MAX.
[ELOOP]	Too many symbolic links were encountered in translating a pathname.
[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.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the superuser.
[EINVAL]	The named file is not a symbolic link.
[EFAULT]	*buf* extends outside the process' allocated address space.

SEE ALSO

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

NAME

rename – change the name of a file

SYNOPSIS

int rename (*old, new*)
char **old*, **new*;

DESCRIPTION

rename changes the name of a file from *old* to *new*. If *old* is a file (not a directory), *new* cannot be a directory, and if *new* is an existing file, it is removed and *old* renamed. If *old* is a directory, and *new* exists, *new* must be empty, in which case it is removed and *old* renamed. If *old* and *new* refer to the same file, *rename* returns successfully without making any changes.

RETURN VALUE

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

ERRORS

rename fails and neither of the files named as arguments are affected if any of the following are true:

[ENOTDIR]	A component of either path prefix is not a directory, or *old* names a directory, and *new* is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters while _POSIX_NO_TRUE is in effect, or an entire pathname exceeded PATH_MAX.
[ENOENT]	The link named by *old* does not exist or either *old* or *new* points to an empty string.
[EACCES]	A component of either path prefix denies search permission, or one of the directories containing *old* or *new* denies write permission, or the requested link requires writing in a directory with a mode that denies write permission.
[EXDEV]	The link named by *to* and the file named by *from* are on different logical devices (file systems).
[EROFS]	The requested link requires writing in a directory on a read-only file system.

[EINVAL]	The *new* pathname contains a path prefix that names *old*.
[EBUSY]	The directory named by *old* or *new* cannot be renamed because it is being used by the system or another process.
[ENOTEMPTY]	The directory named by *new* contains files other than "." *or* "..".
[EISDIR]	The *new* points to a directory, and *old* is not a directory.
[ENOSPC]	The directory that would enter *new* cannot be extended.

SEE ALSO

mv(1), link(2), open(2), symlink(2), unlink(2).

NAME

rfstart – start the Remote File Sharing environment

SYNOPSIS

int rfstart (*aflag*);

DESCRIPTION

rfstart starts the RFS software on a machine. It must be called before the *adv*(2), *unadv*(2), *rfstop*(2), *rmount*(2), or *rumount*(2) system calls can be used.

The argument *aflag* determines whether an attempt should be made to authenticate incoming connect requests. If *aflag* equals 0, incoming requests for the RFS environment will always be accepted. If *aflag* does not equal 0, incoming requests will be verified before the connect request is accepted.

rfstart(2) may be invoked only by the superuser.

ERRORS

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

[EEXIST]	The RFS environment has already been started.
[EPERM]	The effective user ID is not superuser.
[ECOMM]	Communication error.

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

advfs(2) rmount(2) rumount(2) rfstop(2) unadvfs(2)

NAME
rfstop – stop the Remote File Sharing environment

SYNOPSIS
int rfstop( )

DESCRIPTION
rfstop stops the RFS software on a machine.

rfstop may be invoked only by the superuser.

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

[ENONET]	The RFS environment is not currently running.
[EPERM]	The effective user ID is not superuser.
[EBUSY]	This machine still has one or more remote resources mounted locally.
[EADV]	This machine still has one or more local resources advertised.
[ESRMNT]	One or more of this machine's directories is still mounted by a remote machine.

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
rfstart(2)

NAME

rmdir – remove a directory

SYNOPSIS

int rmdir (*path*)
char **path;*

DESCRIPTION

rmdir removes the directory named by the pathname pointed to by *path*. The directory must not have any entries other than "." and "..".

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

[EINVAL]	The current directory may not be removed.
[EINVAL]	The "." entry of a directory may not be removed.
[EEXIST]	The directory contains entries other than those for "." and "..".
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named directory 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 directory to be removed.
[EACCES]	The parent directory has the sticky bit set, the parent directory is not owned by the user, the directory is not owned by the user, the directory is not writable by the user, and the user is not superuser.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EROFS]	The directory entry to be removed is part of 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.
[ENOLINK]	*path* points to a remote machine, and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

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

mkdir(2)
rmdir(1), rm(1), and mkdir(1) in the *User's Reference Manual*.

NAME

rumount – unmount a remote directory

SYNOPSIS

int rumount (*resource*)
char **resource;*

DESCRIPTION

rumount unmounts a remote directory, identified by *resource.*

rumount may be invoked only by the superuser.

ERRORS

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

[ENONET]	The RFS environment has not been booted.
[EPERM]	The effective user ID is not super-user.
[EINVAL]	*resource* is invalid.
[EFAULT]	*resource* points outside the allocated address space of the process.
[ECOMM]	Communications error occurred.
[EBUSY]	*resource* is currently mounted on this machine.

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

advfs(2) rmount(2) unadvfs(2)

NAME

select – synchronous I/O multiplexing

SYNOPSIS

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

nfound = **select** (*nfds, readfds, writefdsf1, exceptfds, timeout*)
int *nfound, nfds;*
fd_set **readfds*, **writefds*, **exceptfds;*
struct timeval **timeout;*

FD_SET (*fd, &fdset*)
FD_CLR (*fd, &fdset*)
FD_ISSET (*fd, &fdset*)
FD_ZEROf (*&fdset*)
int *fd;*
fd_set fdset;

DESCRIPTION

select examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; i.e., the descriptors from 0 through *nfds*-1 in the descriptor sets are examined. On return, *select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in *nfound*.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: *FD_ZERO(&fdset)* initializes a descriptor set *fdset* to the NULL set; *FD_SET(fd, &fdset)* includes a particular descriptor *fd* in *fdset*; *FD_CLR(fd, &fdset)* removes *fd* from *fdset*; and *FD_ISSET(fd, &fdset)* is nonzero if *fd* is a member of *fdset*, otherwise, zero. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is a nonzero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select blocks indefinitely. To affect a poll, the *timeout* argument should be nonzero, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as zero pointers if no descriptors are of interest.

RETURN VALUE

select returns the number of ready descriptors that are contained in the descriptor sets, or –1 if an error occurred. If the time limit expires, *select* returns 0. If *select* returns with an error, including one due to an interrupted call, the descriptor sets are unmodified.

ERRORS

An error return from *select* indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EFAULT]	Any of *readfds*, *writefds*, *exceptfds*, or *timeout* point to an invalid portion of the process address space.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO

read(2), write(2).

BUGS

The default size FD_SETSIZE (currently 256) is somewhat larger than the current kernel limit to the number of open files. However, in order to accommodate programs that might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of **<sys/types.h>**.

select should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value is unmodified by the *select* call.

select returns and *FD_ISSET(fd, &fdset)* is nonzero if *fd* is a member of *fdset* and the read descriptor has closed. The normal behavior is to issue the read and get the errno EBADF.

NAME

semctl – semaphore control operations

SYNOPSIS

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

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
      int val;
      struct semid_ds *buf;
      ushort *array;
} farg;
```

DESCRIPTION

semctl provides a variety of semaphore control operations as specified by *cmd*.

The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum:*

GETVAL	Return the value of semval (see *intro*(2)). {READ}
SETVAL	Set the value of semval to *arg.val*. {ALTER} When this cmd is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.
GETPID	Return the value of sempid. {READ}
GETNCNT	Return the value of semncnt. {READ}
GETZCNT	Return the value of semzcnt. {READ}

The following *cmd*s return and set, respectively, every semval in the set of semaphores.

GETALL	Place semvals into array pointed to by *arg.array*. {READ}
SETALL	Set semvals according to the array pointed to by *arg.array*. {ALTER} When this cmd is successfully executed the semadj values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

IPC_STAT	Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in *intro*(2). {READ}
IPC_SET	Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*: **sem_perm.uid** **sem_perm.gid** **sem_perm.mode /* only low 9 bits */** This cmd can only be executed by a process that has an effective user ID equal to either that of superuser, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.
IPC_RMID	Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This cmd can only be executed by a process that has an effective user ID equal to either that of superuser, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

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

[EINVAL]	*semid* is not a valid semaphore identifier.
[EINVAL]	*semnum* is less than zero or greater than **sem_nsems**.
[EINVAL]	*cmd* is not a valid command.
[EACCES]	Operation permission is denied to the calling process [see *intro*(2)].
[ERANGE]	*cmd* is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum.
[EPERM]	*cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of superuser, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

[EFAULT]	*arg.buf* points to an illegal address.

SEE ALSO

intro(2), semget(2), semop(2)

DIAGNOSTICS

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

GETVAL	The value of semval.
GETPID	The value of sempid.
GETNCNT	The value of semncnt.
GETZCNT	The value of semzcnt.
All others	A value of 0.

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

NAME

semget – get set of semaphores

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (*key*, *nsemsf*, *semflg*)
key_t *key;*
int *nsems*, *semflg;*

DESCRIPTION

semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores (see *intro*(2)) are created for *key* if one of the following is true:

> *key* is equal to **IPC_PRIVATE**.
>
> *key* does not already have a semaphore identifier associated with it, and (*semflg* **& IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

> **Sem_perm.cuid**, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low-order 9 bits of **sem_perm.mode** are set equal to the low-order 9 bits of *semflg*.
>
> **Sem_nsems** is set equal to the value of *nsems*.
>
> **Sem_otime** is set equal to 0 and **sem_ctime** and **sem_cusec** are set equal to the current time.

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

[EINVAL] *nsems* is either less than or equal to zero or greater than the system-imposed limit.

[EACCES] A semaphore identifier exists for *key*, but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *semflg* would not be granted.

[EINVAL]	A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to zero.
[ENOENT]	A semaphore identifier does not exist for *key* and (*semflg* **& IPC_CREAT**) is "false".
[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded.
[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system wide would be exceeded.
[EEXIST]	A semaphore identifier exists for *key* but ((*semflg* **& IPC_CREAT**) and (*semflg* **& IPC_EXCL**)) is "true".

SEE ALSO

intro(2), semctl(2), semop(2)

DIAGNOSTICS

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

NAME

semop – semaphore operations

SYNOPSIS

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

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
unsigned nsops;
```

DESCRIPTION

semop is used to automatically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *sops* is a pointer to the array of semaphore-operation structures. *nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
short  sem_num;  /* semaphore number */
short  sem_op;   /* semaphore operation */
short  sem_flg;  /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

sem_op specifies one of three semaphore operations:

If *sem_op* is a negative integer, one of the following occurs: *

If *semval* (see *intro*(2)) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's *semadj* value (see *exit*(2)) for the specified semaphore.

If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* returns immediately.

If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* increments the *semncnt* associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occurs:

semval becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from *semval* and, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's *semadj* value for the specified semaphore.

The *semid* for which the calling process is awaiting action is removed from the system (see *semctl*(2)). When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

If *sem_op* is a positive integer, the value of *sem_op* is added to *semval* and, if (*sem_flg* & **SEM_UNDO**) is "true", the value of *sem_op* is subtracted from the calling process's *semadj* value for the specified semaphore. *

If *sem_op* is zero, one of the following occurs: *

If *semval* is zero, *semop* returns immediately.

If *semval* is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* returns immediately.

If *semval* is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* increments the *semzcnt* associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:

semval becomes zero, at which time the value of *semzcnt* associated with the specified semaphore is decremented.

The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of *semzcnt* associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

semop fails if one or more of the following are true for any of the semaphore operations specified by *sops*:

[EINVAL]	*semid* is not a valid semaphore identifier.

[EFBIG]	*sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.
[E2BIG]	*nsops* is greater than the system-imposed maximum.
[EACCES]	Operation permission is denied to the calling process (see *intro*(2))
[EAGAIN]	The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true".
[ENOSPC]	The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.
[EINVAL]	The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.
[ERANGE]	An operation would cause a *semval* to overflow the system-imposed limit.
[ERANGE]	An operation would cause a *semadj* value to overflow the system-imposed limit.
[EFAULT]	*sops* points to an illegal address.

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), semctl(2), semget(2).

DIAGNOSTICS

If *semop* returns due to the receipt of a signal, a value of –1 is returned to the calling process and *errno* is set to EINTR. If it returns due to the removal of a *semid* from the system, a value of –1 is returned and *errno* is set to EIDRM.

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

NAME

setgroups – set group access list

SYNOPSIS

#include <sys/param.h>

int setgroups (*ngroups, gidset*)
int *ngroups;*
gid_t **gidset;*

DESCRIPTION

setgroups sets the group access list of the current user process according to the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than NGROUPS, as defined in **<sys/param.h>** .

Only the superuser may set new groups.

RETURN VALUE

A 0 value is returned on success, –1 on error, with a error code stored in *errno*.

ERRORS

The *setgroups* call fails if:

[EINVAL]	The value of *ngroups* is greater than NGROUPS.
[EPERM]	The caller is not the superuser.
[EFAULT]	The address specified for *gidset* is outside the process address space.

SEE ALSO

getgroups(2)

NAME

setpgrp – set process group ID

SYNOPSIS

int setpgrp ()

DESCRIPTION

setpgrp sets the process group ID and the session ID of the calling process to the process ID of the calling process and returns the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), intro(2), kill(2), signal(2), setsid(2), setpgid(2)

DIAGNOSTICS

setpgrp returns the value of the new process group ID.

NAME

setpsr, getpsr – set/get Processor Status Register

SYNOPSIS

#include <sys/m88kbcs.h>

unsigned setpsr (*psr*);
unsigned *psr;*
unsigned getpsr ();

DESCRIPTION

setpsr sets certain bits in the Processor Status Register (PSR) of the calling process. The precise effects of these bits are defined in the Motorola *MC88100 User's Manual.*

Setting the C bit (PSR_C) sets the carry bit; clearing it resets the carry bit.

Setting the MXM bit (PSR_MXM) disables misaligned access exceptions. With the bit clear, a misaligned access causes a SIGBUS signal to be delivered to the process.

Setting the BO bit (PSR_BO) selects Little-Endian byte order. Clearing this bit selects Big-Endian, which is the normal mode is Big-Endian. All interfaces to the system must always be in Big-Endian byte order.

Setting the SER bit (PSR_SER) selects serial operation. Clearing this bit selects concurrent operation, which is the normal mode.

The *psr* value may be formed by the OR of the following:

PSR_C	0x10000000
PSR_MXM	0x00000004
PSR_BO	0x40000000
PSR_SER	0x20000000

RETURN VALUE

setpsr( ) returns the previous value of the PSR.
getpsr( ) returns the current value of the PSR.

NAME

setsid – create a new session

SYNOPSIS

init setsid( )

DESCRIPTION

setsid creates a new session and process group with the process group ID and session ID set to the process ID of the calling process. *setsid* will create a new session unless the following is true:

[EPERM] The calling process is already a process group leader.

SEE ALSO

into(2), exit(2), setpgrp(2), setpgid(2), terminos(7)

DIAGNOSTICS

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

NAME

setuid, setgid – set user and group IDs

SYNOPSIS

int setuid (*uid*)
int *uid;*

int setgid (*gid*)
int *gid;*

DESCRIPTION

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

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

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

If the effective user ID of the calling process is not superuser, but the saved set-user (group) ID from *exec*(2) is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

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

The *uid* is out of range. [EINVAL]

SEE ALSO

getuid(2), intro(2)

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.

NAME

shmctl – shared memory control operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (*shmid, cmd, buf*)
int *shmid, cmd;*
struct shmid_ds **buf;*

DESCRIPTION

shmctl provides a variety of shared memory control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */
```

This *cmd* can only be executed by a process that has an effective user ID equal to that of superuser, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to that of superuser, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

SHM_LOCK Lock the shared memory segment specified by *shmid* in memory. This *cmd* can only be executed by a process that has an effective user ID equal to superuser.

SHM_UNLOCK
Unlock the shared memory segment specified by *shmid*. This *cmd* can only be executed by a process that has an effective user ID equal to superuser.

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

[EINVAL]	*shmid* is not a valid shared memory identifier.
[EINVAL]	*cmd* is not a valid command.
[EACCES]	*cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)).
[EPERM]	*cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of superuser, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.
[EPERM]	*cmd* is equal to **SHM_LOCK** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of superuser.
[EFAULT]	*buf* points to an illegal address.
[ENOMEM]	*cmd* is equal to **SHM_LOCK** and there is not enough memory.

SEE ALSO
shmget(2), shmop(2)

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.

NOTES
The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

shmget – get shared memory segment identifier

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (*key, size, shmflg* [, *physadr*])
key_t *key;*
int *size, shmflg;*
int *physadr;*

DESCRIPTION

shmget returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of at least *size* bytes (see *intro*(2)) are created for *key* if one of the following are true:

key is equal to **IPC_PRIVATE**.

key does not already have a shared memory identifier associated with it, and (*shmflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as:

Shm_perm.cuid, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **shm_perm.mode** are set equal to the low-order 9 bits of *shmflg*. **Shm_segsz** is set equal to the value of *size*.

Shm_lpid, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.

Shm_ctime is set equal to the current time.

If (*shmflg* & *IPC_PHYS*) is "true," then *shmget* retrieves the *physadr* argument and creates a shared memory segment starting at that physical memory address. This physical memory must not be within the kernel's free memory pool. When created, a physical shared memory segment does not remove the associated memory from the system free memory pool. Upon removal, the memory is not returned to the system free memory pool.

For physical shared memory, if (*shmflg* & *IPC_NOCLEAR*) is "true", then the shared memory segment is not cleared on the first attach.

For physical shared memory, if (*shmflg* & *IPC_CI*) is "true", then the hardware cache, if any, is inhibited on this shared memory segment.

For physical shared memory, *physadr* must be aligned to a page boundary.

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

[EINVAL]	*Size* is less than the system-imposed minimum or greater than the system-imposed maximum.
[EACCES]	A shared memory identifier exists for *key* but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *shmflg* would not be granted.
[EINVAL]	A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.
[EINVAL]	*physadr* is not aligned on a page boundary.
[ENOENT]	A shared memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is "false".
[ENOSPC]	A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded.
[ENOMEM]	A shared memory identifier and associated shared memory segment are to be created but the amount of available memory is not sufficient to fill the request.
[EEXIST]	A shared memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) and (*shmflg* & **IPC_EXCL**)) is "true".
[EPERM]	A physical shared memory identifier is to be created but the effective user ID of the calling process is not superuser.

SEE ALSO

intro(2), shmctl(2), shmop(2)

DIAGNOSTICS

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

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

shmop – shared memory operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (*shmid, shmaddr, shmflg*)
int *shmid;*
char **shmaddr;*
int *shmflg;*

int shmdt (*shmaddr*)
char **shmaddr;*

DESCRIPTION

shmat attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND)** is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus **SHMLBA**)).

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND)** is "false", the segment is attached at the address given by *shmaddr*.

shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr.*

The segment is attached for reading if (*shmflg* & **SHM_RDONLY)** is "true" {READ}, otherwise, it is attached for reading and writing {READ/WRITE}.

shmat fails and does not attach the shared memory segment if one or more of the following are true:

[EINVAL]	*shmid* is not a valid shared memory identifier.
[EACCES]	Operation permission is denied to the calling process (see *intro*(2)).
[ENOMEM]	The available data space is not large enough to accommodate the shared memory segment.

[EINVAL]	*shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus **SHMLBA**)) is an illegal address.
[EINVAL]	*shmaddr* is not equal to zero, (*shmflg* & **SHM_RND)** is "false", and the value of *shmaddr* is an illegal address.
[EMFILE]	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
[EINVAL]	*shmdt* fails and does not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), shmctl(2), shmget(2).

DIAGNOSTICS

Upon successful completion, the return value is:

shmat returns the data segment start address of the attached shared memory segment.

shmdt returns a value of 0.

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

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

sigaction – examine or change signal action

SYNOPSIS

#include <signal.h>

int sigaction (*sig,act,oact*)
int *sig;*
struct sigaction **act, *oact;*

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *blocked* or *ignored* . A process may also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process.

All signals have the same priority. Signal routines invoked by *sigaction*(2) execute with the signal that caused their invocation *blocked* , but other signals may yet occur. A global "signal mask" defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a *sigprocmask*(2) call, a *sigsuspend*(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* or *ignored* 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 (see *setjmp*(3C) or *sigsetjump*(3).

sigaction allows the calling process to examine or specify the action to be taken on delivery of a signal. *sig* specifies the signal number.

The sigaction structure is defined in **<signal.h>**:

```
        struct sigaction {
                void (*sa_handler)();
                sigset_t sa_mask;
                int sa_flags;
        };
```

If *act* is not NULL, it points to a structure specifying the action to be taken when the signal is delivered. If *oact* is not NULL, the action previously associated with the signal is stored in the location pointed to by *oact*. If *act* is NULL, signal handling is unchanged; thus if *act* is NULL, *sigaction* can be used to inquire about the current handling of a given signal.

The **sa_flags** field of *act* can be used to modify the delivery of a specific signal. If *sig* is SIGCHILP and the SA_CLDSTOP bit is set in **sa_flags**, SIGCHILP will be generated if a child process stops.

When a signal is caught by a signal-catching function, a new signal mask is calculated and installed for the duration of the signal-catching function or until *sigprocmask* or *sigsuspend* is called. This mask is formed by taking the union of the current signal mask and the set associated with the action for the signal being delivered (i.e., *sa_mask*), then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested by another call to *sigaction* or until one of the *exec* functions is called.

SIGKILL and **SIGSTOP** **cannot be caught or ignored.** **SIGCONT** cannot be ignored. The set of signals specified in **sa_mask** is not allowed to block these signals. This is silently enforced.

If *sigaction* fails, no new signal handler is installed.

The following is a list of the signals with names as in the include file **<signal.h>** :

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT instruction
SIGABRT		
SIGEMT	7*	EMT instruction

SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCHILD	18●	child status has changed
SIGPWR	19	power-fail restart
SIGWINCH	20●	window size change
SIGPOLL	22	pollable event occurred
SIGSTOP	23†	stop (cannot be caught, blocked, or ignored)
SIGTSTP	24†	stop signal generated from keyboard
SIGCONT	25●	continue after stop (cannot be blocked)
SIGTTIN	26†	background read attempted from control terminal
SIGTTOU	27†	background write attempted to control terminal
SIGURG	33●	urgent condition present on socket
SIGVTALRM	37	virtual time alarm (see *setitimer*(2))
SIGPROF	38	profiling timer alarm (see *setitimer*(2))

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

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.

After a *fork*(2), the child inherits all signals, the signal mask, and the signal stack.

exec (2) resets all caught signals to default action. Ignored signals remain ignored; the signal mask remains the same.

RETURN VALUE

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

ERRORS

If any of the following conditions occur, *sigaction* will return -1 and set *errno* to the corresponding value:

[EINVAL]	The value of *sig* is not a valid signal number, or an attempt was made to supply an action for a signal that cannot be caught or ignored.
[EFAULT]	*act* and/or *oact* is an invalid address.

SEE ALSO

exec(2), kill(2), sigsetops(2), sigprocmask(2), sigsuspend(2), sigvec(2)

NAME

signal – specify what to do upon receipt of a signal

SYNOPSIS

```
#include <signal.h>

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

DESCRIPTION

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

sig can be assigned any one of the following except **SIGKILL** or **SIGSTOP**:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1]	trace trap (not reset when caught)
SIGIOT	06[1]	IOT instruction
SIGEMT	07[2]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught, blocked, or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	death of a child
SIGPWR	19[2]	power fail
SIGWINCH	20	window size change
SIGPOLL	22[3]	selectable event pending
SIGSTOP	23[4]	stop (cannot be caught, blocked, or ignored)
SIGTSTP	24[4]	stop signal generated from keyboard
SIGCONT	25[5]	continue after stop (cannot be blocked)
SIGTTIN	26[4]	background read attempted from control terminal
SIGTTON	27[4]	background write attempted to control terminal

SIGURG	33	urgent condition present on socket
SIGVTALARM	37	virtual time alarm (see *setitimer* (2))
SIGPROF	38	profiling timer alarm (see *setitimer*(2))

func is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. **SIG_DFL**, and **SIG_IGN**, are defined in the include file *signal.h*. Each is a macro that expands to a constant expression of type pointer to function returning *void*, and has a unique value that matches no declarable function.

The actions prescribed by the values of *func* are as follows:

SIG_DFL – terminate process upon receipt of a signal with the exception of those noted below:

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). See NOTES [1], [4], [5] below.

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signals **SIGKILL** and SIGSTOP cannot be ignored.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Additional arguments are passed to the signal-catching function for hardware-generated signals. Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**.

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

When a signal that is to be caught occurs during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a –1 to the calling process with *errno* set to EINTR.

signal will not catch an invalid function argument, *func*, and results are undefined when an attempt is made to execute the function at the bad address.

Note: The signal **SIGKILL** cannot be caught.

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

signal will fail if *sig* is an illegal signal number, including **SIGKILL** or **SIG-STOP**. [EINVAL]

NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask (see *umask*(2))

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

[2] For the signals **SIGCLD** and **SIGPWR**, *func* is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values are:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit*(2)].

function address - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*.

The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

In addition, **SIGCLD** affects the *wait*, and *exit* system calls as follows:

wait If the *func* value of **SIGCLD** is set to **SIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of –1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS (see *intro*(2)) file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl* call. Otherwise, the process will never receive **SIGPOLL**.

[4] For these signals, if *func* is set to **SIG_DFL**, the execution of the process will be suspended and will remain so until it receives a **SIGCONT** signal.

[5] The **SIGCONT** signal will always restart a stopped process regardless of what *func* has been set to. If *fund* is set to **SIG_DFL**, the signal will be ignored after the process has been restarted, if necessary.

SEE ALSO

intro(2), kill(2), pause(2), ptrace(2), wait(2), setjmp(3C), sigset(2), sigaction(2)
kill(1) in the *User's Reference Manual*.

DIAGNOSTICS

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and *errno* is set to indicate the error. SIG_ERR is defined in the include file *signal.h*.

NAME

sigpending – examine pending signals

SYNOPSIS

#include <signal.h>

int sigpending (*set*)
sigset_t **set;*

DESCRIPTION

sigpending stores the set of signals that are blocked from delivery and pending for the calling process at the location pointed to by *set*.

RETURN VALUE

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

ERRORS

If the following condition occurs, *sigpending* will return –1 and set *errno* to the corresponding value:

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

SEE ALSO

sigsetops(2), sigprocmask(2)

NAME

sigprocmask – examine and change blocked signals

SYNOPSIS

```
#include <signal.h>

int sigprocmask (how,set,oset)
int how;
sigset_t *set, *oset;
```

DESCRIPTION

sigprocmask allows the calling process to examine or change its signal mask. If the value of *set* is not NULL, it points to a set of signals to be used to change the currently blocked set.

The value of *how* indicated the manner in which the set is changed. The permitted values for *how* are:

SIG_BLOCK	The resulting set will be the union of the current set and the signal set pointed to by *set*.
SIG_UNBLOCK	The resulting set will be the intersection of the current set and the complement of the signal set pointed to by *set*.
SIG_SETMASK	The resulting set will be the signal set pointed to by *set*.

If *oset* is not NULL, the previous mask is stored at the location pointed to by *set*. If the value of *set* is NULL, the value of *how* is ignored and the process's signal mask is unchanged. When *set* is NULL, *sigprocmask* can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to *sigprocmask*, at least one of those signals will be delivered before *sigprocmask* returns.

SIGKILL and **SIGSTOP** cannot be caught or ignored. **SIGCONT** cannot be ignored. It is not possible to block these signals. This is silently enforced.

RETURN VALUE

Upon successful completion, zero is returned. Otherwise, -1 is returned and *errno* is set to indicate the error. If *oset* contains a valid address, its contents will contain the previous signal mask.

ERRORS

If the following condition occurs, *sigprocmask* will return –1 and set *errno* to the corresponding value:

[EINVAL]	The value of *how* is invalid.
[EFAULT]	*set* or *oset* point to an invalid address.

SEE ALSO

sigaction(2), sigpending(2), sigsetops(3P), sigsuspend(2)

NAME

sigret – return from a signal

SYNOPSIS

void sigret (*sigframe*)
char **sigframe*:

DESCRIPTION

The *sigret* system call completely restores the registers saved at the point where the signal was caught, using as its argument a pointer to the saved signal frame. The *sigret* call does not alter the floating point state.

This system call is not intended for general use.

NAME

sigset, sighold, sigrelse, sigignore, sigpause – signal management

SYNOPSIS

#include <signal.h>

void (*sigset (*sig, func*))( )
int sig;
void (**funcf*)( );

int sighold (*sig*)
int *sig;*

int sigrelse (*sig*)
int *sig;*

int sigignore (*sig*)
int *sig;*

int sigpause (*sig*)
int *sig;*

DESCRIPTION

These functions provide signal management for application processes. *sigset* specifies the system signal action to be taken upon receipt of signal *sig*. This action is either calling a process signal-catching handler *func* or performing a system-defined action.

sig can be assigned any one of the following values except SIGKILL or SIGSTOP. Machine or implementation dependent signals are not included (see *NOTES*). Each value of *sig* is a macro, defined in **<signal.h>**, that expands to an integer constant expression.

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT*	quit
SIGILL*	illegal instruction (not held when caught)
SIGTRAP*	trace trap (not held when caught)
SIGABRT*	abort
SIGFPE*	floating point exception
SIGKILL	kill (cannot be caught or ignored)
SIGSYS*	bad argument to system call
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal
SIGUSR1	user-defined signal 1

SIGUSR2	user-defined signal 2
SIGCLD	death of a child (see *WARNING*)
SIGPWR	power fail (see *WARNING*)
SIGPOLL	selectable event pending (see *NOTES*)
SIGTTIN	Background read of control terminal
SIGTTOU	Background write to control terminal
SIGTSTP	Stop signal from keyboard
SIGCONT	Continue after stop (cannot be ignored) (see NOTES).
SIGVTALARM	Virtual time alarm
SIGPROF	Profiling timer alarm
SIGWINCH	Window size change

See below under SIG_DFL about asterisks (*) in the above list.

The following values for the system-defined actions of *func* are also defined in <signal.h>. Each is a macro that expands to a constant expression of type pointer to function returning *void* and has a unique value that matches no declarable function.

SIG_DFL – default system action
Upon receipt of the signal *sig*, the receiving process is to be terminated with all the consequences outlined in *exit*(2), except for those signals marked with an "e" above. For these signals, the receiving process is to be suspended upon the receipt of the signal and remain suspended until the receipt of a SIGCONT signal. In addition, a "core image" is made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named *core* exists and is writable or can be created. If the file must be created, it has the following properties:

a mode of 0666 modified by the file creation mask (see *umask*(2))

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

SIG_IGN – ignore signal
Any pending signal *sig* is discarded and the system signal action is set to ignore future occurrences of this signal type.

SIG_HOLD – hold signal
The signal *sig* is to be held upon receipt. Any pending signal of this type remains held. Only one signal of each type is held.

Otherwise, *func* must be a pointer to a function, the signal-catching handler, that is to be called when signal *sig* occurs. In this case, *sigset* specifies that the process calls this function upon receipt of signal *sig*. Any pending signal of this type is released. This handler address is retained across calls to the other signal management functions listed here.

When a signal occurs, the signal number *sig* is passed as the only argument to the signal-catching handler. Before calling the signal-catching handler, the system signal action is set to SIG_HOLD . During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, *sigrelse* must be called to restore the system signal action and release any held signal of this type.

In general, upon return from the signal-catching handler, the receiving process resumes execution at the point it was interrupted. However, when a signal is caught during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call during a *sigpause* system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal-catching handler is executed and then the interrupted system call may return a –1 to the calling process with *errno* set to EINTR.

sighold and *sigrelse* are used to establish critical regions of code. *sighold* is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by *sigrelse*. *sigrelse* restores the system signal action to that specified previously by *sigset*.

sigignore sets the action for signal *sig* to SIG_IGN (see above).

sigpause suspends the calling process until it receives a signal, the same as *pause*(2). However, if the signal *sig* had been received and held, it is released and the system signal action taken. This system call is useful for testing variables that are changed on the occurrence of a signal. The correct usage is to use *sighold* to block the signal first, then test the variables. If they have not changed, call *sigpause* to wait for the signal. *sigset* fails if one or more of the following are true:

[EINVAL]	*sig* is an illegal signal number (including **SIGKILL**) or the default handling of *sig* cannot be changed.
[EINTR]	A signal was caught during the system call *sigpause*.

DIAGNOSTICS

Upon successful completion, *sigset* returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and *errno* is set to indicate the error. SIG_ERR is defined in <**signal.h**>.

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

SEE ALSO

kill(2), pause(2), sigaction(2), signal(2), wait(2), setjmp(3C).

WARNING

Two signals that behave differently than the signals described above exist in this release of the system:

SIGCLD	death of a child (reset when caught)
SIGPWR	power fail (not reset when caught)

For these signals, *func* is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes does not create zombie processes when they terminate (see *exit*(2)).

function address - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described for *func* equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals are ignored. (This is the default action.)

The **SIGCLD** affects two other system calls (*wait*(2), and *exit*(2)) in the following ways:

wait
  If the *func* value of **SIGCLD** is set to SIG_IGN and a *wait* is executed, the *wait* blocks until all of the calling process's child processes terminate; it then returns a value of –1 with *errno* set to ECHILD.

exit
  If in the exiting process's parent process the *func* value of **SIGCLD** is set to SIG_IGN , the exiting process does not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

NOTES

SIGPOLL is issued when a file descriptor corresponding to a STREAMS (see *intro*(2)) file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl*(2) call (see *streamio*(7)). Otherwise, the process never receives **SIGPOLL**.

For portability, applications should use only the symbolic names of signals rather than their values and use only the set of signals defined here. The action for the signals **SIGKILL** abd **SIGSTOP** can not be changed from the default system action.

Specific implementations may have other implementation-defined signals. Also, additional implementation-defined arguments may be passed to the signal-catching handler for hardware-generated signals. For certain hardware-generated signals, it may not be possible to resume execution at the point of interruption.

The signal type **SIGSEGV** is reserved for the condition that occurs on an invalid access to a data object. If an implementation can detect this condition, this signal type should be used.

The other signal management functions, *signal*(2) and *pause*(2), should not be used in conjunction with these routines for a particular signal type.

The default action for a process upon the receipt of a SIGCONT signal is to ignore the signal unless the process has been suspended, in which case its execution is continued.

NAME

sigaddset, sigdelset, sigismember, sigfillset – manipulate signal sets

SYNOPSIS

#include <fsignal.h>

int sigaddset (*set*,*signo*)
sigset_t **set*;
int *signo*;

int sigdelset (*set*,*signo*)
sigset_t **set*;
int *signo*;

int sigismember (*set*,*signo*)
sigset_t **set*;
int *signo*;

int sigfillset (*set*)
sigset_t **set*;

int siginitset (*set*)
sigset_t **set*;

DESCRIPTION

sigaddset adds the signal specified by *signo* to the set pointed to by *set*.

sigdelset deletes the signal specified by *signo* from the set pointed to by *set*.

This system defines the following signals:

SIGABRT	**SIGTERM**	**SIGIOT**
SIGALRM	**SIGUSR1**	**SIGEMT**
SIGFPE	**SIGUSR2**	**SIGBUS**
SIGHUP	**SIGCHILD**	**SIGSYS**
SIGILL	**SIGCONT**	**SIGPWR**
SIGINT	**SIGSTOP**	**SIGPOLL**
SIGKILL	**SIGTSTP**	**SIGURG**
SIGPIPE	**SIGTTIN**	**SIGWINCH**
SIGQUIT	**SIGTTOU**	**SIGVTALRM**
SIGSEGV	**SIGTRAP**	**SIGPROF**

sigfillset initializes the signal set pointed to by *set* such that all signals listed above are included.

sigismember tests whether the signal specified by *signo* is a member of the set pointed to by *set*. Applications should call *sigemptyset*(3P) or *sigfillset*(3P) for each object of type **sigset_t** before any other use of the object.

RETURN VALUE

Upon successful completion, *sigismember* returns 1 if the specified signal is a member of the specified set and zero if it is not. Upon successful completion, each of the other functions returns zero. For all the above functions, if an error is detected, the function will return -1 and set *errno* to indicate the error.

ERRORS

If the following condition occurs, the function shall return -1 and set *errno* to the corresponding value.

[EINVAL] The value of *signo* is not a valid signal number.

SEE ALSO

sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2), sigvec(2)

NAME

sigsuspend – wait for a signal

SYNOPSIS

#include <signal.h>

int sigsuspend (*sigmask*)
sigset_t **sigmask;*

DESCRIPTION

sigsuspend replaces the process's signal mask with the set of signals pointed to by *sigmask* and then suspends the process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, *sigsuspend* will not return. If the action is to execute a signal-catching function, *sigsuspend* will return after the signal-catching function returns, with the signal mask restored to the set that existed prior to the *sigsuspend* call.

SIGKILL and **SIGSTOP** cannot be caught or ignored. **SIGCONT** cannot be ignored. It is not possible to block these signals. This is silently enforced.

RETURN VALUE

Since *sigsuspend* suspends process execution indefinitely, there is no successful completion return value. If *sigsuspend* returns, it will return -1 and *errno* will be set to indicate the error.

ERRORS

If the following condition occurs, *sigsuspend* will return -1 and set *errno* to the corresponding value:

[EINTR]	A signal is caught by the calling process and control is returned from the signal-catching function.

SEE ALSO

pause(2), sigaction(2), sigsetops(2), sigpending(2), sigprocmask(2)

NAME

stat, fstat, lstat – get file status

SYNOPSIS

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

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

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

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

DESCRIPTION

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

Note that in a Remote File Sharing environment, the information returned by *stat* depends upon the user/group mapping set up between the local and remote computers. (See *idload*(1M)).

lstat is like *stat* except where the named file is a symbolic link, in which case *lstat* returns information about the link itself, while *stat* returns information about the file the link references.

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

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

The contents of the structure pointed to by *buf* include the following members:

```
        mode_t st_mode;  /* File mode (see mknod(2)) */
        ino_t  st_ino;   /* Inode number */
        dev_t  st_dev;   /* ID of device containing */
                         /* a directory entry for this file */
        dev_t  st_rdev;  /* ID of device */
                         /* This entry is defined only for */
```

```
                            /* character special or block special
        files */
        init    st_nlink; /* Number of links */
        uid_t   st_uid;   /* User ID of the file's owner */
        gid_t   st_gid;   /* Group ID of the file's group */
        off_t   st_size;  /* File size in bytes */
        time_t st_atime; /* Time of last access */
        time_t st_ausec;
        time_t st_mtime; /* Time of last data modification */
        time_t st_musec;
        time_t st_ctime; /* Time of last file status change */
        time_t st_cusec; /* Times measured in seconds and
                          /* microseconds since */
                          /* 00:00:00 GMT, Jan. 1, 1970 */
```

st_mode The mode of the file as described in the *mknod*(2) system call.

st_ino This field uniquely identifies the file in a given file system. The pair st_ino and st_dev uniquely identifies regular files.

st_dev This field uniquely identifies the file system that contains the file. Its value may be used as input to the *ustat*(2) system call to determine more information about this file system. No other meaning is associated with this value.

st_rdev This field should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.

st_nlink This field should be used only by administrative commands.

st_uid The user ID of the file's owner.

st_gid The group ID of the file's group.

st_size For regular files, this is the address of the end of the file. For pipes or fifos, this is the count of the data currently in the file. For block special or character special, this is not defined.

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

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

st_ctime, st_cusec

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

The *st_ausec*, *st_musec*, and *st_cusec* fields provide for microsecond time modulation. Presently they are filled with zeroes.

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

[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.
[EFAULT]	*buf* or *path* points to an invalid address.
[EINTR]	A signal was caught during the *stat* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

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

[EBADF]	*fildes* is not a valid open file descriptor.
[EFAULT]	*buf* points to an invalid address.
[ENOLINK]	*fildes* points to a remote machine and the link to that machine is no longer active.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), time(2), unlink(2), utime(2), write(2)

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.

NAME

statfs, fstatfs – get file system information

SYNOPSIS

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

int statfs (*path, buf, len, fstyp*)
char **path;*
struct statfs **buf;*
int *len, fstyp;*

int fstatfsf (*fildes, buf, len, fstyp*)
int *fildes;*
struct statfs **buf;*
int *len, fstyp;*

DESCRIPTION

statfs returns a "generic superblock" describing a file system. It can be used to acquire information about mounted as well as unmounted file systems, and usage is slightly different in the two cases. In all cases, *buf* is a pointer to a structure (described below) which will be filled by the system call, and *len* is the number of bytes of information which the system should return in the structure. *len* must be no greater than **sizeof (struct statfs)** and ordinarily it will contain exactly that value; if it holds a smaller value the system will fill the structure with that number of bytes. (This allows future versions of the system to grow the structure without invalidating older binary programs.)

If the file system of interest is currently mounted, *path* should name a file which resides on that file system. In this case the file system type is known to the operating system and the *fstyp* argument must be zero. For an unmounted file system *path* must name the block special file containing it and *fstyp* must contain the (non-zero) file system type. In both cases read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The *statfs* structure pointed to by *buf* includes the following members:

```
        short  f_fstyp;  /* File system type */
        short  f_bsize;  /* Block size */
        short  f_frsize; /* Fragment size */
        long   f_blocks; /* Total number of blocks */
        long   f_bfree;  /* Count of free blocks */
        long   f_files;  /* Total number of file nodes */
        long   f_ffree;  /* Count of free file nodes */
        char   f_fname[6];/* Volume name */
        char   f_fpack[6];/* Pack name */
```

fstatfs is similar, except that the file named by *path* in *statfs* is instead identified by an open file descriptor *filedes* obtained from a successful *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

statfs obsoletes *ustat*(2) and should be used in preference to it in new programs.

statfs and *fstatfs* will fail if one or more of the following are true:

[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.
[EFAULT]	*buf* or *path* points to an invalid address.
[EBADF]	*fildes* is not a valid open file descriptor.
[EINVAL]	*fstyp* is an invalid file system type; *path* is not a block special file and *fstyp* is nonzero; *len* is negative or is greater than **sizeof (struct statfs)**.
[ENOLINK]	*path* points to a remote machine, and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

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

chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), time(2), unlink(2), utime(2), write(2), fs(4)

NAME

stime – set time

SYNOPSIS

int stime (*tp*)
long ******tp;*

DESCRIPTION

stime sets the system's idea of the time and date. *tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

[EPERM]	*stime* will fail if the effective user ID of the calling process is not superuser.

SEE ALSO

time(2)

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.

NAME

symlink – make symbolic link to a file

SYNOPSIS

int symlink (*name1*, *name2*)
char **name1*, **name2*;

DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name may be an arbitrary path name; the files need not be on the same file system.

RETURN VALUE

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in *errno* and a –1 value is returned.

ERRORS

The symbolic link is made unless one or more of the following are true:

[EPERM]	Either *name1* or *name2* contains a character with the high-order bit set.
[EPERM]	A pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX.
[ELOOP]	Too many symbolic links were encountered in translating a pathname.
[ENOENT]	One of the pathnames specified was too long.
[ENOTDIR]	A component of the *name2* prefix is not a directory.
[EEXIST]	*name2* already exists.
[EACCES]	A component of the *name2* path prefix denies search permission.
[EROFS]	The file *name2* would reside on a read-only file system.
[EFAULT]	*name1* or *name2* points outside the process' allocated address space.

SEE ALSO

ln(1), link(2), readlink(2), unlink(2)

NAME

sync – update super block

SYNOPSIS

void sync ( )

DESCRIPTION

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

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

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

NAME

sys_local – implementation-defined system call

SYNOPSIS

```
#include <sys/system.h>
int sys_local (vendor, function, arg1, arg2, ...);
int vendor;
int function;
int arg1;
int arg2;
```

DESCRIPTION

The first two arguments of the *sys_local* system call consist of a vendor ID and a function number. (Vendor IDs are administered by the 88open Consortium.) The *vendor* and *function* arguments are followed by zero or more implementation-defined arguments.

If a conforming system recognizes the union of the vendor ID and the function number, it interprets the rest of the arguments in an implementation-defined manner. If an implementation does not support the union of vendor ID and function number, it returns –1 and *errno* is set to [ENOSYS].

This system call is not intended for general use.

RETURN REGISTER

r2
Other returned values are implementation-defined.

NAME

sysconf – get configurable system variables

SYNOPSIS

#include <unistd.h>

long sysconf (*name*)
init *name;*

DESCRIPTION

sysconf allows an application to determine the current value of a configurable system variable. *Name* represents the system variable to be queried. Allowable values for *name* are:

_SC_ARG_MAX	1	Maximum length of argument list on *exec*(2)
_SC_CHILD_MAX	2	Maximum number of child processes
_SC_CLK_TCK	3	Clock ticks per second
_SC_NGROUPS_MAX	4	Maximum supplementary groups per user
_SC_OPEN_MAX	5	Maximum open files per process
_SC_JOB_CONTROL	6	POSIX job control supported if non zero
_SC_SAVED_IDS	7	POSIX saved set-uid/gid feature supported
_SC_VERSION	8	POSIX version number
_SC_BCS_VERSION	9	Binary Compatibility Standard version number
_SC_BCS_VENDOR_STAMP	10	Is vendor stamp of system
_SC_BCS_SYS_ID	11	Unique System ID of this machine (if supported)
_SC_MAXUMEMV	12	Maximum user process size (in kilobytes)
_SC_MAXUPROC	13	Maximum number of processes per user
_SC_MAXMSGSZ	14	Maximum size of a message (see *msgct1*(2))
_SC_NMSGHDRS	15	Total number of message headers in system (see *msgctl*(2))
_SC_SHMMAXSZ	16	Maximum size of shared segment (see *shmct1*(2))

_SC_SHMMINSZ	17	Minimum size of shared segment (se *shmct1*(2))
_SC_SHMSEGS	18	Maximum number of attached seg ments per process (see *shmct1*(2))
_SC_NMSYSSEM	19	Total number of semaphores in systen (see *semctl*(2))
_SC_MAXSEMVL	20	Maximum semaphore value (se *semctl*(2))
_SC_NSEMMAP	21	Number of semaphore sets (se *semctl*(2))
_SC_NSEMMSL	22	Number of semaphores per set (se *semctl*(2))
_SC_NSCMMNI	23	Number of shared segments per system (see *shmctl*(2))
_SC_ITIMER_VIRT	24	System supports a virtual timer (see *setitimer*(2))
_SC_ITIMER_PROF	25	System supports a profiling timer (see *setitimer*(2))
_SC_TIMER_GRAN	26	If non zero, specifies the granularity of the real time clock in microseconds. If zero, specifies that 1 second granularity is all that is available.
_SC_PHSYMEM	27	Total physical memory of the system in kilobytes
_SC_AVAILMEM	28	Physical memory available to user processes in kilobytes
_SC_NICE	29	Nice prioritization available
_SC_MEMCTL_UNIT	30	Specifies the number of bytes that comprise a memory unit for the *memctl*(2) function.
_SC_SHMLBA	31	Memory address rounding used in the *shmsys*(2) system call
_SC_SVSTREAMS	32	System V style STREAMS are supported
_SC_CPUID	33	Returns the value of the MC88100 processor identification register

sysconf will fail if one of the following is true:

[EINVAL]
: *name* is not equal to one of the allowable values specified above.

DIAGNOSTICS

Upon successful completion, the specified system variable is returned, if it is available. If the specified system variable is not available, a value of -1 is returned, but *errno* remains unchanged. Otherwise a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), msgctl(2), shmctl(2), semctl(2), shmsys(2), setitimer(2), pathconf(2), and exec(2)

NAME

sysfs – get file system type information

SYNOPSIS

#include <sys/fstyp.h>
#include <sys/fsid.h>

int sysfs (*opcode, fsname*)
int *opcode;*
char **fsname;*

int sysfs (*opcode, fs_index, buf*)
int *opcode;*
int *fs_index;*
char **buf;*

int sysfs (*opcode*)
int *opcode;*

DESCRIPTION

sysfs returns information about the file system types configured in the system. The number of arguments accepted by *sysfs* varies and depends on the *opcode*. The currently recognized *opcodes* and their functions are described below:

GETFSIND	translates *fsname*, a null-terminated file-system identifier, into a file-system type index.
GETFSTYP	translates *fs_index*, a file-system type index, into a null-terminated file-system identifier and writes it into the buffer pointed to by *buf*; this buffer must be at least of size **FSTYPSZ** as defined in **<sys/fstyp.h>**.
GETNFSTYP	returns the total number of file system types configured in the system.

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

[EINVAL]	*fsname* points to an invalid file-system identifier; *fs_index* is zero, or invalid; *opcode* is invalid.
[EFAULT]	*buf* or *fsname* point to an invalid user address.

DIAGNOSTICS

Upon successful completion, *sysfs* returns the file-system type index if the *opcode* is **GETFSIND,** a value of 0 if the *opcode* is **GETFSTYP,** or the number of file system types configured if the *opcode* is **GETNFSTYP**. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

sysmot – machine-specific functions

SYNOPSIS

#include <sys/syslocal.h>
int sysmot (*cmd, arg1, arg2, arg3*);
int *cmd;*
int *arg1;*
int *arg2;*
int *arg3;*

DESCRIPTION

sysmot implements machine specific functions. The *cmd* argument determines the function performed. The number of arguments expected is dependent on the function.

Command RDUBLK
Reads the user block for the specified process. *arg1* is the *pid* of the desired process. *arg2* is the *address* in the calling process's address space to which to copy the user block. *arg3* is the length of the buffer in *user-space.* This function is the used by *ps*(1) to retrieve the user block for each process running on the system. The user block structure is defined in **sys/user.h**.

Command SMOTCONT
When *cmd* is SMOTCONT, the kernel continues with the instruction that was interrupted by a bus error signal to the calling routine.

Command SMOTSTIME
When *cmd* is SMOTSTIME, the argument is used as the new value for the system time and date. The argument contains the time as measured in seconds from 00:00:00 GMT January 1, 1970. Note that this command is only available to the superuser. This command is redundant in that *stime*(2) may also be used to set the system time but this command is included for compatibility with previous releases.

Command SMOTSETNAME
When *cmd* is SMOTSETNAME, the argument is expected to be a pointer to a character string. The system name and node name are set to the character string specified by the argument. Note that this command is available to the superuser only.

Command SMOTRTODC
When *cmd* is SMOTRTODC, the value of the real time clock (rtc) is returned to the address specified by the argument. If there is no real time clock on the system, the current time is returned. Note that this command is available to the superuser only.

Command S88CACHEON
When *cmd* is S88CACHEON, the caches of the MC88200 are selectively enabled based on the contents of arg1. If bit 0 of arg1 is set, the CODE cache is enabled. If bit 1 of arg1 is set, the DATA cache is enabled. Note that this command is available to the superuser only.

Command S88CACHEOFF
When *cmd* is S88CACHEOFF, the caches of the MC88200 are selectively disabled based on the contents of arg1. If bit 0 of arg1 is set, the CODE cache is disabled. If bit 1 of arg1 is set, the DATA cache is disabled. Note that this command is available to the superuser only.

Command S88CACHEFLUSH
When *cmd* is S88CACHEFLUSH, the caches of the MC88200 are selectively flushed based on the contents of arg1. If bit 0 of arg1 is set, the CODE cache is flushed. If bit 1 of arg1 is set, the DATA cache is flushed. Note that this command is available to the superuser only.

Command S88NUMPROCESSOR
When *cmd* is S88NUMPROCESSOR, no arguments are expected. The number of system processors is returned. This is an integer value between 1 and 4, inclusive.

Command SMOTDELMEM
When *cmd* is SMOTDELMEM, the argument is used as the number of pages to delete from the free list. Note that this command is available to the superuser only. This command is intended to allow stress tests to verify system behavior with low free memory.

Command SMOTADDMEM
When *cmd* is SMOTADDMEM, the argument is used as the number of pages to add to the free list. Note that this command is available to the superuser only. If more pages are added with this command than were deleted with SMOTDELMEM, only the amount previously deleted is added back.

Command SMOTMEMSIZE
When *cmd* is SMOTMEMSIZE, no arguments are expected. The size of the virtual memory space (in bytes) is returned.

Command SMOTPMEMSIZE
When *cmd* is SMOTPMEMSIZE, no arguments are expected. The amount of physical memory (in bytes) is returned.

Command SMOTSWAP
When *cmd* is SMOTSWAP, individual swapping areas may be added or deleted, or the current areas determined. The address of an appropriately primed swap buffer is passed as the only argument. (Refer to *sys/swap.h* header for details of loading the buffer.}

The format of the swap buffer is:

```
    struct swapint {
    char      si_cmd;      /*commands: list, add, delete*/
    char     *si_buf;      /*swap file path pointer*/
    int       sw_swplo;    /*start block*/
    int       si_nblks;     /*swap size*/
    }
```

Note that the add and delete options of the command may only be exercised by the superuser.

Typically, a swap area is added by a single call to sysmot. First, the swap buffer is primed with appropriate entries for the structure members. Then *sysmot* is invoked.

```
    #include <sys/syslocal.h>
    #include <sys/swap.hZ
    struct swapint swapbuf;
    sysmot (SMOTSWAP, & swapbuf);
```

SEE ALSO
swap(1M) in the *System Administrator's Reference Manual.*

NAME

time – get time

SYNOPSIS

#include <sys/types.h>

time_t time (*tloc*)
long **tloc;*

DESCRIPTION

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

If *tloc* is non-zero, the return value is also stored in the location to which *tloc* points.

SEE ALSO

stime(2)

WARNING

time fails and its actions are undefined if *tloc* points to an illegal address.

DIAGNOSTICS

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

NAME

times – get process and child process times

SYNOPSIS

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

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

DESCRIPTION

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

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

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. All times are reported in clock ticks per second. Clock ticks are a system-dependent parameter. The specific value for an implementation is defined by the variable HZ, found in the include file param.h.

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

Tms_stime is the CPU time used by the system on behalf of the calling process.

Tms_cutime is the sum of the *tms_utime*s and *tms_cutime*s of the child processes.

Tms_cstime is the sum of the *tms_stime*s and *tms_cstime*s of the child processes.

[EFAULT] *times* will fail if *buffer* points to an illegal address.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

DIAGNOSTICS

Upon successful completion, *times* returns the elapsed real time, in clock ticks per second, from an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a −1 is returned and *errno* is set to indicate the error. On a 3B2 Computer clock ticks occur 100 times per second.

NAME

uadmin – administrative control

SYNOPSIS

#include <sys/uadmin.h>

int uadmin (*cmd*, *fcn*, *mdep*)
int *cmd*, *fcn*, *mdep*;

DESCRIPTION

uadmin provides control for basic administrative functions. This system call is tightly coupled to the system administrative procedures and is not intended for general use. The argument *mdep* is provided for machine-dependent use and is not defined here.

As specified by *cmd*, the following commands are available:

A_SHUTDOWN	The system is shutdown. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.

AD_HALT	Halt the processor and turn off the power.
AD_BOOT	Reboot the system, using /unix.
AD_IBOOT	Interactive reboot; user is prompted for system name.

A_REBOOT	The system stops immediately without any further processing. The action to be taken next is specified by *fcn* as above.
A_REMOUNT	The root file system is mounted again after having been fixed. This should be used only during the startup process.

uadmin fails if any of the following are true:

[EPERM]	The effective user ID is not superuser.

DIAGNOSTICS

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

A_SHUTDOWN	Never returns.
A_REBOOT	Never returns.
A_REMOUNT	0

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

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 regular file size limit of the process. 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 regular file size limit of the process to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *ulimit* fails and the limit is unchanged if a process with an effective user ID other than super-user attempts to increase its regular file size limit. [EPERM]

3 Get the maximum possible break value (see *brk*(2)).

4 Get the current value of the maximum number of open files per process configured in the system.

SEE ALSO

brk(2), write(2)

WARNING

ulimit is effective in limiting the growth of regular files. Pipes are currently limited to 8,192 bytes.

DIAGNOSTICS

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

NAME

umask – set and get file creation mask

SYNOPSIS

int umask (*cmask*)
int *cmask*;

DESCRIPTION

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

SEE ALSO

chmod(2), creat(2), mknod(2), open(2)
mkdir(1), sh(1) in the *User's Reference Manual*.

DIAGNOSTICS

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

NAME

umount – unmount a file system

SYNOPSIS

int umount (*file*)
char **file*;

DESCRIPTION

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

umount may be invoked only by the superuser.

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

[EPERM]	The process's effective user ID is not superuser.
[EINVAL]	*file* does not exist.
[ENOTBLK]	*file* is not a block special device.
[EINVAL]	*file* is not mounted.
[EBUSY]	A file on *file* is busy.
[EFAULT]	*file* points to an illegal address.
[EREMOTE]	*file* is remote.
[ENOLINK]	*file* is on a remote machine, and the link to that machine is no longer active.
[EMULTIHOP]	Components of the path pointed to by *file* require hopping to multiple remote machines.

SEE ALSO

mount(2)

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.

NAME

unadv – unadvertise a directory

SYNOPSIS

int unadv (*resource*)
char **resource;*

DESCRIPTION

unadv unadvertises *resource,* which is the advertised domain name of a local directory. *unadv* withdraws the directory so that future attempts to remotely mount it will fail. It does not affect remote users who already have *resource* mounted; they may continue to access the directory normally.

unadv may be invoked only by the superuser.

ERRORS

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

[ENONET]	The Shared Resource environment has not been started.
[EPERM]	The effective user ID is not superuser.
[ENODEV]	*resource* is not advertised.
[EFAULT]	*resource* points outside the allocated address space of the 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

advfs(2), rmount(2)

NAME

unadvfs – unadvertise a directory

SYNOPSIS

int unadvfs (*resource*)
char **resource;*

DESCRIPTION

unadvfs unadvertises *resource*, which is the advertised domain name of a local directory. *unadvfs* withdraws the directory so that future attempts to remotely mount it fails. It does not affect remote users who already have *resource* mounted; they may continue to access the directory normally.

unadvfs may be invoked only by the superuser.

ERRORS

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

[ENONET]	The Shared Resource environment has not been started.
[EPERM]	The effective user ID is not superuser.
[ENODEV]	*resource* is not advertised.
[EFAULT]	*resource* points outside the allocated address space of the 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

advfs(2), rmount(2).

NAME

uname – get name of current system

SYNOPSIS

#include <sys/utsname.h>

int uname (*name*)
struct utsname **name;*

DESCRIPTION

uname stores information identifying the current system in the structure pointed to by *name*.

uname uses the structure defined in **<sys/utsname.h>** whose members are:

```
char   sysname[256];
char   nodename[256];
char   release[256];
char   version[256];
char   machine[256];
```

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. *release* and *version* further identify the operating system. *machine* contains a standard name that identifies the hardware that the system is running on.

Under SYSTEM V/88, the *sysname* and the *nodename* fields both return the same string.

[EFAULT] *uname* fails if *name* points to an invalid address.

SEE ALSO

uname(1) in the *User's Reference Manual*.

DIAGNOSTICS

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

NAME

unlink – remove directory entry

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

unlink removes the directory entry named by the path name pointed to by *path*.

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

[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]	The parent directory has the sticky bit set and the file is not writable by the user and the user does not own the parent directory and the user does not own the file and the user is not superuser
[EPERM]	The named file is a directory and the effective user ID of the process is not super-user.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[ETXTBSY]	The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.
[EROFS]	The directory entry to be unlinked is part of a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[EINTR]	A signal was caught during the *unlink* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

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

SEE ALSO

close(2), link(2), open(2)
rm(1) in the *User's Reference Manual.*

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.

NAME

ustat – get file system statistics

SYNOPSIS

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

int ustat (*dev, buf*)
dev_t *dev;*
struct ustat **buf;*

DESCRIPTION

ustat returns information about a mounted file system. *dev* is a device number identifying a device containing a mounted file system. *buf* is a pointer to a *ustat* structure that includes the following elements:

```
daddr_t              f_tfree;/* Total free blocks */
ino_t   f_tinode;    /* Number of free inodes */
char    f_fname[6];  /* Filsys name */
char    f_fpack[6];  /* Filsys pack name */
```

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

[EINVAL]	*dev* is not the device number of a device containing a mounted file system.
[EFAULT]	*buf* points outside the process's allocated address space.
[EINTR]	A signal was caught during a *ustat* system call.
[ENOLINK]	*dev* is on a remote machine and the link to that machine is no longer active.
[ECOMM]	*dev* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

stat(2), fs(4)

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.

NAME

utime – set file access and modification times

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

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

[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EACCES]	Search permission is denied by a component of the path prefix.
[EPERM]	The effective user ID is not superuser and not the owner of the file and *times* is not **NULL**.
[EACCES]	The effective user ID is not superuser and not the owner of the file and *times* is **NULL** and write access is denied.

[EROFS]	The file system containing the file is mounted read-only.
[EFAULT]	*times* is not **NULL** and points outside the process's allocated address space.
[EFAULT]	*path* points outside the process's allocated address space.
[EINTR]	A signal was caught during the *utime* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[EINVAL]	Either access, modification time or both have a negative value.

SEE ALSO

stat(2)

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.

NAME

wait – wait for child process to stop or terminate

SYNOPSIS

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

DESCRIPTION

wait suspends the calling process until until one of the immediate children terminates or until a child that is being traced stops, because it has hit a break point. The *wait* system call will return prematurely if a signal is received and if a child process stopped or terminated prior to the call on *wait*, return is immediate.

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

> If the child process stopped, the high order 8 bits of status will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to 0177.
>
> If the child process terminated due to an *exit* call, the low order 8 bits of status will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit* (see *exit*(2)).
>
> If the child process terminated due to a signal, the high order 8 bits of status will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit 200) is set, a ''core image'' will have been produced (see *signal*(2)).

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

wait will fail and return immediately if one or more of the following are true:

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

SEE ALSO

exec(2), exit(2), fork(2), intro(2), pause(2), ptrace(2), signal(2)

WARNING

wait fails and its actions are undefined if *stat_loc* points to an invalid address.

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

DIAGNOSTICS

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

NAME

waitpid – wait for child process to stop or terminate

SYNOPSIS

#include <sys/wait.h>

int waitpid (*statloc, options*)
int **statloc, options;*

DESCRIPTION

waitpid provides both non-blocking status collection and collection of the status of children that are stopped.

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

> If the child process stopped, the high order 8 bits of *status* will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to 0177.
>
> If the child process terminated due to an *exit* call, the low order 8 bits of *status* will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit* ; see **exit**(2).
>
> If the child process terminated due to a signal, the high order 8 bits of *status* will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit 200) is set, a ''core image'' will have been produced; see **signal**(3).

If *options* is zero, the behavior of *waitpid* is identical to *wait*(2). Otherwise, *options* consists of the logical OR of one or both of the following flags:

WNOHANG Return immediately, even if there are no children to wait for. In this case, a return value of zero indicates that no children have terminated (or stopped, if WUNTRACED is also set).

WUNTRACED

Return the status of stopped children. If the child process has stopped due to the delivery of a **SIGTTIN, SIGTTOU, SIGTSTP**, or **SIGSTOP** signal, its status may be collected by setting this flag.

If **WUNTRACED** is set and the *status* of a stopped child process is reported, the high order 8 bits of *status* shall contain the number of the signal that caused the process to stop and the low order eight bits shall be set to the octal value 0177.

RETURN VALUE

waitpid returns –1 if there are no children not previously waited for; zero is returned if **WNOHANG** is specified and there are no stopped or terminated children.

ERRORS

If any of the following conditions occur, *waitpid* will return -1 and set *errno* to the corresponding value:

[ECHILD]	The calling process has no existing unwaited-for child processes.
[EINVAL]	*waitpid* was called with an invalid *options* value.

SEE ALSO

exit(2), wait(2), wait3(2N)

NAME

write, writev – write to a file

SYNOPSIS

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

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

int writev (*fildes, iov, iovcnt*)
int *fildes;*
struct iovec **iov;*
int *iovcnt;*

DESCRIPTION

fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*. *writev* performs the same action, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt–1].

For *writev*, the *iovec* structure is defined as:

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. *writev* always writes a complete area before proceeding to the next.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

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

For regular files, if the O_SYNC flag of the file status flags is set, the write does not return until both the file data and file status have been physically updated. This function is for special applications that require extra reliability at the cost of performance. For block special files, if O_SYNC is set, the write does not return until the data has been physically updated.

A write to a regular file is blocked if mandatory file/record locking is set (see *chmod*(2)), and there is a record lock owned by another process on the segment of the file to be written. If O_NDELAY is not set, the write sleeps until the blocking record lock is removed.

For STREAMS (see *intro*(2)) files, the operation of *write* is determined by the values of the minimum and maximum *nbyte* range ("packet size"). These values are contained in the topmost *stream* module. Unless the user pushes (see I_PUSH in *streamio*(7)) the topmost module, these values can not be set or tested from user level. If *nbyte* falls within the packet size range, *nbyte* bytes are written. If *nbyte* does not fall within the range and the minimum packet size value is zero, *write* breaks the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is nonzero, *write* fails with *errno* set to ERANGE. Writing a zero-length buffer (*nbyte* is zero) sends zero bytes with zero returned.

For STREAMS files, if O_NDELAY is not set and the *stream* cannot accept data (the *stream* write queue is full due to internal flow control conditions), *write* blocks until data can be accepted. O_NDELAY prevents a process from blocking due to flow control conditions. If O_NDELAY is set and the *stream* cannot accept data, *write* fails. If O_NDELAY is set and part of the buffer has been written when a condition in which the *stream* cannot accept additional data occurs, *write* terminates and returns the number of bytes written.

When using nonblocking I/O on devices that are subject to flow control, *write* and *writev* may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

write fails and the file pointer remains unchanged if one or more of the following are true:

[EAGAIN]	Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.
[EAGAIN]	Total amount of system memory available when reading via raw IO is temporarily insufficient.
[EAGAIN]	Attempt to write to a *stream* that can not accept data with the O_NDELAY flag set.
[EBADF]	*fildes* is not a valid file descriptor open for writing.
[EDEADLK]	The write was going to go to sleep and cause a deadlock situation to occur.
[EFAULT]	*buf* points outside the process's allocated address space.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size [see *ulimit*(2)].
[EINTR]	A signal was caught during the *write* system call.
[EINVAL]	Attempt to write to a *stream* linked below a multiplexor.
[ENOLCK]	The system record lock table was full, so the write could not go to sleep until the blocking record lock was removed.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.
[ENOSPC]	During a *write* to an ordinary file, there is no free space left on the device.
[ENXIO]	A hangup occurred on the *stream* being written to.
[EPIPE and SIGPIPE signal]	An attempt is made to write to a pipe that is not open for reading by any process.
[ERANGE]	Attempt to write to a *stream* with *nbyte* outside specified minimum and maximum write range, and the minimum value is nonzero.

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* (see *ulimit*(2)) or the physical end of a medium), only as many bytes as there is room for are written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512-bytes returns 20. The next write of a nonzero number of bytes gives a failure return (except as noted below).

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) returns a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) blocks until space becomes available.

A write to a STREAMS file can fail if an error message has been received at the stream head. In this case, *errno* is set to the value included in the error message.

SEE ALSO

creat(2), dup(2), fcntl(2), intro(2), lseek(2), open(2), pipe(2), select(2), ulimit(2).

DIAGNOSTICS

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

NAME

jcsetpgrp – set process group ID for job control

SYNOPSIS

int jcsetpgrp (*pgrp*)
int pgrp;

DESCRIPTION

jcsetpgrp sets the process group ID of the calling process to *pgrp*. If *pgrp* is equal to the process ID of the calling process, the calling process becomes a job control process group leader unless the process is already the process group leader.

jcsetpgrp is part of the POSIX Job Control Option.

RETURN VALUE

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

ERRORS

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

[EINVAL]	The value of *pgrp* is less than or equal to zero or exceeds {PID_MAX}.
	The calling process is the process group leader and *pgrp* does not match the process ID.
[EPERM]	The value of *pgrp* is greater than zero and less than or equal to {PID_MAX} and there are processes already in the process group indicated by *pgrp* and none of these processes have the same controlling terminal as the calling process.
[ENOTTY]	The calling process does not have a controlling terminal.

SEE ALSO

tcsetpgrp(3P)

NAME

pathconf, fpathconf – get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>

long pathconf (path,name)
char *path;
int name;

long fpathconf (fildes,name)
int fildes, name;
```

DESCRIPTION

pathconf and *fpathconf* provide a method for an application to determine the current value of a configurable limit or option that is associated with a file or directory.

For *fpathconf, path* points to a pathname of a file or directory. For *fpathconf, fields* is an open file descriptor. *name* is the variable to be queried relative to the file or directory. The following variables can be queried:

```
_PC_LINK_MAX
_PC_MAX_CANON
_PC_MAX_INPUT
_PC_NAME_MAX
_PC_PATH_MAX
_PC_PIPE_BUF
_PC_CHOWN_RESTRICTED
_PC_NO_TRUNC
_PC_BLKSIZE
_PC_VDISABLE
```

RETURN VALUE

If *name* is not a valid variable name, or if the variable cannot be associated with the specified file or directory, or if the process does not have permission to query the file specified by *path*, or *path* does not exist, *pathconf* returns –1, and *errno* is set to indicate the error. If the named variable is not defined on the system, a value of –1 is returned and *errno* remains unchanged.

Otherwise, *pathconf* and *fpathconf* returns the current value associated with the variable for the file or directory.

ERRORS

pathconf and *fpathconf* fails if one or more of the following are true:

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

[EPERM]
A pathname contains a character with the high-order bit set.

[ENAMETOOLONG]
A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX.

[ELOOP]
Too many symbolic links were encountered in translating a path-name.

[ENOENT]
The named file does not exist.

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

[EFAULT]
path points to an invalid address.

fpathconf also fails if the following condition occurs:

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

[EINVAL]
Name is not equal to one of the allowable values above or name cannot be associated with the specified file or directory.

SEE ALSO

sysconf(2P)

NAME

setpgid – set process group ID for job control.

SYNOPSIS

int setpgid (*pid, pgid*)
int *pid, pgid;*.

DESCRIPTION

The *setpgid* function is used to either create a new process group or move the calling process or one of it's children into an already existing process group.

Upon successful completion, the process group ID of the process with a process ID which matches *pid* is set to *pgid*. If *pid* is zero, the *pid* of the calling process is used for *pid*. If *pgid* is set to 0, *pid* is used for *pgid*. *setpgid* does not set the process group ID for process *pid* equal to *pgid* if one of the following is true:

[EACCESS]
pid matches the process ID of a child process of the calling process which has successfully completed an *exec*(2) call.

[EINVAL]
pgid does not fall within the range of valid process group ID numbers.

[EPERM]
pid matches the process ID of a session leader, or *pid* matches the process ID of a child of the calling process which does not belong to the calling processes session, or there is no process with a process ID which matches the *pgid* argument within the session of the calling process.

[ESRCH]
pid does not match the process ID of the calling process or the process ID of a child of the calling process.

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

intro(2), exec(2), exit(2), fork(2), getpid(2), kill(2), setsid(2), sigaction(2).
termios(7) in the *System Administrator's Reference Manual*.

NAME

sigaction – examine or change signal action

SYNOPSIS

#include <signal.h>

int sigaction *(sig,act,oact)*
int *sig;*
struct sigaction **act, *oact;*

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *blocked* or *ignored* . A process may also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process.

All signals have the same priority. Signal routines invoked by *sigaction*(2) execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global "signal mask" defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a *sigprocmask*(2) call, a *sigsuspend*(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* or *ignored* by the process, it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process resumes execution in the context from before the signal's delivery. If the process wishes to resume in a different context, it must arrange to restore the previous context itself (see *setjmp*(3C) or *sigsetjump*(3).

sigaction allows the calling process to examine or specify the action to be taken on delivery of a signal. *sig* specifies the signal number.

The sigaction structure is defined in **<signal.h>**:

```
        struct sigaction {
                void (*sa_handler)();
                sigset_t sa_mask;
                int sa_flags;
        };
```

If *act* is not NULL, it points to a structure specifying the action to be taken when the signal is delivered. If *oact* is not NULL, the action previously associated with the signal is stored in the location pointed to by *oact*. If *act* is NULL, signal handling is unchanged; thus, if *act* is NULL, *sigaction* can be used to inquire about the current handling of a given signal.

The **sa_flags** field of *act* can be used to modify the delivery of a specific signal. If *sig* is SIGCHILP and the SA_CLDSTOP bit is set in **sa_flags**, SIGCHILP will be generated if a child process stops.

When a signal is caught by a signal-catching function, a new signal mask is calculated and installed for the duration of the signal-catching function or until *sigprocmask* or *sigsuspend* is called. This mask is formed by taking the union of the current signal mask and the set associated with the action for the signal being delivered (i.e., **sa_mask**), then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested by another call to *sigaction* or until one of the *exec* functions is called.

SIGKILL and **SIGSTOP** cannot be caught or ignored. **SIGCONT** cannot be ignored. The set of signals specified in **sa_mask** is not allowed to block these signals. This is silently enforced.

If *sigaction* fails, no new signal handler is installed.

The following is a list of the signals with names as in the include file **<signal.h>**:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT instruction
SIGABRT		

SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCHILD	18●	child status has changed
SIGPWR	19	power-fail restart
SIGWINCH	20●	window size change
SIGPOLL	22	pollable event occurred
SIGSTOP	23†	stop (cannot be caught, blocked, or ignored)
SIGTSTP	24†	stop signal generated from keyboard
SIGCONT	25●	continue after stop (cannot be blocked)
SIGTTIN	26†	background read attempted from control terminal
SIGTTOU	27†	background write attempted to control terminal
SIGURG	33●	urgent condition present on socket
SIGVTALRM	37	virtual time alarm (see *setitimer*(2))
SIGPROF	38	profiling timer alarm (see *setitimer*(2))

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

The default action for a signal may be reinstated by setting **sa_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 **sa_handler** is **SIG_IGN,** the signal is subsequently ignored, and pending instances of the signal are discarded.

After a *fork*(2), the child inherits all signals, the signal mask, and the signal stack.

exec(2) resets all caught signals to default action. Ignored signals remain ignored; the signal mask remains the same.

RETURN VALUE

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

ERRORS

If any of the following conditions occur, *sigaction* returns –1 and sets *errnc* to the corresponding value:

[EINVAL]
The value of *sig* is not a valid signal number, or an attempt was made to supply an action for a signal that cannot be caught or ignored.

[EFAULT]
act and/or *oact* is an invalid address.

SEE ALSO

exec(2), kill(2), sigsetops(2P), sigprocmask(2P), sigsuspend(2P).

NAME

sigpending – examine pending signals

SYNOPSIS

#include <signal.h>

int sigpending *(set)*
sigset_t ******set;*

DESCRIPTION

sigpending stores the set of signals that are blocked from delivery and pending for the calling process at the location pointed to by *set*.

RETURN VALUE

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

ERRORS

If the following condition occurs, *sigpending* returns –1 and sets *errno* to the corresponding value:

[EFAULT]
set points to an invalid address.

SEE ALSO

sigsetops(2P), sigprocmask(2P).

NAME

sigprocmask – examine and change blocked signals

SYNOPSIS

#include <signal.h>

int sigprocmask *(how,set,oset)*
int *how;*
sigset_t **set, *oset;*

DESCRIPTION

sigprocmask allows the calling process to examine or change its signal mask. If the value of *set* is not NULL, it points to a set of signals to be used to change the currently blocked set.

The value of *how* indicated the manner in which the set is changed. The permitted values for *how* are:

SIG_BLOCK
The resulting set is the union of the current set and the signal set pointed to by *set*.

SIG_UNBLOCK
The resulting set is the intersection of the current set and the complement of the signal set pointed to by *set*.

SIG_SETMASK
The resulting set is the signal set pointed to by *set*.

If *oset* is not NULL, the previous mask is stored at the location pointed to by *set*. If the value of *set* is NULL, the value of *how* is ignored and the process's signal mask is unchanged. When *set* is NULL, *sigprocmask* can be used to inquire about currently blocked signals.

If there are any pending unblocked signals after the call to *sigprocmask*, at least one of those signals is delivered before *sigprocmask* returns.

SIGKILL and **SIGSTOP** cannot be caught or ignored. **SIGCONT** cannot be ignored. It is not possible to block these signals. This is silently enforced.

RETURN VALUE

Upon successful completion, zero is returned. Otherwise, –1 is returned and *errno* is set to indicate the error. If *oset* contains a valid address, its contents contains the previous signal mask.

ERRORS

If the following condition occurs, *sigprocmask* returns –1 and sets *errno* to the corresponding value:

[EINVAL]
The value of *how* is invalid.

[EFAULT]
set or *oset* point to an invalid address.

SEE ALSO

sigaction(2P), sigpending(2P), sigsetops(2P), sigsuspend(2P).

NAME

sigaddset, sigdelset, sigismember, sigfillset – manipulate signal sets

SYNOPSIS

#include <fsignal.h>

int sigaddset (*set,signo*)
sigset_t **set;*
int *signo;*

int sigdelset (*set,signo*)
sigset_t **set;*
int *signo;*

int sigismember (*set,signo*)
sigset_t **set;*
int *signo;*

int sigfillset (*set*)
sigset_t **set;*

int sigemptyset (*set*)
sigset_t **set;*

DESCRIPTION

sigaddset adds the signal specified by *signo* to the set pointed to by *set.*

sigdelset deletes the signal specified by *signo* from the set pointed to by *set.*

This system defines the following signals:

SIGABRT	SIGTERM	SIGIOT
SIGALRM	SIGUSR1	SIGEMT
SIGFPE	SIGUSR2	SIGBUS
SIGHUP	SIGCHILD	SIGSYS
SIGILL	SIGCONT	SIGPWR
SIGINT	SIGSTOP	SIGPOLL
SIGKILL	SIGTSTP	SIGURG
SIGPIPE	SIGTTIN	SIGWINCH
SIGQUIT	SIGTTOU	SIGVTALRM
SIGSEGV	SIGTRAP	SIGPROF

sigfillset initializes the signal set pointed to by *set* such that all signals listed above are included.

sigismember tests whether the signal specified by *signo* is a member of the set pointed to by *set*. Applications should call *sigemptyset*(3P) or *sigfillset*(3P) for each object of type **sigset_t** before any other use of the object.

RETURN VALUE

Upon successful completion, *sigismember* returns 1 if the specified signal is a member of the specified set and zero if it is not. Upon successful completion, each of the other functions returns zero. For all the above functions, if an error is detected, the function returns –1 and sets *errno* to indicate the error.

ERRORS

If the following condition occurs, the function returns –1 and sets *errno* to the corresponding value.

[EINVAL]	The value of *signo* is not a valid signal number.

SEE ALSO

sigaction(2P), sigpending(2P), sigprocmask(2P), sigsuspend(2P).

NAME

sigsuspend – wait for a signal

SYNOPSIS

#include <signal.h>

int sigsuspend *(sigmask)*
sigset_t **sigmask;*

DESCRIPTION

sigsuspend replaces the process's signal mask with the set of signals pointed to by *sigmask*, then suspends the process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, *sigsuspend* does not return. If the action is to execute a signal-catching function, *sigsuspend* returns after the signal-catching function returns, with the signal mask restored to the set that existed prior to the *sigsuspend* call.

SIGKILL and **SIGSTOP** cannot be caught or ignored. **SIGCONT** cannot be ignored. It is not possible to block these signals; this is silently enforced.

RETURN VALUE

Since *sigsuspend* suspends process execution indefinitely, there is no successful completion return value. If *sigsuspend* returns, it returns –1 and *errno* is set to indicate the error.

ERRORS

If the following condition occurs, *sigsuspend* returns –1 and sets *errno* to the corresponding value:

[EINTR]
A signal is caught by the calling process and control is returned from the signal-catching function.

SEE ALSO

pause(2), sigaction(2P), sigsetops(2P), sigpending(2P), sigprocmask(2P).

NAME

sysconf – get configurable system variables

SYNOPSIS

#include <unistd.h>

long sysconf *(name)*
init *name;*

DESCRIPTION

sysconf allows an application to determine the current value of a configurable system variable. *name* represents the system variable to be queried. Allowable values for *name* are:

_SC_ARG_MAX	1	Maximum length of argument list on *exec*(2)
_SC_CHILD_MAX	2	Maximum number of child processes
_SC_CLK_TCK	3	Clock ticks per second
_SC_NGROUPS_MAX	4	Maximum supplementary groups per user
_SC_OPEN_MAX	5	Maximum open files per process
_SC_JOB_CONTROL	6	POSIX job control supported if non zero
_SC_SAVED_IDS	7	POSIX saved set-uid/gid feature supported
_SC_VERSION	8	POSIX version number
_SC_BCS_VERSION	9	Binary Compatibility Standard version number
_SC_BCS_VENDOR_STAMP	10	Is vendor stamp of system
_SC_BCS_SYS_ID	11	Unique System ID of this machine (if supported)
_SC_MAXUMEMV	12	Maximum user process size (in kilobytes)
_SC_MAXUPROC	13	Maximum number of processes per user
_SC_MAXMSGSZ	14	Maximum size of a message [see *msgct1*(2)]
_SC_NMSGHDRS	15	Total number of message headers in system [see *msgctl*(2)]
_SC_SHMMAXSZ	16	Maximum size of shared segment [see *shmct1*(2)]

_SC_SHMMINSZ	17	Minimum size of shared segment [see *shmct1*(2)]
_SC_SHMSEGS	18	Maximum number of attached segments per process [see *shmct1*(2)]
_SC_NMSYSSEM	19	Total number of semaphores in system [see *semctl*(2)]
_SC_MAXSEMVL	20	Maximum semaphore value [see *semctl*(2)]
_SC_NSEMMAP	21	Number of semaphore sets [see *semctl*(2)]
_SC_NSEMMSL	22	Number of semaphores per set [see *semctl*(2)]
_SC_NSCMMNI	23	Number of shared segments per system [see *shmctl*(2)]
_SC_ITIMER_VIRT	24	System supports a virtual timer [see *setitimer*(2)]
_SC_ITIMER_PROF	25	System supports a profiling timer [see *setitimer*(2)]
_SC_TIMER_GRAN	26	If non zero, specifies the granularity of the real time clock in microseconds. If zero, specifies that 1 second granularity is all that is available.
_SC_PHSYMEM	27	Total physical memory of the system in kilobytes
_SC_AVAILMEM	28	Physical memory available to user processes in kilobytes
_SC_NICE	29	Nice prioritization available
_SC_MEMCTL_UNIT	30	Specifies the number of bytes that comprise a memory unit for the *memctl*(2) function.
_SC_SHMLBA	31	Memory address rounding used in the *shmsys*(2) system call
_SC_SVSTREAMS	32	System V style STREAMS are supported
_SC_CPUID	33	Returns the value of the MC88100 processor identification register

sysconf will fail if one of the following is true:

[EINVAL]

name is not equal to one of the allowable values specified above.

IAGNOSTICS

Upon successful completion, the specified system variable is returned, if it is available. If the specified system variable is not available, a value of -1 is returned, but *errno* remains unchanged. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

EE ALSO

intro(2), msgctl(2), shmctl(2), semctl(2), shmsys(2), setitimer(2), exec(2), and pathconf(2P).

NAME

waitpid – wait for child process to stop or terminate

SYNOPSIS

#include <sys/wait.h>

int waitpid *(statloc, options)*
int **statloc, options;*

DESCRIPTION

waitpid provides both non-blocking status collection and collection of the status of children that are stopped.

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

If the child process stopped, the high order 8 bits of *status* contains the number of the signal that caused the process to stop and the low order 8 bits are set equal to 0177.

If the child process terminated due to an *exit* call, the low order 8 bits of *status* are zero and the high order 8 bits contains the low order 8 bits of the argument that the child process passed to *exit*; see *exit (2)*.

If the child process terminated due to a signal, the high order 8 bits of *status* are zero and the low order 8 bits contains the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit 200) is set, a "core image" will have been produced; see *signal (3)*.

If *options* is zero, the behavior of *waitpid* is identical to *wait*(2). Otherwise, *options* consists of the logical OR of one or both of the following flags:

WNOHANG
Return immediately, even if there are no children to wait for. Here, a return value of zero indicates that no children have terminated (or stopped, if **WUNTRACED** is also set).

WUNTRACED
Return the status of stopped children. If the child process has stopped due to the delivery of a **SIGTTIN, SIGTTOU, SIGTSTP**, or **SIGSTOP** signal, its status may be collected by setting this flag.

If **WUNTRACED** is set and the *status* of a stopped child process is reported, the high order 8 bits of *status* contains the number of the signal that caused the process to stop and the low order 8 bits are set to the octal value 0177.

RETURN VALUE

waitpid returns –1 if there are no children not previously waited for.

Zero is returned if **WNOHANG** is specified and there are no stopped or terminated children.

ERRORS

If any of the following conditions occur, *waitpid* returns -1 and sets *errno* to the corresponding value:

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

[EINVAL]
waitpid was called with an invalid *options* value.

SEE ALSO

exit(2), wait(2), wait3(2N).

NAME

intro – introduction to functions and libraries

DESCRIPTION

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

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). (For this reason the (3C) and (3S) sections together comprise one section of this manual.) The link editor *ld*(1) searches this library under the **–lc** option. A "shared library" version of *libc* can be searched using the **–lc_s** option, resulting in smaller *a.out*s. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(3S) These functions constitute the ''standard I/O package'' [see *stdio*(3S)]. These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.

(3M) These functions constitute the Math Library, *libm*. They are not automatically loaded by the C compiler, *cc*(1); however, the link editor searches this library under the **–lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**. Several generally useful mathematical constants are also defined there [see *math*(5)].

(3N) This contains sets of functions constituting the Network Services library. These sets provide protocol independent interfaces to networking services based on the service definitions of the OSI (Open Systems Interconnection) reference model. Application developers access the function sets that provide services at a particular level.

The function sets contained in the library are:

TRANSPORT INTERFACE (TI) - provide the services of the OSI Transport Layer. These services provide reliable end-to-end data transmission using the services of an underlying network. Applications written using the TI functions are independent of the underlying protocols. Declarations for these functions may be obtained from the **#include** file **<tiuser.h>**. The link editor *ld*(1) searches this library under the **–lnsl_s** option.

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

(3P) These functions constitute the ANSI/IEEE POSIX operating system interface specifications. These functions are in the library *libc*, already mentioned.

DEFINITIONS

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

Netbuf In the Network Services library, *netbuf* is a structure used in various TI functions to send and receive data and information. It contains the following members:

```
unsigned int maxlen;
unsigned int len;
char  *buf;
```

buf points to a user input and/or output buffer. *len* generally specifies the number of bytes contained in the buffer. If the structure is used for both input and output, the function will replace the user value of *len* on return.

maxlen generally has significance only when *buf* is used to receive output from the TI function. In this case, it specifies the physical size of the buffer, the maximum value of *len* that can be set by the function. If *maxlen* is not large enough to hold the returned information, an TBUFOVFLW error will generally result. However, certain functions may return part of the data and not generate an error.

FILES

LIBDIR usually /lib
LIBDIR/libc.a
LIBDIR/libc_s.a
LIBDIR/libm.a

SEE ALSO

ar(1), cc(1), ld(1), lint(1), nm(1), intro(2), stdio(3S), math(5).

DIAGNOSTICS

Functions in the C and Math Libraries (3C and 3M) may return the conventional values **0** or ±**HUGE** (the largest-magnitude single-precision floating-point numbers; **HUGE** is defined in the <*math.h*> header file) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* (see *intro*(2)) is set to the value EDOM or ERANGE.

WARNING

Many of the functions in the libraries call and/or refer to other functions and external variables described in this section and in Section 2 (*System Calls*). If a program inadvertently defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded. The *lint*(1) program checker reports name conflicts of this kind as "multiple declarations" of the names in question. Definitions for Sections 2, 3C, and 3S are checked automatically. Other definitions can be included by using the **–l** option. (For example, **–l***m* includes definitions for Section 3M, the Math Library.) Use of *lint* is highly recommended.

NAME

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

SYNOPSIS

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

DESCRIPTION

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

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

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

a64l scans the character string from left to right, decoding each character as a 6 bit Radix 64 number.

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

CAVEAT

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

NAME

abort – generate an IOT fault

SYNOPSIS

int abort ( )

DESCRIPTION

abort does the work of *exit*(2), but instead of just exiting, *abort* causes **SIGABRT** to be sent to the calling process. If **SIGABRT** is neither caught nor ignored, all *stdio*(3S) streams are flushed prior to the signal being sent, and a core dump results.

abort returns the value of the *kill*(2) system call.

SEE ALSO

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

DIAGNOSTICS

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

NAME

abs – return integer absolute value

SYNOPSIS

```
int abs (i)
int i;
```

DESCRIPTION

abs returns the absolute value of its integer operand.

SEE ALSO

floor(3M).

CAVEAT

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

NAME

bsearch – binary search a sorted table

SYNOPSIS

#include <search.h>

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 a datum instance 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 as accordingly the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

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

#define      TABSIZE      1000

struct node {                        /* these are stored in the table */
        char *string;
        int length;
};
struct node table[TABSIZE];          /* table to be searched */
        .
        .
        .
{
        struct node *node_ptr, node;
        int node_compare( );         /* routine to compare 2 nodes */
        char str_space[20];          /* space to read string into */
```

```
                .
                .
                .
                node.string = str_space;
                while (scanf("%s", node.string) != EOF) {
                        node_ptr = (struct node *)bsearch((char *)(&node),
                                (char *)table, TABSIZE,
                                sizeof(struct node), node_compare);
                        if (node_ptr != NULL) {
                                (void)printf("string = %20s, length = %d\n'
                                node_ptr->string, node_ptr->length);
                        } else {
                                (void)printf("not found: %s\n", node.strin
                        }
                }
        }
        /*
                This routine compares two nodes based on an
                alphabetical ordering of the string field.
        */
        int
        node_compare(node1, node2)
        char *node1, *node2;
        {
                        return (strcmp(
                                ((struct node *)node1)->string,
                                ((struct node *)node2)->string));
        }
```

NOTES

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

SEE ALSO

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

DIAGNOSTICS

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

NAME

clock – report CPU time used

SYNOPSIS

long clock ( )

DESCRIPTION

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

The resolution of the clock is 16 milliseconds on SYSTEM V/88 computers.

SEE ALSO

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

BUGS

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

NAME

conv: 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.

The macros *_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. The macro *_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), getc(3S).

NAME

crypt, setkey, encrypt – generate hashing encryption

SYNOPSIS

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

void setkey (*key*)
char *∗key;*

void encrypt (*block, ignored*)
char *∗block;*
int *ignored;*

DESCRIPTION

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

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

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

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *setkey*. *ignored* is unused by *encrypt* but it must be present.

SEE ALSO

getpass(3C), passwd(4).
login(1), passwd(1) in the *User's Reference Manual*.

CAVEAT

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

NAME

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

SYNOPSIS

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

char *ctime (*clock*)
time_t **clock;*

struct tm *localtime (*clock*)
time_t **clock;*

struct tm *gmtime (*clock*)
time_t **clock;*

char *asctime (*tm*)
struct tm **tm;*

int cftime (*buf, fmt, clock*)
char **buf,* **fmt;*
time_t **clock;*

int ascftime (*buf, fmt, tm*)
char **buf,* **fmt;*
struct tm **tm;*

extern long timezone, altzone;

extern int daylight;

extern char *tzname[2];

void tzset ( )

DESCRIPTION

ctime, *localtime*, and *gmtime* accept arguments of type **time_t** (declared in **<sys/types.h>**), pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970. *ctime* returns a pointer to a 26 character string in the following form (all fields have constant width):

Fri Sep 13 00:00:00 1986\n\0

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

asctime converts a "tm" structure to a 26 character string, as shown in the previous example, and returns a pointer to the string.

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

```
struct tm {
    int    tm_sec;    /* seconds after the minute - [0, 59] */
    int    tm_min;    /* minutes after the hour - [0, 59] */
    int    tm_hour;   /* hour since midnight - [0, 23] */
    int    tm_mday;   /* day of the month - [1, 31] */
    int    tm_mon;    /* months since January - [0, 11] */
    int    tm_year;   /* years since 1900 */
    int    tm_wday;   /* days since Sunday - [0, 6] */
    int    tm_yday;   /* days since January 1 - [0, 365] */
    int    tm_isdst;  /* flag for daylight savings time */
};
```

tm_isdst is non-zero if the alternate time zone is in effect.

cftime and *ascftime* provide the capabilities of *ctime* and *asctime*, respectively, as well as additional ones. *cftime* takes an integer of type *time_t* pointed to by *clock* and converts it to a character string. *ascftime* takes a pointer to a "tm" structure and converts it to a character string. In both functions, the characters are placed into the array pointed to by *buf* (plus a terminating \0) and the value returned is the number of such characters (not counting terminating \0). *fmt* controls the format of the resulting string.

fmt is a character string that consists of field descriptors and text characters, reminiscent of *printf*(3S). Each field descriptor consists of a % character followed by another character which specifies the replacement for the field descriptor. All other characters are copied from *fmt* into the result. The following field descriptors are supported:

%%	same as %
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%d	day of month (01 - 31)
%D	date as %m/%d/%y
%e	day of month (1-31; single digits are preceded by a blank)
%h	abbreviated month name

%H	hour (00 - 23)
%I	hour (00 - 12)
%j	day number of year (001 - 366)
%m	month number (01 - 12)
%M	minute (00 - 59)
%n	same as \n
%p	ante meridian or post meridian
%r	time as %I:%M:%S %p
%R	time as %H:%M
%S	seconds (00 - 59)
%t	insert a tab
%T	time as %H:%M:%S
%U	week number of year (01 - 52), Sunday is the first day of week
%w	weekday number (Sunday = 0)
%W	week number of year (01 - 52), Monday is the first day of week
%x	Local specific date format
%X	Local specific time format
%y	year within century (00 - 99)
%Y	year as ccyy (e.g. 1986)
%Z	time zone name

The difference between **%U** and **%W** lies in which day is counted as the first of the week. Week number 01 is the first week with four or more January days in it.

The example below shows what the values in the "tm" structure would look like for Thursday, August 28, 1986 at 12:44:36 in New Jersey:

ascftime (buf, "%A %m %d %j", tm)

This example results in the buffer containing "Thursday Aug 28 240".

If *fmt* is (char *)0, the value of the environment variable CFTIME is used. If CFTIME is undefined or empty, a default format is used. The default format string is taken from the file that contains the date and time strings associated with the then current language (see below for details on changing the current language and *cftime*(4) for a description of the structure of these files).

The user can request that the output of *cftime* and *ascftime* be in a specific language by setting the environment variable LANGUAGE to the desired language. If LANGUAGE is empty, unset, or set to an unsupported language, the last language requested will be used (the default is the **usa-english** strings).

The external **long** variable *timezone* contains the difference, in seconds, between GMT and the main time zone; the external **long** variable *altzone* contains the difference, in seconds, between GMT and the alternate time zone; both, *timezone* and *altzone* default to 0 (GMT). The external variable *daylight* is non-zero if an alternate time zone exists. The time zone names are contained in the external variable *tzname*, which by default is set to:

char *tzname[2] = { "GMT", "   " };

The functions know about the peculiarities of this conversion for various time periods for the U.S.A. (specifically, the years 1974, 1975, and 1987). The functions will handle the new daylight savings time starting with the first Sunday in April, 1987.

tzset uses the contents of the environment variable **TZ** to override the value of the different external variables. The syntax of **TZ** is described as:

```
TZ                    →        zone
                           | zone signed_time
                           | zone signed_time zone
                           | zone signed_time zone dst
zone                  →        letter letter letter
signed_time           →        sign time
                           | time
time                  →        hour
                           | hour : minute
                           | hour : minute : second
dst                   →        signed_time
                           | signed_time; dst_date,dst_date
                           | ; dst_date , dst_date
dst_date              →        julian
                           | julian / time
letter                →      a | A | b | B | ... | z | Z
hour                  →     00 | 01 | ... | 23
minute                →     00 | 01 | ... | 59
second                →     00 | 01 | ... | 59
```

```
julian          →          001 | 002 | ...| 366
sign            →                     - | +
```

tzset scans the contents of the environment variable and assigns the different fields to the respective variable. For example, the setting for New Jersey in 1986 could be:

"EST5EDT4;117/2:00:00,299/2:00:00" .

or simply

EST5EDT

A southern hemisphere setting such as the Cook Islands could be:

i"KDT9:30KST10:00;64/5:00,303/20:00"|f1

When the longer format is used, the variable must be surrounded by double quotes as shown. For more details, see *timezone*(4) and *environ*(5). In the longer version of the New Jersey example of **TZ**, *tzname*[0] is EST, *timezone* will be set to 5*60*60, *tzname*(1) is EDT, *altzone* will be set to 4*60*60, the starting date of the alternate time zone is the 117th day at 2 AM, the ending date of the alternate time zone is the 299th day at 2 AM, and *daylight* will be set to non-zero.

Starting and ending times are relative to the alternate time zone. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be midnight. The effects of *tzset* are thus to change the values of the external variables *timezone*, *altzone*, *daylight* and *tzname*. *tzset* is called by *localtime* and may also be called explicitly by the user.

Note that in most installations, **TZ** is set to the correct value by default when the user logs on, via the local **/etc/profile** file (see *profile*(4)).

FILES

/lib/cftime directory that contains the language specific printable files

SEE ALSO

time(2), getenv(3C), putenv(3C), printf(3S), cftime(4), profile(4), timezone(4), environ(5).

CAVEAT

The return values for *ctime*, *localtime* and *gmtime* point to static data whose content is overwritten by each call.

Setting the time during the interval of change from *timezone* to *altzone* or vice versa can produce unpredictable results.

The system administrator must change the Julian start and end days annually if the full form of the **TZ** variable is specified.

NAME

isdigit, isxdigit, islower, isupper, isalpha, isalnum, isspace, iscntrl, ispunct, isprint, isgraph, isascii, tolower, toupper, toascci, _tolower, _toupper, setchrcla

SYNOPSIS

#include <ctype.h>

int isdigit (c);
int c;

. . .

tolower(c)
int c;

. . .

int setchrclass (*chrclass*)
char **chrclass;*

DESCRIPTION

The character classification macros listed below return nonzero for true, zero for false. *isascii* is defined on all integer values; the rest are defined on valid members of the character set and on the single value **EOF** [see *stdio*(3S)] (guaranteed not to be a character set member).

isdigit	tests for the digits 0 through 9.
isxdigit	tests for any character for which *isdigit* is true or for the letters *a* through *f* or *A* through *F*.
islower	tests for any lowercase letter as defined by the character set.
isupper	tests for any uppercase letter as defined by the character set.
isalpha	tests for any character for which *islower* or *isupper* is true and possibly any others as defined by the character set.
isalnum	tests for any character for which *isalpha* or *isdigit* is true.
isspace	tests for a space, horizontal-tab, carriage return, newline, vertical-tab, or form-feed.
iscntrl	tests for "control characters" as defined by the character set.

ispunct	tests for any character other than the ones for which *isalnum, iscntrl,* or *isspace* is true or space.
isprint	tests for a space or any character for which *isalnum* or *ispunct* is true or other "printing character" as defined by the character set.
isgraph	tests for any character for which *isprint* is true, except for space.
isascii	tests for an ASCII character (a non-negative number less than 0200.)

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	if the character is one for which *isupper* is true and there a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.
toupper	if the character is one for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.
toascii	turns off the bits that are not part of the ASCII character set.
_tolower	returns the lowercase representation of a character for which *isupper* is true, otherwise undefined.
_toupper	returns the uppercase representation of a character for which *islower* is true, otherwise undefined.

The conversion macros have the same functionality of the functions on valid input, but the macros are faster because they do not do range checking.

All the character classification macros and the conversion functions and macros do a table lookup.

setchrclass initializes the table used by these functions and macros to a specific character classification set. *setchrclass* uses the value of its argument or the value of the environment variable **CHRCLASS** as the name of the datafile containing the information for the desired character set. These datafiles are searched for in the special directory **/lib/chrclass**.

If ***chrclass*** is (char *)0, the value of the environment variable **CHRCLASS** is used. If **CHRCLASS** is not set or is undefined, the table retains its current value, which at initialization time is **ascii**.

FILES

/lib/chrclass directory containing the datafiles for *setchrclass*

SEE ALSO

chrtbl(1), stdio(3S), ascii(5), environ(5).

DIAGNOSTICS

If the argument to any of the character handling macros is not in the domain of the function, the result is undefined.

If *setchrclass* does not successfully fill the table, the table will not change (initially "ascii") and –1 is returned. If everything works, *setchrclass* returns 0.

NAME

dial – establish an out-going terminal line connection

SYNOPSIS

#include <dial.h>

int dial (*call*)
CALL *call;*

void undial (*fd*)
int *fd;*

DESCRIPTION

dial returns a file-descriptor for a terminal line open for read/write. The argument to *dial* is a CALL structure (defined in the **<dial.h>** header file).

When finished with the terminal line, the calling program must invoke *undial* to release the semaphore that has been set during the allocation of the terminal device.

The definition of CALL in the **<dial.h>** header file is:

```
typedef struct {
  struct termio *attr;   /* pointer to termio attribute struct */
  int           baud;   /* transmission data rate */
  int           speed; /* 212A modem: low=300, high=1200 */
  char          *line;   /* device name for out-going line */
  char          *telno; /* pointer to tel-no digits string */
  int           modem; /* specify modem control for direct lines */
  char          *device; /* unused */
  int           dev_len;   /* unused */
} CALL;
```

The CALL element *speed* is intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high- or low-speed setting on the 212A modem. Note that the 113A modem or the low-speed setting of the 212A modem will transmit at any rate between 0 and 300 bits per second. However, the high-speed setting of the 212A modem transmits and receives at 1200 bits per second only. The CALL element *baud* is for the desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200). However, if *speed* is set to 1200, *baud* must be set to high (1200).

If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the *line* element in the CALL structure. Legal values for such terminal device names are kept in the **Devices** file. In this case, the value of the *baud* element should be set to -1. This causes **dial** to determine the correct value from the **Devices** file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of these characters:

0-9	dial 0-9
*****	dial *
#	dial #
=	wait for secondary dail tone
–	delay for approximately 4 seconds

The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element *attr* is a pointer to a *termio* structure, as defined in the **termio.h** header file. A **NULL** value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it are set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

The CALL elements device and *dev_len* are no longer used. They are retained in the CALL structure for compatibility reasons.

FILES

/usr/lib/uucp/Devices
/usr/lib/uucp/Systems
/usr/spool/uucp/LCK..tty-device

SEE ALSO

alarm(2), read(2), write(2)
acu(7), termio(7) in the *System Administrator's Reference Manual*.
uucp(1C) in the *User's Reference Manual*.

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure is returned. Mnemonics for these negative indices as listed here are defined in the **<dial.h>** header file.

```
INTRPT   -1   /* interrupt occurred */
D_HUNG   -2   /* dialer hung (no return from write) */
NO_ANS   -3   /* no answer within 10 seconds */
ILL_BD   -4   /* illegal baud-rate */
A_PROB   -5   /* acu problem (open() failure) */
L_PROB   -6   /* line problem (open() failure) */
NO_Ldv   -7   /* cannott open Devices file */
DV_NT_A -8    /* requested device not available */
DV_NT_K -9    /* requested device not known */
NO_BD_A -10   /* no device available at requested baud */
NO_BD_K -11   /* no device known at requested baud */
DV_NT_E -12   /* requested speed does not match */
BAD_SYS -13   /* system not in Systems file*/
```

WARNINGS

Including the **dial.h** header file automatically includes the **termio.h** header file.

The above routine uses **stdio.h**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

An *alarm*(2) system call for 3600 seconds is made (and caught) within the *dial* module for the purpose of "touching" the *LCK*.. file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1C) may simply delete the *LCK*.. entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a *read*(2) or *write*(2) system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from *read*s should be checked for **(errno==EINTR)**, and the *read* possibly reissued.

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

double drand48 ( )

double erand48 (*xsubi*)
unsigned short *xsubi* **[3];**

long lrand48 ( )

long nrand48 (*xsubi*)
unsigned short *xsubi* **[3];**

long mrand48 ( )

long jrand48 (*xsubi*)
unsigned short *xsubi* **[3];**

void srand48 (*seedval*)
long *seedval;*

unsigned short *seed48 (*seed16v*)
unsigned short *seed16v* **[3];**

void lcong48 (*param*)
unsigned short *param* **[7];**

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval $[0.0,~1.0)$.

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval $[0,~2^{31})$.

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Functions *srand48, seed48,* and *lcong48* are initialization entry points, one of which should be invoked before either *drand48, lrand48,* or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48, lrand48* or *mrand48* is called without a prior call to an initialization entry point.) Functions *erand48, nrand48,* and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \qquad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed Unless *lcong48* has been invoked, the multiplier value a and the addenc value c are given by

$$a = 5DEECE66D_{16} = 273673163155_8$$
$$c = B_{16} = 13_8.$$

The value returned by any of the functions *drand48, erand48, lrand48, nrand48, mrand48,* or *jrand48* is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (left-most) bits of X_i and transformed into the returned value.

The functions *drand48, lrand48,* and *mrand48* store the last 48-bit X_i generated in an internal buffer, and must be initialized prior to being invoked. The functions *erand48, nrand48,* and *jrand48* require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. These routines do not have to be initialized; the calling program must place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions *erand48, nrand48,* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function *seed48* sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48,* and a pointer to this buffer is the value returned by *seed48*. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize via *seed48* when the program is restarted.

The initialization function *lcong48* allows the user to specify the initial X_i, the multiplier value *a*, and the addend value *c*. Argument array elements *param[0-2]* specify X_i, *param[3-5]* specify the multiplier *a*, and *param[6]* specifies the 16-bit addend *c*. After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values, *a* and *c*, specified on the previous page.

NOTES

The source code for the portable version can be used on computers which do not have floating-point arithmetic. In such a situation, functions *drand48* and *erand48* are replaced by the two new functions below.

long irand48 (m)
unsigned short m;

long krand48 (xsubi, m)
unsigned short xsubi[3], m;

Functions *irand48* and *krand48* return non-negative long integers uniformly distributed over the interval $[0, m-1]$.

SEE ALSO

rand(3C).

NAME

dup2 – duplicate an open file descriptor

SYNOPSIS

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

DESCRIPTION

fildes is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than NOFILES. *dup2* causes *fildes2* to refer to the same file as *fildes.* If *fildes2* already referred to an open file, it is closed first.

dup2 will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid open file descriptor.
[EMFILE]	NOFILES file descriptors are currently open.

SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2), lockf(3C).

DIAGNOSTICS

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

NAME

ecvt, fcvt, gcvt – convert floating-point number to string

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;*
int *ndigit;*
char **buf;*

DESCRIPTION

ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

fcvt is identical to *ecvt*, except that the correct digit has been rounded for printf "%f" (FORTRAN F-format) output of the number of digits specified by *ndigit*.

gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

SEE ALSO

printf(3S).

BUGS

The values returned by *ecvt* and *fcvt* point to a single static data array whose content is overwritten by each call.

NAME

end, etext, edata – last locations in program

SYNOPSIS

extern end;
extern etext;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with *end*, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output (*stdio*(3S)), the profile (–p) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk (char *)(0)** (see *brk*(2)).

SEE ALSO

cc(1), brk(2), malloc(3C), stdio(3S).

NAME

frexp, ldexp, modf – manipulate parts of floating-point numbers

SYNOPSIS

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

double ldexp (*value, exp*)
double *value;*
int *exp;*

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

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \leq |x| < 1.0$, and the "exponent" n is an integer. *frexp* returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by *frexp* are zero.

ldexp returns the quantity $value * 2^{exp}$.

modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If *ldexp* would cause overflow, ±**HUGE** (defined in **<math.h>**) is returned (according to the sign of *value*), and *errno* is set to **ERANGE**.
If *ldexp* would cause underflow, zero is returned and *errno* is set to **ERANGE.**

NAME

ftw – walk a file tree

SYNOPSIS

#include <ftw.h>

int ftw (*path, fn, depth*)
char **path;*
int (**fn*) ( );
int *depth;*

DESCRIPTION

ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure (see *stat*(2)) containing information about the object, and an integer. Possible values of the integer, defined in the **<ftw.h>** header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within *ftw* (i.e., an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns –1, and sets the error type in *errno*.

ftw uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *Depth* must not be greater than the number of file descriptors currently available for use. *ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

CAVEAT

ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

NAME

getcwd – get path-name of current working directory

SYNOPSIS

char *getcwd (*buf*, *size*)
char **buf*;
int *size*;

DESCRIPTION

getcwd returns a pointer to the current directory path name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free*.

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
void exit(), perror();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(2);
}
printf("%s\n", cwd);
```

SEE ALSO

malloc(3C), popen(3S).
pwd(1) in the *User's Reference Manual*.

DIAGNOSTICS

Returns NULL with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

NAME

getenv – return value for environment name

SYNOPSIS

char *getenv (*name*)
**char ****name;*

DESCRIPTION

getenv searches the environment list [see *environ*(5)] for a string of the form *name*=*value*, and returns a pointer to the *value* in the current environment if such a string is present, otherwise a **NULL** pointer.

SEE ALSO

exec(2), putenv(3C), environ(5).

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent – get group file entry

SYNOPSIS

#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (*gid*)
int *gid*;

struct group *getgrnam (*name*)
char ***name*;

void setgrent ( )

void endgrent ( )

struct group *fgetgrent (f)
FILE *f;

DESCRIPTION

getgrent, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the **/etc/group** file. Each line contains a "group" structure, defined in the <*grp.h*> header file.

```
struct group {
        char    *gr_name;       /* the name of the group */
        char    *gr_passwd;     /* the encrypted group password */
        int     gr_gid;         /* the numerical group ID */
        char    **gr_mem;       /* vector of pointers to member names */
};
```

getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *Getgrgid* searches from the beginning of the file until a numerical group id matching *gid* is found and returns a pointer to the particular structure in which it was found. *getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.
A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *endgrent* may be called to close the group file when processing is complete.

fgetgrent returns a pointer to the next group structure in the stream *f*, which matches the format of **/etc/group**.

FILES

/etc/group

SEE ALSO

getlogin(3C), getpwent(3C), group(4).

DIAGNOSTICS

A **NULL** pointer is returned on EOF or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getlogin – get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

getlogin returns a pointer to the login name as found in **/etc/utmp**. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns a **NULL** pointer. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

DIAGNOSTICS

Returns the **NULL** pointer if *name* is not found.

CAVEAT

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

NAME

getopt – get option letter from argument vector

SYNOPSIS

```
int getopt (argc, argv, optstring)
int argc;
char **argv, *opstring;

extern char *optarg;
extern int optind, opterr;
```

DESCRIPTION

getopt returns the next option letter in *argv* that matches a letter in *optstring*. It supports all the rules of the command syntax standard (see *intro*(1)). So all new commands will adhere to the command syntax standard, they should use *getopts*(1) or *getopt*(3C) to parse positional parameters and check for options that are legal for that command.

optstring must contain the option letters the command using *getopt* will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must 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. **optind** is external and is initialized to **1** before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **–1**. The special option "—" may be used to delimit the end of the options; when it is encountered, **–1** will be returned, and "—" will be skipped.

DIAGNOSTICS

getopt prints an error message on standard error and returns a question mark (?) when it encounters an option letter not included in *optstring* or no option-argument after an option that expects one. This error message may be disabled by setting **opterr** to **0**.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the option **o**, which requires an option-argument:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern char *optarg;
        extern int optind;
        .
        .
        .
        while ((c = getopt(argc, argv, "abo:")) != -1)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc( );
                        break;
                case 'o':
                        ofile = optarg;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                (void)fprintf(stderr, "usage: . . . ");
                exit (2);
        }
        for ( ; optind < argc; optind++) {
                if (access(argv[optind], 4)) {
        .
        .
        .
}
```

WARNING

Although the following command syntax rule (see *intro*(1)) relaxations are permitted under the current implementation, they should not be used because they may not be supported in future releases of the system. As in the **EXAMPLE** section above, **a** and **b** are options, and the option **o** requires an option-argument:

```
cmd –aboxxx file   (Rule 5 violation:  options with
        option-arguments must not be grouped with other options)
cmd –ab –oxxx file   (Rule 6 violation:  there must be
        white space after an option that takes an option-argument)
```

SEE ALSO

getopts(1), intro(1) in the *User's Reference Manual*.

WARNING

Although the following command syntax rule (see *intro*(1)) relaxations are permitted under the current implementation, they should not be used because they may not be supported in future releases of the system. As in the following example, **a** and **b** are options, and the option **o** requires an option-argument:

Example:

The following fragment of a shell program shows how one might process the arguments for a command that can take the options **a** or **b**, as well as the option **o**, which requires an option-argument:

```
while getopts abo: c
do
        case $c in
        a | b)      FLAG=$c;;
        o)          OARG=$OPTARG;;
        \?)         echo $USAGE
                    exit 2;;
        esac
done
shift expr $OPTIND - 1
```

This code will accept any of the following as equivalent:

```
cmd –a –b –o "xxx z yy" file
cmd –a –b –o "xxx z yy" — file
cmd –ab –o xxx,z,yy file
cmd –ab –o "xxx z yy" file
cmd –o xxx,z,yy –b –a file
cmd –aboxxx file  (Rule 5 violation:  options with
        option-arguments must not be grouped with other options)
cmd –ab –oxxx file  (Rule 6 violation:  there must be
        white space after an option that takes an option-argument)
```

Changing the value of the shell variable **OPTIND** or parsing different sets of arguments may lead to unexpected results.

Changing the value of the variable **optind**, or calling *getopt* with different values of *argv*, may lead to unexpected results.

NAME

getpass – read a password

SYNOPSIS

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

DESCRIPTION

getpass reads up to a newline or EOF from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a **NULL** pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

FILES

/dev/tty

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs not otherwise using standard I/O, more than might be expected.

CAVEAT

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

NAME

getpw – get name from UID

SYNOPSIS

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

DESCRIPTION

getpw searches the password file for a user id number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. *getpw* returns non-zero if *uid* cannot be found.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

getpw returns non-zero on error.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase, more than might be expected, the size of programs not otherwise using standard I/O.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent – get password file entry

SYNOPSIS

#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (*uid*)
int *uid;*

struct passwd *getpwnam (*name*)
char **name;*

void setpwent ( )

void endpwent ( )

struct passwd *fgetpwent (*f*)
FILE **f;*

DESCRIPTION

getpwent, *getpwuid* and *getpwnam* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/passwd** file. Each line in the file contains a "passwd" structure, declared in the **<pwd.h>** header file:

```
struct passwd {
        char  *pw_name;
        char  *pw_passwd;
        int   pw_uid;
        int   pw_gid;
        char  *pw_age;
        char  *pw_comment;
        char  *pw_gecos;
        char  *pw_dir;
        char  *pw_shell;
};
```

This structure is declared in **<pwd.h>** so it is not necessary to redeclare it.

The fields have meanings described in *passwd*(4).

getpwent when first called returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found and returns a pointer to the particular structure in which it was found. *getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an EOF or an error is encountered on reading, these functions return a **NULL** pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *endpwent* may be called to close the password file when processing is complete.

fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of **/etc/passwd**.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(4).

DIAGNOSTICS

A **NULL** pointer is returned on EOF or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

AME

gettimeofday – get time of day

YNOPSIS

#include <sys/time.h>

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

DESCRIPTION

The system's notion of the current Greenwich time and the current time zone is obtained with the *gettimeofday*() call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware dependent. If *tzp* is NULL, the current time zone information is not returned. If *tp* is NULL, the Greenwich time information is not returned.

RETURN VALUE

A 0 return value indicates the call succeeded. A –1 return value indicates an error occurred.

SEE ALSO

time(2), ctime(3C).

NAME

getut: getutent, getutid, getutline, pututline, setutent, endutent, utmpname – access utmp file entry

SYNOPSIS

#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (*id*)
**struct utmp ****id;*

struct utmp *getutline (*line*)
**struct utmp ****line;*

void pututline (*utmp*)
**struct utmp ****utmp;*

void setutent ( )

void endutent ( )

void utmpname (*file*)
**char ****file;*

DESCRIPTION

getutent, *getutid* and *getutline* each return a pointer to a structure of the following type:

```
struct utmp {
        char    ut_user[8];  /* User login name */
        char    ut_id[4];    /* /etc/inittab id (usually line #) */
        char    ut_line[12]; /* device name (console, lnxx) */
        short   ut_pid;      /* process id */
        short   ut_type;     /* type of entry */
        struct  exit_status {
            short  e_termination; /* Process termination status */
            short  e_exit;        /* Process exit status */
           } ut_exit;                /* The exit status of a process
                                     /* marked as DEAD_PROCESS. */
           time_t   ut_time;         /* time entry was made */
};
```

getutent reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

getutid searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id–>ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS, then *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id–>ut_id*. If the end of file is reached without a match, it fails.

getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the *line–>ut_line* string. If the end of file is reached without a match, it fails.

pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

endutent closes the currently open file.

utmpname allows the user to change the name of the file examined, from **/etc/utmp** to any other file. It is most often expected that this other file will be **/etc/wtmp**. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. *utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

NOTES

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

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 character should be cast to pointer-to-character.) *action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a **NULL** pointer.

hcreate allocates sufficient space for the table, and must be called before *hsearch* is used. *nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

hdestroy destroys the search table, and may be followed by another call to *hcreate*.

NOTES

hsearch uses *open addressing* with a *multiplicative* hash function. However, its source code has many other options available which the user may select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV
Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.

USCR
Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a mannner similar to *strcmp* (see *string*(3C)).

CHAINED
Use a linked list to resolve collisions. If this option is selected, the following other options become available:

START
Place new entries at the beginning of the linked list (default is at the end).

SORTUP
Keep the linked list sorted by key in ascending order.

SORTDOWN
Keep the linked list sorted by key in descending order.

In addition, there are preprocessor flags for obtaining debugging printout (**–DDEBUG**) and for including a test driver in the calling routine (**–DDRIVER**). Consult the source code for further details.

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

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

struct info {    /* this is the info stored in the table */
    int age, room;     /* other than the key. */
};
#define NUM_EMPL 5000 /* # of elements in search table */

main( )
{
        /* space to store strings */
        char string_space[NUM_EMPL*20];
        /* space to store employee info */
        struct info info_space[NUM_EMPL];
        /* next avail space in string_space */
        char *str_ptr = string_space;
        /* next avail space in info_space */
        struct info *info_ptr = info_space;
        ENTRY item, *found_item, *hsearch( );
        /* name to look for in table */
        char name_to_find[30];
```

```
            int i = 0;

            /* create table */
            (void) hcreate(NUM_EMPL);
            while (scanf("%s%d%d", str_ptr, &info_ptr->age,
                  &info_ptr->room) != EOF && i++ < NUM_EMPL) {
           /* put info in structure, and structure in item */
           item.key = str_ptr;
           item.data = (char *)info_ptr;
           str_ptr += strlen(str_ptr) + 1;
           info_ptr++;
           /* put item into table */
           (void) hsearch(item, ENTER);
            }

            /* access table */
            item.key = name_to_find;
            while (scanf("%s", item.key) != EOF) {
              if ((found_item = hsearch(item, FIND)) != NULL) {
             /* if item is in the table */
             (void)printf("found %s, age = %d, room = %d\n",
                   found_item->key,
                   ((struct info *)found_item->data)->age,
                   ((struct info *)found_item->data)->room);
               } else {
                   (void)printf("no such employee %s\n",
                          name_to_find)
               }
            }
     }
```

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), malloc(3X), string(3C), tsearch(3C).

DIAGNOSTICS

hsearch returns a **NULL** pointer if either the action is **FIND** and the item could not be found or the action is **ENTER** and the table is full.

hcreate returns zero if it cannot allocate sufficient space for the table.

WARNING

hsearch and *hcreate* use *malloc*(3C) to allocate space.

CAVEAT

Only one hash search table may be active at any given time.

NAME

initgroups – initialize group access list

SYNOPSIS

initgroups (*name*, *basegid*)
char * *name;*
gid_t *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.

RETURN VALUE

initgroups returns –1 if it was not invoked by the superuser.

FILES

/etc/group
/etc/passwd

SEE ALSO

setgroups(2).

BUGS

initgroups uses the routines based on *getgrent (3)*. If the invoking program uses any of these routines, the group structure will be overwritten in the call to *initgroups* .

NAME

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

SYNOPSIS

void l3tol (*lp, cp, n*)
long **lp;*
char **cp;*
int *n;*

void ltol3 (*cp, lp, n*)
char **cp;*
long **lp;*
int *n;*

DESCRIPTION

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

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

These functions are useful for file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(4).

CAVEAT

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

lockf – record locking on files

SYNOPSIS

```
#include <unistd.h>

int lockf (fildes, function, size)
long size;
int fildes, function;
```

DESCRIPTION

The *lockf* command will allow sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file (see *chmod*(2)). Locking calls from other processes which attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. (See *fcntl*(2) for more information about record locking.)

fildes is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish lock with this function call.

function is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>**:

```
#define  F_ULOCK  0  /* Unlock a previously locked section */
#define  F_LOCK   1  /* Lock a section for exclusive use */
#define  F_TLOCK  2  /* Test and lock a section for exclusive use */
#define  F_TEST   3  /* Test section for other processes locks */
```

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

size is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset through the largest file offset is locked (i.e., from the current offset through the present or any future EOF). An area need not be allocated to the file in order to be locked as such locks may exist past the EOF.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections occur, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK will cause the calling process to sleep until the resource is available. F_TLOCK will cause the function to return a –1 and set *errno* to [EACCES] error if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus, calls to *lockf* or *fcntl* scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(2) command may be used to provide a timeout facility in applications which require this facility.

The *lockf* utility will fail if one or more of the following are true:

[EBADF]
fildes is not a valid open descriptor.

[EACCES]
cmd is F_TLOCK or F_TEST and the section is already locked by another process.

[EDEADLK]
cmd is F_LOCK and a deadlock would occur. Also the *cmd* is either F_LOCK, F_TLOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.

[ECOMM]
fildes is on a remote machine and the link to that machine is no longer active.

SEE ALSO
chmod(2), close(2), creat(2), fcntl(2), intro(2), open(2), read(2), write(2).

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.

WARNINGS
Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The standard I/O package is the most common source of unexpected buffering.

Because in the future the variable *errno* will be set to EAGAIN instead of EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

NAME

lsearch, lfind – linear search and update

SYNOPSIS

#include <stdio.h>
#include <search.h>

char *lsearch ((*char* *)*key*, (*char* *)*base*, *nelp*, *sizeof*(**key*), *compar*)
unsigned **nelp*;
int (**compar*)( );

char *lfind ((*char* *)*key*, (*char* *)*base*, *nelp*, *sizeof*(**key*), *compar*)
unsigned **nelp*;
int (**compar*)( );

DESCRIPTION

lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. *key* points to the datum to be sought in the table. *base* points to the first element in the table. *nelp* points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. *compar* is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a **NULL** pointer is returned.

NOTES

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

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

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

EXAMPLE

This fragment will read in less than TABSIZE strings of length less than ELSIZE and store them in a table, eliminating duplicates.

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

#define TABSIZE 50
#define ELSIZE 120

      char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
      unsigned nel = 0;
      int strcmp( );
      . . .
      while (fgets(line, ELSIZE, stdin) != NULL &&
         nel < TABSIZE)
            (void) lsearch(line, (char *)tab, &nel,
                    ELSIZE, strcmp);
      . . .
```

SEE ALSO

bsearch(3C), hsearch(3C), string(3C), tsearch(3C).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns **NULL** and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

malloc, free, realloc, calloc, mallopt, mallinfo – fast main memory allocator

SYNOPSIS

#include <malloc.h>

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

void free (*ptr*)
char **ptr;*

char *realloc (*ptr, size*)
char **ptr;*
unsigned *size;*

char *calloc (*nelem, elsize*)
unsigned nelem, *elsize;*

int mallopt (*cmd, value*)
int cmd, *value;*

struct mallinfo mallinfo( )

DESCRIPTION

malloc and *free* provide a simple general-purpose memory allocation package. *malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed, this space is made available for further allocation and its contents destroyed (see *mallopt* below for a way to change this behavior).

Undefined results occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*. *

malloc returns NULL if an attempt is made to allocate 0 bytes.

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 are unchanged up to the lesser of the new and old sizes. *

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

mallopt provides for control over the allocation algorithm. The available values for *cmd* are:

M_MXFAST
: Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups, then doles them out very quickly. The default value for *maxfast* is 24.

M_NLBLKS
: Set *numlblks* to *value*. The above mentioned *large groups* each contain *numlblks* blocks. *numlblks* must be greater than 0. The default value for *numlblks* is 100.

M_GRAIN
: Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *grain* must be greater than 0. The default value of *grain* is the smallest number of bytes which allows alignment of any data type. Value is rounded up to a multiple of the default when *grain* is set.

M_KEEP
: Preserve data in a freed block until the next *malloc*, *realloc*, or *calloc*. This option is provided only for compatibility with the old version of *malloc* and is not recommended.

These values are defined in the **<malloc.h>** header file.

mallopt may be called repeatedly, but may not be called after the first small block is allocated.

mallinfo provides instrumentation describing space usage. It returns the structure:

```
struct mallinfo  {
        int arena;     /* total space in arena */
        int ordblks;   /* number of ordinary blocks */
        int smblks;    /* number of small blocks */
        int hblkhd;    /* space in holding block headers */
        int hblks;     /* number of holding blocks */
        int usmblks;   /* space in small blocks in use */
        int fsmblks;   /* space in free small blocks */
        int uordblks;  /* space in ordinary blocks in use */
        int fordblks;  /* space in free ordinary blocks */
        int keepcost;  /* space penalty if keep option */
                       /* is used */
  }
```

This structure is defined in the **<malloc.h>** header file.

Each allocation routine returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

:E ALSO

brk(2)

IAGNOSTICS

malloc, *realloc*, and *calloc* return a NULL pointer if there is not enough available memory. When *realloc* returns NULL, the block pointed to by *ptr* is left intact. If *mallopt* is called after any allocation or if *cmd* or *value* are invalid, nonzero is returned; otherwise, it returns zero.

*

/ARNINGS

Note that this package does not preserve the contents of a block when it is freed, unless the M_KEEP option of *mallopt* is used.

NAME

memory: memccpy, memchr, memcmp, memcpy, memset – memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

memccpy copies characters from memory area **s2** into **s1**, stopping after the first occurrence of character **c** has been copied, or after **n** characters have been copied, whichever comes first. It returns a pointer to the character after the copy of **c** in **s1**, or a `NULL` pointer if **c** was not found in the first **n** characters of **s2**.

memchr returns a pointer to the first occurrence of character **c** in the first **n** characters of memory area **s**, or a `NULL` pointer if **c** does not occur.

memcmp compares its arguments, looking at the first **n** characters only, and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**.

memcpy copies **n** characters from memory area **s2** to **s1**. It returns **s1**.

memset sets the first **n** characters in memory area **s** to the value of character **c**. It returns **s**.

For user convenience, all these functions are declared in the optional **<memory.h>** header file.

CAVEATS

memcmp is implemented by using the most natural character comparison on the machine. Thus the sign of the value returned when one of the characters has its high order bit set is not the same in all implementations and should not be relied upon.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

mktemp – make a unique file name

SYNOPSIS

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

DESCRIPTION

mktemp replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing Xs; *mktemp* will replace the Xs with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

DIAGNOSTIC

mktemp will assign to *template* the **NULL** string if it cannot create a unique name.

CAVEAT

If called more than 17,576 time in a single process, this function will start recycling previously used names.

NAME

monitor – prepare execution profile

SYNOPSIS

#include <mon.h>

void monitor (*lowpc, highpc, buffer, bufsize, nfunc*)
int (**lowpc*)( ), (**highpc*)( );
WORD **buffer;*
int bufsize, *nfunc;*

DESCRIPTION

An executable program created by **cc –p** automatically includes calls for *monitor* with default parameters; *monitor* need not be called explicitly except to gain fine control over profiling.

monitor is an interface to *profil*(2). *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* WORDs (defined in the <*mon.h*> header file). *monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *lowpc* may not equal 0 for this use of *monitor*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **–p** of *cc*(1) are recorded.

For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use:

```
extern etext;
...
monitor ((int (*)())2, &etext, buf, bufsize, nfunc);
```

etext lies just above all the program text; see *end*(3C).

To stop execution monitoring and write the results, use:

```
monitor ((int (*)())0, 0, 0, 0, 0);
```

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

The name of the file written by *monitor* is controlled by the environment variable PROFDIR. If PROFDIR does not exist, "mon.out" is created in the current directory. If PROFDIR exists but has no value, *monitor* does not do any profiling and creates no output file. Otherwise, the value of PROFDIR is used as the name of the directory in which to create the output file. If PROFDIR is *dirname*, then the file written is "*dirname*/*pid*.mon.out" where *pid* is the program's process id. (When *monitor* is called automatically by compiling via **cc –p**, the file created is "*dirname*/*pid.progname*" where *progname* is the name of the program.)

FILES

mon.out

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

BUGS

The "*dirname*/*pid*.**mon.out**" form does not work;
the "*dirname*/*pid.progname*" form (automatically called via **cc –p**) does work.

NAME

nlist – get entries from name list

SYNOPSIS

#include <nlist.h>

int nlist (filename, nl)
char *filename;
struct nlist *nl;

DESCRIPTION

nlist examines the name list in the executable file whose name is pointed to by *filename*, and selectively extracts a list of values and puts them in the array of nlist structures pointed to by *nl*. The name list *nl* consists of an array of structures containing names of variables, types and values. The list is terminated with a null name; that is, a null string is in the name position of the structure. Each variable 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. The type field will be set to 0 unless the file was compiled with the –g option. If the name is not found, both entries are set to 0. See *a.out*(4) for a discussion of the symbol table structure.

This function is useful for examining the system name list kept in the file **/unix**. In this way programs can obtain system addresses that are up to date.

NOTES

The **<nlist.h>** header file is automatically included by **<a.out.h>** for compatability. However, if the only information needed from **<a.out.h>** is for use of *nlist*, then including **<a.out.h>** is discouraged. If **<a.out.h>** is included, the line "**`#undef n_name`**" may need to follow it.

SEE ALSO

a.out(4).

DIAGNOSTICS

All value entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

nlist returns –1 upon error; otherwise, it returns 0.

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

void perror (*s*)
char **s;*

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;

DESCRIPTION

perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. (However, if s="" the colon is not printed.) To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index into this table to get the message string without the new-line. *sys_nerr* is the number of messages in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2).

NAME

putenv – change or add value to environment

SYNOPSIS

int putenv (*string*)
char **string;*

DESCRIPTION

string points to a string of the form "*name=value.*" *putenv* makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

SEE ALSO

exec(2), getenv(3C), malloc(3C), environ(5).

DIAGNOSTICS

putenv returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise, zero.

WARNINGS

putenv manipulates the environment pointed to by *environ*, and can be used with *getenv*. However, *envp* (the third argument to *main*) is not changed.

This routine uses *malloc*(3C) to enlarge the environment.

After *putenv* is called, environmental variables are not in alphabetical order.

A potential error is to call *putenv* with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

putpwent is the inverse of *getpwent*(3C). Given a pointer to a passwd structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwent* writes a line on the stream *f*, which matches the format of **/etc/passwd**.

SEE ALSO

getpwent(3C).

DIAGNOSTICS

putpwent returns non-zero if an error was detected during its operation, otherwise, zero.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

NAME

qsort – quicker sort

SYNOPSIS

void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

base points to the element at the base of the table. *nel* is the number of elements in the table. *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. As the function must return an integer less than, equal to, or greater than zero, so must the first argument to be considered be less than, equal to, or greater than the second.

NOTES

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

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

The order in the output of two items which compare as equal is unpredictable.

SEE ALSO

bsearch(3C), lsearch(3C), string(3C).

sort(1) in the *User's Reference Manual.*

NAME

rand, srand – simple random-number generator

SYNOPSIS

int rand ( )

void srand (*seed*)
unsigned *seed;*

DESCRIPTION

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

srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTES

The spectral properties of *rand* are limited. *drand48*(3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C).

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

#include <setjmp.h>

int setjmp (*env*)
jmp_buf *env;*

void longjmp (*env, val*)
jmp_buf *env;*
int *val;*

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

setjmp saves its stack environment in *env* (whose type, *jmp_buf*, is defined in the **<setjmp.h>** header file) for later use by *longjmp*. It returns the value 0.

longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. At the time of the second return from *setjmp*, all external and static variables have values as of the time *longjmp* is called (see example). The values of register and automatic variables are undefined.

In a future release, C language users will be able to identify syntactically those automatic variables on whose values they need to rely after the second return from *setjmp*.

EXAMPLE

```
#include <setjmp.h>

jmp_buf env;
int i = 0;
main ()
{
   void exit();

   if(setjmp(env) != 0) {
        (void) printf("value of i on 2nd return from setjmp: %d\n", i);
        exit(0);
        }
        (void) printf("value of i on 1st return from setjmp: %d\n", i);
        i = 1;
        g();
        /*NOTREACHED*/
}

g()

{
        longjmp(env, 1);
        /*NOTREACHED*/
}
```

If the **a.out** resulting from this C language code is run, the output is:

```
value of i on 1st return from setjmp:0

value of i on 2nd return from setjmp:1
```

SEE ALSO

signal(2).

WARNING

If *longjmp* is called even though *env* was never primed by a call to *setjmp*, or when the last such call was in a function that has since returned, absolute chaos is guaranteed.

NAME

setlocale – set or query current locale

SYNOPSIS

char *
setlocale (*category*, *locale*)
int *category*;
char **locale*;

DESCRIPTION

setlocale is defined by ANSI C. Since it is not defined by Common Usage C, it currently returns **NULL**.

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep (*seconds*)
unsigned *seconds;*

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the *sleep* routine returns. But if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2), pause(2), signal(2).

NAME

ssignal, gsignal – software signals

SYNOPSIS

#include <signal.h>

int (*ssignal (*sig, action*))( )
int *sig*, (**action*)( );

int gsignal (*sig*)
int *sig*;

DESCRIPTION

ssignal and *gsignal* implement a software facility similar to *signal*(2). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions, and is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 16. A call to *ssignal* associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) *action function* or one of the manifest constants **SIG_DFL** (default) or **SIG_IGN** (ignore). *ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns **SIG_DFL**.

gsignal raises the signal identified by its argument, *sig*:

> If an action function has been established for *sig*, then that action is reset to **SIG_DFL** and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.
>
> If the action for *sig* is **SIG_IGN**, *gsignal* returns the value 1 and takes no other action.
>
> If the action for *sig* is **SIG_DFL**, *gsignal* returns the value 0 and takes no other action.
>
> If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

SEE ALSO

signal(2), sigset(2).

NOTES

There are some additional signals with numbers outside the range 1 through 16 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 16 are legal, although their use may interfere with the operation of the Standard C Library.

NAME

stdipc: ftok – standard interprocess communication package

SYNOPSIS

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

key_t ftok (*path, id*)
char **path;*
char *id;*

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *path* must be the path name of an existing file that is accessible to the process. *id* is a character which uniquely identifies a project. Note that *ftok* will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

DIAGNOSTICS

ftok returns **(key_t) –1** if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to *ftok* is removed when keys still refer to the file, future calls to *ftok* with the same *path* and *id* will return an error. If the same file is recreated, then *ftok* is likely to return a different key than it did the original time it was called.

NAME

strftime – convert date and time to string

SYNOPSIS

```
#include <time.h>
int strftime (buf, max, fmt, tm)
char *buf, *fmt;
int max;
struct tm *tm;
```

DESCRIPTION

The *strftime* function places characters into the array pointed to by *buf* as controlled by the string pointed to by *fmt*. The *fmt* string consists of zero or more field descriptors and ordinary characters. A field descriptor consists of a % character followed by a character that determines the field descriptor's behavior. All ordinary characters (including the terminating NULL character) are copied unchanged into the array. No more than *maxsize* characters are placed into the array. Each field descriptor is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the values contained in the structure pointed to by *tm*.

%%
is replaced by %

%a
abbreviated weekday name

%A
full weekday name

%b
abbreviated month name

%B
full month name

%c
local specific date format

%d
day of month (01 - 31)

%H
hour (00 - 23)

%I
hour (00 - 12)

%j
day number of year (001 - 366)

%m
month number (01 - 12)

%M
minute (00 - 59)

%p
ante meridiem or post meridiem (AM or PM)

%S
seconds (00 - 59)

%U
week number of year (00 - 52); Sunday is the first day of week

%w
weekday number (Sunday = 0)

%W
week number of year (00 - 52); Monday is the first day of week

%x
local specific date format

%X
local specific time format

%y
year within century (00 - 99)

%Y
year as ccyy (e.g., 1986)

%Z
time zone name

The difference between **%U** and **%W** lies in which day is counted as the first of the week. Week number 01 is the first complete week starting on the designated day.

If the total number of resulting characters, including the terminating NULL character, is more than *max*, *strftime* returns the number of characters placed into the array pointed to by *buf* not including the terminating NULL character; otherwise, zero is returned and the contents of the array are indeterminate.

SEE ALSO

time(2), ctime(3C), cftime(4), timezone(4), environ(5).

NAME

string: strcat, strdup, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok – string operations

SYNOPSIS

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

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

char *strdup (s1)
char *s1;

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

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

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

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

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

int strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

char *strrchr (s, c)
char *s;
int c;

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

int strspn (s1, s2)
char *s1, *s2;
```

```
int strcspn (s1, s2)
char *s1, *s2;

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

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

DESCRIPTION

The arguments **s1, s2** and **s** point to strings (arrays of characters terminated by a NULL character). The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**.

strcat appends a copy of string **s2** to the end of string **s1**.

strdup returns a pointer to a new string that is a duplicate of the string pointed to by **s1**. The space for the new string is obtained using *malloc*(3C). If the new string cannot be created, NULL is returned.

strncat appends, at most, **n** characters. Each returns a pointer to the null-terminated result.

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

strcpy copies string **s2** to **s1**, stopping after the NULL character has been copied. *strncpy* copies exactly **n** characters, truncating **s2** or adding NULL characters to **s1** if necessary. The result is not null-terminated if the length of **s2** is **n** or more. Each function returns **s1**.

strlen returns the number of characters in **s**, not including the terminating NULL character.

strchr (*strrchr*) returns a pointer to the first (last) occurrence of character **c** in string **s**, or a NULL pointer if **c** does not occur in the string. The NULL character terminating a string is considered to be part of the string.

strpbrk returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a NULL pointer if no character from **s2** exists in **s1**.

strspn (*strcspn*) returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

strstr locates the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating NULL character) in the string pointed to by **s2**.

strtok considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2**. The first call (with pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a NULL character into **s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) work through the string **s1** immediately following that token. In this way, subsequent calls work through the string **s1** until no tokens remain. The separator string **s2** may be different from call to call. When no token remains in **s1**, a NULL pointer is returned.

For user convenience, all these functions are declared in the optional **<string.h>** header file.

SEE ALSO

malloc(3C)

CAVEATS

strcmp and *strncmp* are implemented by using the most natural character comparison on the machine; thus the sign of the value returned when one of the characters has its high-order bit set are not the same in all implementations and should not be relied upon.

Character movement is performed differently in different implementations; thus overlapping moves may yield surprises.

NAME

strtod, atof – convert string to double-precision number

SYNOPSIS

double strtod (*str, ptr*)
char ∗*str*, ∗∗*ptr*;

double atof (*str*)
char ∗*str*;

DESCRIPTION

strtod returns as a double-precision floating-point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

strtod recognizes an optional string of "white-space" characters (as defined by *isspace* in *ctype*(3C)), then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, **ptr* is set to *str*, and zero is returned.

atof(str) is equivalent to *strtod(str, (char **)NULL)*.

SEE ALSO

ctype(3C), scanf(3S), strtol(3C).

DIAGNOSTICS

If the correct value would cause overflow, ±**HUGE** (as defined in **<math.h>**) is returned (according to the sign of the value), and *errno* is set to **ERANGE.**

If the correct value would cause underflow, zero is returned and *errno* is set to **ERANGE.**

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

long strtol (*str, ptr, base*)
char **str, **ptr;*
int *base;*

long atol (*str*)
char **str;*

int atoi (*str*)
char **str;*

DESCRIPTION

strtol returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters (as defined by *isspace* in *ctype*(3C)) are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thusly: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment or by an explicit cast.

atol(str) is equivalent to *strtol(str, (char **)NULL, 10)*.

atoi(str) is equivalent to *(int) strtol(str, (char **)NULL, 10)*.

SEE ALSO

ctype(3C), scanf(3S), strtod(3C).

CAVEAT

Overflow conditions are ignored.

NAME

swab – swap bytes

SYNOPSIS

void swab (*from, to, nbytes*)
char **from*, **to;*
int *nbytes;*

DESCRIPTION

swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. *nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*–1 instead. If *nbytes* is negative, *swab* does nothing.

NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

SYNOPSIS

#include <search.h>

char *tsearch ((char *) *key*, (char **) *rootp*, *compar*)
int (**compar*)( );

char *tfind ((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, *tfind*, *tdelete*, and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

tsearch is used to build and access the tree. **key** is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. **rootp** points to a variable that points to the root of the tree. A NULL value for the variable pointed to by **rootp** denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by **rootp** will be changed if the deleted node was the root of the tree. *tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

twalk traverses a binary search tree. **root** is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum* { *preorder*, *postorder*, *endorder*, *leaf* } *VISIT;* (defined in the <*search.h*> header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

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

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

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

struct node {     /* pointers to these are stored in the tree */
       char *string;
       int length;
};
char string_space[10000];   /* space to store strings */
struct node nodes[500];     /* nodes to store */
struct node *root = NULL;   /* this points to the root */

main( )
{
       char *strptr = string_space;
       struct node *nodeptr = nodes;
       void print_node( ), twalk( );
       int i = 0, node_compare( );

       while (gets(strptr) != NULL && i++ < 500)  {
              /* set node */
              nodeptr->string = strptr;
              nodeptr->length = strlen(strptr);
              /* put node into the tree */
              (void) tsearch((char *)nodeptr, (char **) &root,
                      node_compare);
              /* adjust pointers, so we don't overwrite tree */
```

```
                        strptr += nodeptr->length + 1;
                        nodeptr++;
                }
                twalk((char *)root, print_node);
        }
        /*
                This routine compares two nodes, based on an
                alphabetical ordering of the string field.
        */
        int
        node_compare(node1, node2)
        char *node1, *node2;
        {
                return strcmp(((struct node *)node1)->string,
                ((struct node *) node2)->string);
        }
        /*
                This routine prints out a node, the first time
                twalk encounters it.
        */
        void
        print_node(node, order, level)
        char **node;
        VISIT order;
        int level;
        {
                if (order == preorder || order == leaf)  {
                        (void)printf("string = %20s,  length = %d\n",
                                (*((struct node **)node))->string,
                                (*((struct node **)node))->length);
                }
        }
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

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 *tfind* and *tdelete* if **rootp** is NULL on entry. If the datum is found, both *tsearch* and *tfind* return a pointer to it. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted item.

WARNINGS

The **root** argument to *twalk* is one level of indirection less than the rootp arguments to *tsearch* and *tdelete*.

There are two nomenclatures used to refer to the order in which tree nodes are visited. *tsearch* uses preorder, postorder and endorder to respectively refer to visting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

CAVEAT

If the calling function alters the pointer to the root, results are unpredictable.

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char *ttyname (*fildes*)
int *fildes;*

int isatty (*fildes*)
int *fildes;*

DESCRIPTION

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

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

FILES

/dev/*

DIAGNOSTICS

ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.

CAVEAT

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

NAME

ttyslot – find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

DESCRIPTION

ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/inittab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

FILES

/etc/inittab
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

bessel: j0, j1, jn, y0, y1, yn – Bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

DESCRIPTION

j0 and *j1* return Bessel functions of *x* of the first kind of orders 0 and 1 respectively. *jn* returns the Bessel function of *x* of the first kind of order *n*.

y0 and *y1* return Bessel functions of *x* of the second kind of orders 0 and 1 respectively. *yn* returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

SEE ALSO

matherr(3M)

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return the value **–HUGE** and to set *errno* to **EDOM**. In addition, a message indicating DOMAIN error is printed on the standard error output.

Arguments too large in magnitude cause *j0*, *j1*, *y0* and *y1* to return zero and to set *errno* to **ERANGE**. In addition, a message indicating TLOSS error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

DESCRIPTION

erf returns the error function of x, defined as $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

erfc, which returns 1.0 – *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large x and the result subtracted from 1.0 (e.g., for $x = 5$, 12 places are lost).

SEE ALSO

exp(3M)

NAME

exp, log, log10, pow, sqrt – exponential, logarithm, power, square root functions

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 e^x.

log returns the natural logarithm of x. The value of x must be positive.

log10 returns the logarithm base ten of x. The value of x must be positive.

pow returns x^y. If x is zero, y must be positive. If x is negative, y must be an integer.

sqrt returns the non-negative square root of x. The value of x may not be negative.

SEE ALSO

hypot(3M), matherr(3M), sinh(3M)

DIAGNOSTICS

exp returns **HUGE** when the correct value would overflow, or 0 when the correct value would underflow, and sets *errno* to **ERANGE**.

log and *log10* return **–HUGE** and set *errno* to **EDOM** when x is non-positive. A message indicating DOMAIN error (or SING error when x is 0) is printed on the standard error output.

pow returns 0 and sets *errno* to **EDOM** when x is 0 and y is non-positive, or when x is negative and y is not an integer. In these cases, a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow or underflow, *pow* returns $\pm$**HUGE** or 0 respectively, and sets *errno* to **ERANGE.**

sqrt returns 0 and sets *errno* to **EDOM** when x is negative. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

NAME

floor, ceil, fmod, fabs – floor, ceiling, remainder, absolute value 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 (as a double-precision number) not greater than x.

ceil returns the smallest integer not less than x.

fmod returns the floating-point remainder of the division of x by y: x if y is zero or if x/y would overflow; otherwise 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, $|x|$.

SEE ALSO

abs(3C)

NAME

gamma – log gamma function

SYNOPSIS

```
#include <math.h>

double gamma (x)
double x;

extern int signgam;
```

DESCRIPTION

gamma returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t} t^{x-1} dt$. The sign of $\Gamma(x)$ is returned in the external integer *signgam*. The argument x may not be a non-positive integer.

The following C program fragment might be used to calculate Γ:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
        error( );
y = signgam * exp(y);
```

where LN_MAXDOUBLE is the least value that causes *exp*(3M) to return a range error, and is defined in the **<values.h>** header file.

SEE ALSO

exp(3M), matherr(3M), values(5)

DIAGNOSTICS

For non-negative integer arguments **HUGE** is returned, and *errno* is set to **EDOM**. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, *gamma* returns **HUGE** and sets *errno* to **ERANGE**.

These error-handling procedures may be changed with the function *matherr*(3M).

NAME

hypot – Euclidean distance function

SYNOPSIS

```
#include <math.h>

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

DESCRIPTION

hypot returns:

sqrt(x * x + y * y),

taking precautions against unwarranted overflows.

SEE ALSO

matherr(3M)

DIAGNOSTICS

When the correct value would overflow, *hypot* returns **HUGE** and sets *errno* to **ERANGE.**

These error-handling procedures may be changed with the function *matherr*(3M).

NAME

matherr – error-handling function

SYNOPSIS

#include <math.h>

int matherr (*x*)
struct exception **x*;

DESCRIPTION

matherr is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors, by including a function named *matherr* in their programs. *matherr* must be of the form described above. When an error occurs, a pointer to the exception structure *x* will be passed to the user-supplied *matherr* function. This structure, which is defined in the **<math.h>** header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	argument domain error
SING	argument singularity
OVERFLOW	overflow range error
UNDERFLOW	underflow range error
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element *name* points to a string containing the name of the function that incurred the error. The variables *arg1* and *arg2* are the arguments with which the function was invoked. *retval* is set to the default value that will be returned by the function unless the user's *matherr* sets it to a different value.

If the user's *matherr* function returns non-zero, no error message will be printed, and *errno* will not be set.

If *matherr* is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to EDOM or ERANGE and the program continues.

EXAMPLE

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
   switch (x->type) {
   case DOMAIN:
      /* change sqrt to return sqrt(-arg1), not 0 */
      if (!strcmp(x->name, "sqrt")) {
         x->retval = sqrt(-x->arg1);
         return (0); /* print message and set errno */
      }
   case SING:
      /* all other domain or sing errors, print message and abort */
      fprintf(stderr, "domain error in %s\n", x->name);
      abort( );
   case PLOSS:
      /* print detailed error message */
      fprintf(stderr, "loss of significance in %s(%g) = %g\n",
         x->name, x->arg1, x->retval);
      return (1); /* take no other action */
}
   return (0); /* all other errors, execute default procedure */
}
```

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors					
type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
BESSEL:	–	–	–	–	M, 0	*
y0, y1, yn (arg < 0)	M, –H	–	–	–	–	–
EXP:	–	–	H	0	–	–
LOG, LOG10:						
(arg < 0)	M, –H	–	–	–	–	–
(arg = 0)	–	M, –H	–	–	–	–
POW:	–	–	±H	0	–	–
neg ** non-int	M, 0	–	–	–	–	–
0 ** non-pos						
SQRT:	M, 0	–	–	–	–	–
GAMMA:	–	M, H	H	–	–	–
HYPOT:	–	–	H	–	–	–
SINH:	–	–	±H	–	–	–
COSH:	–	–	H	–	–	–
SIN, COS, TAN: –	–	–	–	M, 0	*	
ASIN, ACOS, ATAN2: M, 0	–	–	–	–	–	

ABBREVIATIONS

*	As much as possible of the value is returned.
M	Message is printed (EDOM error).
H	HUGE is returned.
–H	–HUGE is returned.
±H	HUGE or –HUGE is returned.
0	0 is returned.

NAME

sinh, cosh, tanh – hyperbolic functions

SYNOPSIS

#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;

DESCRIPTION

sinh, *cosh*, and *tanh* return, respectively, the hyberbolic sine, cosine and tangent of their argument.

SEE ALSO

matherr(3M)

DIAGNOSTICS

sinh and *cosh* return **HUGE** (and *sinh* may return **–HUGE** for negative *x*) when the correct value would overflow and set *errno* to **ERANGE**.

These error-handling procedures may be changed with the function *matherr*(3M).

NAME

trig: 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 (y, x)
double y, x;
```

DESCRIPTION

sin, *cos* and *tan* return respectively the sine, cosine and tangent of their argument, *x*, measured in radians.

asin returns the arcsine of *x*, in the range [–π/2,π/2].

acos returns the arccosine of *x*, in the range [0,π].

atan returns the arctangent of *x*, in the range [–π/2,π/2].

atan2 returns the arctangent of *y*/*x*, in the range (–π,π], using the signs of both arguments to determine the quadrant of the return value.

SEE ALSO

matherr(3M)

DIAGNOSTICS

sin, *cos*, and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return zero when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments causing partial loss of significance, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to **ERANGE.**

If the magnitude of the argument of *asin* or *acos* is greater than one, or if both arguments of *atan2* are zero, zero is returned and *errno* is set to **EDOM**. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

MOTOROLA INC.

Microcomputer Division
2900 South Diablo Way
Tempe, Arizona 85282
P.O. Box 2953
Phoenix, Arizona 85062

11045 PRINTED IN USA (3/90) WPC 2,500