Burroughs

# XE 500 CENTIX™

Operations Reference Manual

Volume 3: System Operations, Part 1

Relative To Release Level 6.0
Priced Item
November 1986

Distribution Code SA
Printed in U S America
1192192

Burroughs

# XE 500 CENTIX™

Operations Reference Manual

## Volume 3: System Operations, Part 1

Relative To Release Level 6.0
Priced Item
November 1986

Distribution Code SA
Printed in U S America
1192192

# About This Manual

## Purpose

The purpose of the *XE 500 CENTIX Operations Reference Manual* is to provide a comprehensive reference for the XE 500 CENTIX operating system.

## Scope

This manual describes the commands, system calls, libraries, data files, and device interfaces that make up the CENTIX Operating System running on the XE 500 computer.

## Audience

Volumes 1 and 2 of this manual are intended for all users of the CENTIX operating system. CENTIX system programmers are the primary audience for Volumes 3 and 4.

## Prerequisites

General users of the CENTIX system should be familiar with the particular environments in which they will be working. A section called **Getting Started,** preceding the Shell Command descriptions in Volumes 1 and 2, provides a generic CENTIX tutorial.

Programmers should have an understanding of the CENTIX operating system structure and should be experienced at writing programs in the C programming language.

## How to Use This Manual

Use this manual as a starting point to find the documentation for a CENTIX feature with which you are unfamiliar. To find the entry you need, refer to the following:

- Permuted Index. This indexes each significant word in each entry's description. A complete Permuted Index for the whole manual is in each volume.
- Contents Listing. Included in the Contents Listing is an alphabetical list of entries, under the appropriate sections, together with the entry descriptions. Each volume contains the Contents Listing.
- Related Shell Command Entries. This section, for Volumes 1 and 2 only, groups together related shell command entries that are in Section 1.

## Organization

This manual consists of six sections:

Section 1, Shell Commands, describes programs that are intended to be invoked directly by the user through the CENTIX System shell.

Section 2, System Calls, describes the entries into the CENTIX kernel, including the C language interfaces.

Section 3, Library Functions, describes the available library functions and subroutines.

Section 4, Special File Formats, documents the structure of particular kinds of files.

Section 5, Miscellaneous Facilities, includes descriptions of macro packages, character set tables, and so on.

Section 6, Device Files, describes various device files that refer to specific hardware peripherals and CENTIX System device drivers.

## Related Product Information

*XE 500 CENTIX Administration Guide*

*XE 500 CENTIX centrEASE Operations Reference Manual*

*XE 500 CENTIX C Language Programming Reference Manual*

*XE 500 CENTIX Programming Guide*

*XE 500 CENTIX Operations Guide*

# Contents

## Volume 1: Shell Operations, Part 1

# Tables

# System Calls

## intro

### Name

intro - introduction to system calls and error numbers

### Format

```
#include <errno.h>
```

### Description

This section describes all of the XE 500 CENTIX system calls. System calls are functions that call the CENTIX kernel. They are used to perform a variety of system-dependent tasks, such as accessing files, opening pipes, and controlling your CENTIX environment. System calls are handled in a way similar to C language library functions (see Section 3), although they are accessible from other languages, as well.

Most system calls have one or more error returns. An error condition is indicated by an otherwise impossible return value, which is almost always -1; the individual descriptions specify the details. An error number is also made available in the external variable **errno** (see **perror** in Section 3). **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 the super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

## intro

**2** ENOENT No such file or directory

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

**3** ESRCH No such process

No process can be found corresponding to that specified by *pid* in a **kill**.

**4** EINTR Interrupted system call

An asynchronous signal (such as interrupt or quit), which the user has chosen to catch, has occurred during a system call. If execution is resumed after the signal is processed, 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 that 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. On local terminals, it may indicate that the host terminal lacks the specified channel.

**7** E2BIG Arg list too long

An argument list longer than 10K bytes is presented to a member of the **exec** family, or an item (such as a file) would become too large.

**8** ENOEXEC Exec format error

A request is made to execute a file that, although it has the appropriate permissions, does not start with a valid magic number (see **a.out** in Section 4).

## intro

**9** EBADF Bad file number

Either a file descriptor refers to an unopen file, or a read (or write) request is made to a file that is only open for writing (or reading).

**10** ECHILD No child processes

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

**11** EAGAIN No more processes

A **fork** failed because the system's process table is full or the user is not allowed to create any more processes.

**12** ENOMEM Not enough space

During an **exec**, **brk**, or **sbrk**, a program asks for more space than the system is able to supply. The maximum allocation is 3.5 megabytes; a program that gets this condition with a smaller allocation may work at another time when other large programs are not hogging the swap file. If this problem recurs, the system administrator may want to consider enlarging the swap file.

The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a **fork**.

**13** EACCES Permission denied

An attempt was made to access a file in a way forbidden by the protection system. From **locking**, an attempt was made to do a checking lock on bytes already under a lock.

**14** EFAULT Bad address

The system encountered a bad pointer 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 (such as in **mount**).

## intro

**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 appropriate context, such as **link**.

**18** EXDEV Cross-device link

A link to a file on another device was attempted.

**19** ENODEV No such device

An attempt was made to apply an inappropriate system call to a device (for example, 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**).

**21** EISDIR Is a directory

An attempt was made to write on a directory.

**22** EINVAL Invalid argument

Some invalid argument (such as dismounting a non–mounted device, mentioning an undefined signal in **kill**, reading or writing a file for which **lseek** has generated a negative pointer). Also set by the math functions described in the math library functions in Section 3 of this manual.

**23** EINFILE File table overflow

The system table file is full, and temporarily no more **opens** can be accepted.

## intro

**24** EMFILE Too many open files

No process may have more than 20 file descriptors open at a time.

**25** ENOTTY Not a character device

An attempt was made to perform an **ioctl** call to 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 a pure-procedure program that is being executed.

**27** EFBIG File too large

The size of a file exceeded the maximum file size (1,082,201,088 bytes).

**28** ENOSPC No space left on device

During a write to an ordinary file, there is no free space left on the device. This can occur in a PILF file when the file system lacks unallocated clusters as big as the file's cluster size. On tape files, it indicates a read past the end of the tape.

**29** ESPIPE Illegal seek

An **lseek** 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 (1000) 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.

## intro

**33** EDOM Math argument

The argument of a function in the math package (see **intro** to Section 3) is out of the domain of the function.

**34** ERANGE Result too large

The value of a function in the math package (see **intro** to Section 3) 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.

**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.

**37** ECHRNG Channel number out of range.

**38** EL2NSYNC Level 2 not synchronized.

**39** EL3HLT Level 3 halted.

**40** EL3RST Level 3 reset.

**41** ELNRNG Link number out of range.

**42** EUNATCH Protocol driver not attached.

**43** ENOCSI No CSI structure available.

**44** EL2HLT level 2 halt.

**50** EBADE Invalid exchange

Use of an invalid Inter-CPU Communication exchange descriptor.

**51** EBADR Invalid request descriptor

Use of an invalid Inter-CPU Communication request descriptor.

# intro

**52** EXFULL Exchange full

An Inter-CPU Communication request failed because an exchange is full. The exchange might be the request's response exchange or the service exchange.

**53** ENOANO No anode

The Application Processor has as many files open as it can handle.

**54** EBADRQC Invalid request code

No CENTIX or BTOS process is servicing the specified request code.

**56** EDEADLOCK Deadlock error

Call cannot be honored because of potential deadlock or because lock table is full. See **locking**.

## Definitions

The following definitions describe terms that are used frequently throughout the system call and library function documentation.

### Directory

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

### 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, 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**.

# intro

## 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 super-user.

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 (070) 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 (07) of the file mode is set.

## 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 19. A process may have no more than 20 file descriptors (0-19) open simultaneously. A file descriptor is returned by system calls such as **open** or **pipe**. The file descriptor is used as an argument by calls such as **read**, **write**, **ioctl**, and **close**.

# intro

### File Name

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 file names because of the special meaning attached to these characters by the shell (see **sh** in Section 1). Although permitted, avoid the use of unprintable characters in file names.

### Message Queue Identifier

A message queue identifier (*msqid*) is a unique positive integer created by an **msgget** 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 msqid_ds {
  struct   ipc_perm msg_perm;   /*operation permiss. struct*/

  ushort   msg_qnum;            /*number of msgs on q*/
  ushort   msg_qbytes;          /*max number of bytes on q*/

  ushort   msg_lspid;           /*pid of last msgsnd oper.*/

  ushort   msg_lrpid;           /*pid of last msgrcv oper.*/

  time_t   msg_stime;           /*last msgsnd time*/
  time_t   msg_rtime;           /*last msgrcv time*/
  time_t   msg_ctime;           /*last change time*/
                                /*Times measured in seconds*/

                                /*since 00:00:00 GMT, 1/1/70*/

}
```

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

```
      struct msg_perm {
        ushort  cuid;      /*creator user ID*/
        ushort  cgid;      /*creator group ID*/
        ushort  uid;       /*user ID*/
        ushort  gid;       /*group ID*/
        ushort  mode;      /*r/w permission*/
      }
```

## intro

*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 an msgsnd operation. *Msg_lrpid* is the process ID of the last process that performed an **msgrcv** operation. *Msg_stime* is the time of the last **msgsnd** operation, *msg_rtime* is the time of the last **msgrcv** operation, and *msg_ctime* is the time of the last **msgctl** operation that changed a member of the above structure.

### Message Queue Operation Permissions

In the **msgop** and **msgctl** 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. |
| 00060 | Read, Write by group. |
| 00006 | Read, Write by others. |

Read and Write permissions on a message queue identifier are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user

The effective user ID of the process matches *msg_perm.[c]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 user ID of the process does not match *msg_perm.[c]uid* and the effective group ID of the process matches *msg_perm.[c]gid* and the appropriate bit of the "group" portion (060) of msg_perm.mode is set.

# intro

The effective user ID of the process does not match *msg_perm.[c]uid* and the effective group ID of the process does not match *msg_perm.[c]gid* and the appropriate bit of the "other" portion (06) of msg_perm.mode is set.

Otherwise, the corresponding permissions are denied.

### Parent Process ID

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

### Path Name and Path Prefix

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

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

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

where <file-name> is a string of 1 to 14 characters other than the ASCII slash and null, and <dirname> is a string of 1 to 14 characters (other than the ASCII slash and null) that names a directory.

If a path name begins with a slash, the path search begins at the root directory. A slash by itself names the root directory.

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

# intro

### Process Group ID

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

### 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.

### Real User ID and Real Group ID

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

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

### Root Directory and Current Working Directory

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

# intro

## Semaphore Identifier

A semaphore identifier (*semid*) is a unique positive integer created by a **semget** 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 semid_ds {
   struct  ipc_perm sem_perm;    /*operation permiss. struct*/

   ushort  sem_nsems;            /*number of sems in set*/
   time_t  sem_otime;            /*last operation time*/
   time_t  sem_ctime;            /*last change time*/
                                 /*Times measured in seconds*/

                                 /*since 00:00:00 GMT, 1/1/70*/

}
```

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

```
struct sem_perm {
   ushort  cuid;     /*creator user ID*/
   ushort  cgid;     /*creator group ID*/
   ushort  uid;      /*user ID*/
   ushort  gid;      /*group ID*/
   ushort  mode;     /*r/a permission*/
}
```

The value of *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* is the time of the last **semop** operation, and *sem_ctime* is the time of the last **semctl** operation that changed a member of the above structure.

## intro

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

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

*Semval* is a non-negative integer. *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** and **semctl** 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. |
| 00060 | Read, Alter by group. |
| 00006 | Read, 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 super-user.

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

## intro

The effective user ID of the process does not match *sem_perm.[c]uid* and the effective group ID of the process matches *sem_perm.[c]gid* and the appropriate bit of the "group" portion (060) of sem_perm.mode is set.

The effective user ID of the process does not match *sem_perm.[c]uid* and the effective group ID of the process does not match *sem_perm.[c]gid* and the appropriate bit of the "other" portion (06) 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** system call. Each *shmid* has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as *shmid*_ds and contains the following members:

```
struct shmid_ds {
  struct   ipc_perm shm_perm;    /*operation permiss. struct*/
  int      shm_segsz;            /*size of segment*/
  ushort   shm_cpid;             /*creator pid*/
  ushort   shm_lpid;             /*pid of last operation*/
  short    shm_nattch;           /*number of current attaches*/
  time_t   shm_atime;            /*last attach time*/
  time_t   shm_dtime;            /*last detach time*/
  time_t   shm_ctime;            /*last change time*/
                                 /*Times measured in seconds*/
                                 /*since 00:00:00 GMT, 1/1/70*/
}
```

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

```
struct shm_perm {
  ushort   cuid;        /*creator user ID*/
  ushort   cgid;        /*creator group ID*/
  ushort   uid;         /*user ID*/
  ushort   gid;         /*group ID*/
  ushort   mode;        /*r/w permission*/
}
```

# intro

*Shm_segsz* specifies the size of the shared memory segment. *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** operation. *Shm_nattch* is the number of processes that currently have this segment attached. *Shm_atime* is the time of the last **shmat** operation, *shm_dtime* is the time of the last **shmdt** operation, and *shm_ctime* is the time of the last **shmctl** operation that changed one of the members of the above structure.

## Shared Memory Operation Permissions

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

| | |
|---|---|
| 00400 | Read by user. |
| 00200 | Write by user. |
| 00060 | Read, Write by group. |
| 00006 | Read, 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 super-user.

The effective user ID of the process matches *shm_perm.[c]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 user ID of the process does not match *shm_perm.[c]uid* and the effective group ID of the process matches *shm_perm[c]gid* and the appropriate bit of the "group" portion (060) of shm_perm.mode is set.

## intro

The effective user ID of the process does not match *shm_perm.[c]uid* and the effective group ID of the process does not match *shm_perm.[c]gid* and the appropriate bit of the "other" portion (06) of shm_perm.mode is set.

Otherwise, the corresponding permissions are denied.

#### 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.

#### Super-user

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

#### tty Group ID

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

### See Also

**apnum, devnm** in Section 1; **close, ioctl, open, pipe, read, write; intro** in Section 3.

# access

## Name

**access** - determines the accessibility of a file

## Format

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

## Description

*Path* points to a path name naming a file. **access** checks the named file for accessibility according to the bit pattern contained in *amode*. It uses 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 is true:

| | |
|---|---|
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [ENOENT] | Read, write, or execute (search) permission is requested for a null path name. |
| [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. |

## access

| | |
|---|---|
| [ETXTBSY] | Write **access** is requested for a pure procedure (shared text) file that is being executed. |
| [EACCESS] | Permission bits of the file mode do not permit the requested **access**. |
| [EFAULT] | Path points outside the allocated address space of the process. |

The owner of a file has access permission checked with respect to the owner read, write, and execute mode bits. Other members of the file group have permission checked with respect to the group mode bits. All others have permission checked with respect to the other mode bits.

### Returns

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

### See Also

**chmod, stat.**

# acct

## Name

**acct** - enable or disable process accounting

## Format

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

## Description

The **acct** function 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. Terminations can be caused by one of two things: an **exit** call or a **signal**; see **exit** and **signal**. The effective user ID of the calling process must be super-user to use this call.

*Path* points to a path name naming the accounting file. The accounting file format is given in **acct** in Section 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.

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

| | |
|---|---|
| [EPERM] | The effective user of the calling process is not super-user. |
| [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 path name do not exist. |

## acct

| | |
|---|---|
| [EACCES] | A component of the path prefix denies search permission. |
| [EACCES] | The file named by *path* is not an ordinary file. |
| [EACCES] | *Mode* permission is denied for the named accounting file. |
| [EISDIR] | The named file is a directory. |
| [EROFS] | The named file resides on a read-only file system. |
| [EFAULT] | *Path* points to an illegal address. |

### Returns

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

**exit, signal; acct** in Section 4.

# alarm

## Name

**alarm** - set a process alarm clock

## Format

```
unsigned alarm (sec)
unsigned sec;
```

## Description

The **alarm** system call 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.**

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.

## Returns

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

## See Also

**pause, signal; sleep** in Section 1.

# brk

## Name

**brk, sbrk** - change data segment space allocation

## Format

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

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

## Description

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

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

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

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

- Such a change would result in more space being allocated than is allowed by a system-imposed maximum (see **ulimit**). Note that due to a lack of swap space this may be less than what **ulimit** reports. [ENOMEM]
- Such a change would result in the break value being greater than or equal to the start address of any attached shared memory segment (see **shmop**).

# brk

## Returns

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

## See Also

**exec.**

# chdir

## Name

**chdir** - changes the current working directory.

## Format

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

## Description

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

**chdir** fails and the current working directory is not changed 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. |

## Returns

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

## See Also

**chroot**.

# chmod

## Name

**chmod** - change mode of file

## Format

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

## Description

*Path* points to a path name naming a file. The **chmod** system call sets the access permission portion of the named file mode according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

| | |
|---|---|
| 04000 | Set user ID on execution. |
| 02000 | Set group ID on execution. |
| 01000 | Save text image after execution. |
| 00400 | Read by owner. |
| 00200 | Write by owner. |
| 00100 | Execute (or search if a directory) by owner. |
| 00070 | Read, write, execute (search) by group. |
| 00007 | Read, write, execute (search) by others. |

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

If the effective user of the process is not the super-user, mode bit 01000 (save text image on execution) is cleared.

If the effective user of the process is not super-user 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.

## chmod

If an executable file is prepared for sharing, then mode bit 01000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus, when the next user of the file executes it, the text does not have to be read from the file system. It can simply be swapped in, thus saving time.

**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 super-user. |
| [EROFS] | The named file resides on a read-only file system. |
| [EFAULT] | *Path* points outside the allocated address space of the process. |

### Returns

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

### See Also

**chown, mknod.**

# chown

## Name

**chown** - changes the owner and/or group of a file.

## Format

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

## Description

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

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

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

**chown** fails and the owner and group of the named file remain 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 either the owner of the file or the superuser. |
| [EROFS] | The named file resides on a read-only file system. |
| [EFAULT] | *Path* points outside the allocated address space of the process. |

## chown

### Returns

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

### See Also

**chown** in Section 1; **chmod**.

# chroot

## Name

**chroot** - change the root directory.

## Format

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

## Description

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

The effective user of the process must be the super-user 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** fails and the root directory remains 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 that of the super-user. |
| [EFAULT] | *Path* points outside the allocated address space of the process. |

## chroot

### Returns

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

**chdir**.

# close

## Name

**close** - close a file descriptor.

## Format

```
int close (fildes)
int fildes;
```

## Description

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

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

## Returns

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

## See Also

**creat**, **dup**, **exec**, **fcntl**, **open**, **pipe**.

# creat

## Name

**creat** - create a new file or rewrite an existing one.

## Format

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

## Description

The **creat** system call creates a new ordinary file or prepares to rewrite an existing file named by the path name indicated by *path*.

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

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

The process's cluster size exponent determines the cluster size of files created on PILF file systems. See **syslocal**.

Upon successful completion, the file descriptor, which is a non–negative integer, is returned and the file is opened 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**. No process may have more than 20 files open simultaneously. A new file may be created with a mode that forbids writing.

## creat

**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] | Twenty file descriptors are currently open. |
| [EFAULT] | *Path* points outside the allocated address space of the process. |
| [ENFILE] | The system file table is full. |

## Returns

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

## See Also

**chmod**, **close**, **dup**, **fcntl**, **locking**, **lseek**, **open**, **read**, **umask**, **write**.

# dup

## Name

**dup** - duplicate an open file descriptor.

## Format

```
int dup (fildes)
int fildes;
```

## Description

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

- Same open file (or pipe).
- Same file pointer (that is, 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** systems calls. See **fcntl**.

The file descriptor returned is the lowest one available.

**dup** fails if one or more of the following are true:

| | |
|---|---|
| [EBADF] | *Fildes* is not a valid open file descriptor. |
| [EMFILE] | 20 file descriptors are currently open. |

## Returns

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

## See Also

**creat, close, exec, fcntl, open, pipe.**

# exAllocExch

## Name

**exAllocExch** - allocate exchange

## Format

```
#include <exch.h>

unsigned char exAllocExch ();
```

## Description

See **Exchanges**.

# exCall

## Name

**exCall** - send a request and wait for the response

## Format

```
#include <exch.h>

exCall (reqbl);
struct reqheader *reqbl;
```

## Description

The **exCall** system call sends a request and waits for the response. *Reqbl* must point to a request block that describes the message. The request block has four parts: a request header, control information, request PbCbs, and response PbCbs.

The ICC user include file defines a request header in the following way:

```
struct rqheader {
            unsigned short r_sCntInfo;
            unsigned char r_nReqPbCb;
            unsigned char r_nRespPbCb;
            unsigned short r_userNum;
            unsigned short r_exchResp;
            unsigned short r_ercRet;
            unsigned short r_rqCode;
};
```

The client sets the following fields: *r_sCntInfo* (which must be even), *r_nReqPbCb*, and *r_nRespPbCb* specify the size of the rest of the request block; *r_exchResp* specifies where the response must be sent; and *r_rqCode* specifies the destination of the request. The kernel and server ignore any values in *r_userNum* or *r_ercRet*. Each request code requires specific values for *r_sCntInfo*, *r_nReqPbCb*, and *r_nRespPbCb*.

# exCall

The client uses the control information to send fixed-length data fields to the server.

A PbCb has the following structure:

```
struct PbCb {
            char *pc_offset;
            unsigned short pc_count;
};
```

The client uses Request PbCbs to send blocks of data to the server. Each PbCb gives the location (*pc_offset*) and size (*pc_count*) of a data block.

The client uses Response PbCbs to pass response data areas (*pc_offset*) and maximum lengths (*pc_count*) to the server and kernel. If the server ignores the restrictions, the kernel right-truncates the offending fields.

The memory containing the variable-length fields need not immediately follow the request block.

## Returns

-1 indicates an error, with an error code in **errno**. See **perror** in Section 3.

## Cautions

If the service is provided by BTOS, integer data must have Intel byte ordering. See **shortswap** in Section 3.

The **lint** shell command may complain that **exCall** argument types are inconsistent, especially if the client uses more than one kind of request block. To suppress these complaints, cast the argument to its official type:

```
exCall((struct rqheader*) reqbl);
```

Use of this cast does not affect the object code.

## exCall

If an **exCall** is being used by a program that also traps signals, users must beware when a -1 is returned with **errno** equal to EINTR. In this case, the **exCall** has effectively done an **exRequest**. The user must then do an **exWait** and an **exCpResponse** (and correctly handle the interrupted **exWait** by restarting it). Otherwise, your program will lose one of its responses and will consume valuable kernel heap space to hold the response until your program exits.

### See Also

**exchanges** in the *XE 500 CENTIX Programming Guide*

# exchanges

## Name

**exAllocExch**, **exDeallocExch** - obtain and abandon exchanges.

## Format

```
#include <exch.h>

unsigned char exAllocExch ();

exDeallocExch (ex)
unsigned char ex;
```

## Description

A process must own exchanges in order to receive messages. Each exchange has an exchange descriptor, which is unique to the owner of the exchange.

**exAllocExch** allocates a new exchange and returns its exchange descriptor. The calling process can use this exchange to receive both requests and responses.

**exDeallocExch** deallocates the specified exchange. Any requests still waiting or on their way to the exchange are rejected with a return code of 0xFF. Any responses still waiting or on their way to the exchange are discarded.

A process's death deallocates all its exchanges, but an **exec** has no effect on exchanges.

## Returns

-1 indicates error, with an error code in **errno**. See **perror** in Section 3.

# exCheck

## Name

**exCheck** - examine an ICC message queue

## Format

```
#include <exch.h>

exCheck (ex, mstat);
unsigned char ex;
struct msgret *mstat;
```

## Description

See **exWait**.

# exCnxSendOnDealloc

## Name

**exCnxSendOnDealloc** - make final requests

## Format

```
#include <exch.h>

exCnxSendOnDealloc (req)
unsigned short req;
```

## Description

See **exfinal.**

# exCpRequest

## Name

**exCpRequest**, **exReject** - remove a request from an exchange.

## Format

```
#include <exch.h>

exCpRequest (reqdes, reqst)
unsigned short reqdes;
struct rqheader *reqst;

exReject (reqdes, r_ercRet)
unsigned short reqdes;
unsigned short r_ercRet;
```

## Description

The **exCpRequest** and **exReject** system calls both remove a request from a server's exchange. A server that wants to examine the request uses **exCpRequest**; a server that has no interest in the message's contents uses **exReject**.

**exCpRequest** copies the message indicated by the request descriptor, *reqdes*. The kernel places the request block and request data blocks together at the location pointed to by *reqst*. *Reqst* must be an even address; each data block appears at an even address. (The amount of memory the message requires is returned by a check on the message queue; see **exWait**.) The kernel sets the request PbCbs to point to the server's copies of the data blocks.

**exReject** discards the contents of the indicated message. It sends the response, with the return code (*m_ercRet* in the request block header) set to *r_ercRet*.

## exCpRequest

### Returns

-1 indicates error, with an error code in **errno**. See **perror** in Section 3.

### Files

/usr/include/exch.h - ICC user include file

# exCpResponse

## Name

**exCpResponse, exDiscard** - remove a response from an exchange.

## Format

```
#include <exch.h>

exCpResponse (reqdes, reqst)
unsigned short reqdes;
struct rqheader *reqst;

exDiscard (reqdes)
unsigned short reqdes;
```

## Description

The **exCpResponse** and **exDiscard** system calls both remove a response from an exchange. A client that wants to examine the response uses **exCpResponse**; a client that has no interest in the message's contents uses **exDiscard**.

**exCpResponse** copies the message indicated by the request descriptor *reqdes*. The kernel uses the request block pointed to by *reqst* to place the parts of the response:

- The error code goes in the *r_ercRet* field of the request block header.
- The kernel examines each response PbCb in the request block. The *pc_offset* field should be set to the location reserved for the data; *pc_count* should be set to the number of bytes available at that location. If the server provided more than *pc_count* bytes, the kernel right-truncates the data to fit. The kernel overwrites *pc_count* with the number of bytes actually transferred.

## exCpResponse

**exDiscard** discards the contents of the indicated message. It returns the message's return code field (*m_ercRet* in the request block header).

### Returns

-1 indicates error, with an error code in **errno**. See **perror** in Section 3.

### Caution

If the service is provided by BTOS, integer data has Intel byte ordering. See **shortswap** in Section 3.

### Files

/usr/include/exch.h - ICC user include file.

# exDeallocExch

## Name

**exDeallocExch** - deallocate exchange

## Format

```
#include <exch.h>

exDeallocExch (ex)
unsigned char ex;
```

## Description

See **Exchanges**.

# exDiscard

## Name

**exDiscard** - remove a response from an exchange

## Format

```
#include <exch.h>

exDiscard (reqdes)
unsigned short reqdes;
```

## Description

See **exCpResponse**.

# exec

## Name

**execl, execv, execle, execve, execlp, execvp** - execute files

## Format

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

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

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

int execve (path, argv, envp)
char *file, *arg0, *arg1, ..., *argn;

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

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

## Description

The **exec** system call 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** in Section 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.

## exec

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

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

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least 1, and the first member of the array points to a string containing the name of the file.

*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** in Section 5). The environment is supplied by the shell (see **sh** in Section 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 is used to pass the environment of the calling process to the new process.

## exec

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

Signals set to terminate the calling process are set to terminate the new process. Signals set to be ignored by the calling process are set to be ignored by the new process. Signals set to be caught by the calling process are set to terminate the new process. See **signal**.

If the set-user-ID mode bit of the new process file is set (see **chmod**), **exec** sets the effective user ID of the new process equal 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.

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

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

nice value (see **nice**)
process ID
parent process ID
process group ID
ICC exchanges, with unremoved messages addressed to them
semadj values (see **semop**)
tty group ID (see **exit** and **signal**)
trace flag (see **ptrace** request 0)
time left until an alarm clock signal (see **alarm**)
current working directory
root directory
file mode creation mask (see **umask**)
file size limit (see **ulimit**)
**utime**, **stime**, **cutime**, and **cstime** (see **times**)
PILF cluster size exponent for this process

## exec

An **exec** fails and returns to the calling process if one or more of the following are true:

| | |
|---|---|
| [ENOENT] | One or more components of the new process file path name do not exist. |
| [ENOTDIR] | A component of the new process file path prefix is not a directory. |
| [EACCES] | Search permission is denied for a directory listed in the new process file 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 argument list is greater than the system-imposed limit of 10,240 bytes. |
| [EFAULT] | The new process file is not as long as indicated by the size values in its header. |
| [EFAULT] | *path, argv,* or *envp* point to an illegal address. |

## Returns

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

## See Also

**sh** in Section 1; **alarm, exit, fork, nice, ptrace, semop, signal, times, ulimit, umask**; **a.out** in Section 4; **environ** in Section 5.

# execl

## Name

**execl** - execute files

## Format

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

## Description

See **exec**.

# execle

## Name

**execle** - execute a file

## Format

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

## Description

See **exec**.

# execlp

## Name

**execlp** - execute a file

## Format

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

## Description

See **exec**.

# execv

## Name

**execv** - execute a file

## Format

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

## Description

See **exec**.

# execve

## Name

**execve** - execute a file

## Format

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

## Description

See **exec**.

# execvp

## Name

**execvp** - execute a file

## Format

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

## Description

See **exec**.

# exfinal

## Name

**exSendOnDealloc**, **exCnxSendOnDealloc** - make final requests.

## Format

```
#include <exch.h>

unsigned short exSendOnDealloc (reqblk)
struct reqheader *reqblk;

exCnxSencOnDealloc (req)
unsigned short req;
```

## Description

The **exSendOnDealloc** system call specifies a request and returns a request descriptor in precisely the same manner as **exRequest**. But where **exRequest** dispatches the request immediately, **exSendOnDealloc** puts a hold on the request. When the client process deallocates the request's response exchange (either by dying or by a call to **exDealloc**; see **exchanges**), the kernel delivers the message.

The **exCnxSendOnDealloc** system call cancels the specified message. *Req* must be a value returned by a call to **exSendOnDealloc**.

## Returns

-1 indicates error, with an error code in **errno**. See **perror** in Section 3.

# exfinal

## Cautions

The server must respond to the message, even though there is no one to read the response.

## Files

/usr/include/exch.h - ICC user include file

# exit

## Name

**exit, _exit** - terminate process

## Format

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

## Description

The **exit** system call terminates the calling process with the following consequences:

- All of the file descriptors opened in the calling process are closed.
- If the parent process of the calling process is executing a **wait**, it is notified of the calling process termination and the low order eight bits (that is, bits 0377) of status are made available to it; see **wait**.
- If the parent process of the calling process is not executing a **wait**, the calling process is transformed into a zombie process. A zombie process is a process that occupies a slot only 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 is set to 1 for all of the child processes and zombie processes created by the calling process. This means the initialization process inherits each of these processes.
- All ICC exchanges are deallocated. This is the only way to deallocate the default response exchange.
- If the process ID, tty group ID, and process group ID of the calling process are equal, the SIGHUP signal is sent to each process that has a process group ID equal to that of the calling process.

## exit

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

### See Also

**acct**, **intro**, **exchanges**, **semop**, **signal**, **wait**.

# exReject

## Name

exReject - remove a request from an exchange

## Format

```
#include <exch.h>

exReject (reqdes, r_ercRet)
unsigned short reqdes;
unsigned short r_ercRet;
```

## Description

See **exCpRequest**.

# exRequest

## Name

**exRequest** - send a message to a server

## Format

```
#include <exch.h>

unsigned short exRequest (reqbl);
struct reqheader *reqbl;
```

## Description

The **exRequest** system call sends a message to a server. *reqbl* points to a request block that describes the message. **exRequest** returns a request descriptor; this descriptor appears in subsequent references to the request by the client or the kernel.

The request block has four parts: a request header, control information, request PbCbs, and response PbCbs.

A request header has the following structure.

```
struct rqheader {
        unsigned short r_sCntInfo;
        unsigned char r_nReqPbCb;
        unsigned char r_nRespPbCb;
        unsigned short r_userNum;
        unsigned short r_exchResp;
        unsigned short r_ercRet;
        unsigned short r_rqCode;
};
```

## exRequest

The client sets the following fields: *r_sCntlnfo* (which must be even), *r_nReqPbCb*, and *r_nRespPbCb* specify the size of the rest of the request block; *r_exchResp* specifies where the response must be sent; and *r_rqCode* specifies the destination of the request. The kernel and server ignore any values in *r_userNum* or *r_ercRet*. Each request code requires specific values for *r_sCntlnfo*, *r_nReqPbCb*, and *r_nRespPbCb*.

The client uses the control information to send fixed-length data fields to the server.

A PbCb has the following structure:

```
struct PbCb {
        char *pc_offset;
        unsigned short pc_count;
};
```

The client uses Request PbCbs to send request data blocks to the server. Each PbCb gives the location (*pc_offset*) and size (*pc_count*) of a data block.

The client uses Response PbCbs to pass response data-length restrictions to the server. The client sets the *pc_count* field of each response PbCb to the maximum length for that data block.

The locations containing the client request data need not immediately follow the request block.

The kernel copies the complete message immediately. Once **exRequest** returns, it is safe to modify the message.

After the client has sent the request, it must watch for the corresponding response **(exWait)** and specify the response's disposition **(exCpResponse)**

# exRequest

## Returns

-1 indicates error, with an error code in **errno**. See **perror** in Section 3.

## Cautions

Use of **exRequest** requires more client-kernel interaction than is necessary for most requests. Compare **exCall**.

If the service is provided by BTOS, integer data must have Intel byte ordering. See **shortswap** in Section 3.

The **lint** compiler may complain the **exRequest** argument types are inconsistent, especially if the client uses more than one kind of request block. To suppress these complaints, cast the argument to its official type:

```
    exRequest((struct rqheader *) reqbl);
```

Use of this cast does not affect the object code.

# exRespond

## Name

**exRespond** - send a message to a client

## Format

```
#include <exch.h>

exRespond (reqdes, reqbl)
unsigned short reqdes;
struct reheader *reqbl;
```

## Description

The **exRespond** system call issues a response to a specific request. The request descriptor *reqdes* specifies that request. *reqbl* points to a request block that dsecribes the response. This request block has the same format as the request block that described the request (see **exRequest**). The server sets only the error return code fields and each of the response PbCbs.

The kernel copies the complete message immediately. Once **exRespond** returns, it is safe to modify the message.

The memory containing the server's variable-length response fields need not directly follow the request block.

## Returns

-1 indicates error, with an error code in **errno**. See **perror** in Section 3.

# exSendOnDealloc

## Name

**exSendOnDealloc** - make final requests

## Format

```
#include <exch.h>

unsigned short exSendOnDealloc (reqblk)
struct rqheader *reqblk;
```

## Description

See **exfinal**

# exServRq

## Name

**exServeRq** - appropriate a request code

## Format

```
#include <exch.h>

exServeRq (exch, code);
unsigned char exch;
unsigned short code;
```

## Description

A server (a process that receives requests) must own a request code for use by clients (processes that send requests). **exServeRq** appropriates *code* as a request code and assigns the request to the exchange specified by *exch*. If *exch* is zero, the process gives up *code*, which can then be appropriated by another server.

Any process can appropriate a request code, but only one can own it at a time.

Codes 0 through 0xBFFF (49151) are reserved for Burroughs system services. Each installation should reserve additional codes for local system services. User services must not use reserved codes, even if they do not currently identify a service.

## Returns

-1 indicates error, with an error code in **errno**. See **perror** in Section 3.

# exWait

## Name

**exWait, exCheck** - examine an ICC message queue.

## Format

```
#include <exch.h>

exWait (ex, mstat);
unsigned char ex;
struct msgret *mstat;

exCheck (ex, mstat);
unsigned char ex;
struct msgret *mstat;
```

## Description

Each call to **exWait** or **exCheck** returns with information on the oldest unnoticed message waiting at the exchange whose descriptor is *ex*. An unnoticed message is one that **exWait** and **exCheck** have not reported on since the last time a message was removed from the exchange. When an exchange's owner removes a message, all messages still waiting become "unnoticed" again; see **exCpResponse** and **exCpRequest**. **exCall** never affects the "noticed" status of any message.

**exWait** and **exCheck** write a report to the memory pointed to by *mstat*. The report has the following structure:

```
struct msgret {
     unsigned short m_rqCode;
     unsigned short m_reqdes;
     int m_size;
     char m_flag;
     unsigned short m_ercRet;
     unsigned char m_cputype;
     unsigned char m_slot;
     struct request *m_offset;
};
```

## exWait

When the process takes further action on this message (copying it from the message queue; if it's a request, sending a response) it passes the kernel *m_reqdes* to identify the specific message.

**exWait** and **exCheck** differ only in their "no messages" action. If no unnoticed messages wait at the specified exchange, **exWait** waits for a new one to arrive; **exCheck** returns immediately with an error code.

The calling process must specify some action on each message. See **exCpResponse** and **exCpRequest.**

### Returns

If **exWait** or **exCheck** terminate unsuccessfully, a value of -1 is returned and **errno** is set to indicate the error.

# fcntl

## Name

**fcntl** - file control

## Format

```
#include <fcntl.h>

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

## Description

The **fcntl** system call 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 as follows:<br><br>Lowest numbered available file descriptor greater than or equal to *arg*.<br><br>Same open file (or pipe) as the original file.<br><br>Same file pointer as the original file (that is, both file descriptors share one file pointer).<br><br>Same access mode (read, write or read/write).<br><br>Same file status flags (that is, both file descriptors share the same file status flags)<br><br>The close-on-exec flag associated with the new file descriptor is set to remain open across **exec** 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). |

# fcntl

| | |
|---|---|
| F_GETFL | Get file status flags. |
| F_SETFL | Set file status flags to *arg*. Only certain flags can be set. See **fcntl** in Section 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, then the structure is passed back unchanged except for the lock type (which will be 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** in Section 5). The command **F_SETLK** is used to establish read (**F_RDLCK**) and write (**F_WRLCK**) locks, as well as to remove either type of lock (**F_UNLCK**). If a read or write lock cannot be set, **fcntl** will return immediately with an error value of -1. |
| F_SETLKW | This is the same as **F_SETLK**, except that if a read or write lock is blocked by other locks, the process will sleep 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*), and process id (*l_pid*) of the segment of the file to be affected. The process id field is used only with the **F_GETLK** command to return the value 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.

# fcntl

If such a lock also has *l_start* set to zero, the whole file will be 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 affect. 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** system call.

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

| | |
|---|---|
| [EBADF] | *Fildes* is not a valid open file descriptor. |
| [EMFILE] | *Cmd* is F_DUPFD and 20 file descriptors are currently open. |
| [EMFILE] | *Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock and there are no more file-locking headers available (too many files have segments locked).D |
| [EINVAL] | *Cmd* is F_DUPFD and *arg* is negative, greater than or equal to 20. |
| [EINVAL] | *Cmd* is F_GETLK, F_SETLK, or F_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) or write (F_WRLCK) lock, and the segment of a file to be locked is already write-locked by another process, or the type is a write lock and the segment of a file to be locked is already read- or write-locked by another process. |
| [ENOSPC] | *Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock and there are no more file-locking headers available (too many files have segments locked), or there are no more record locks available (too many file segments locked). |
| [EDEADLK] | *Cmd* is F_SETLK, when the lock is blocked by some lock from another process and sleeping (waiting) for that lock to become free; this causes a deadlock situation. |

# fcntl

## Returns

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

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

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

## See Also

**close, exec, open; fcntl** in Section 5.

# fork

## Name

**fork** - create a new process.

## Format

```
int fork ()
```

## Description

The **fork** system call 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**)
signal handling settings (such as SIG_DFL, function address)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
nice value (see **nice**)
process group ID
tty group ID (see **exit**)
current working directory
root directory
file mode creation mask (see **umask**)
file size limit
PILF cluster size exponent (see **pilf** in Section 5)

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 (the process ID of the parent process).
- The child process has its own copies of its parent process file descriptors. Each child process file descriptor shares a common file pointer with the corresponding file descriptor of the parent.

# fork

- The child process inherits no ICC exchanges from the parent. Initially, the child's only exchange is the default response exchange.

The **fork** call fails and no process is 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. |
| [EXFULL] | A default response exchange cannot be allocated for the process. |

## Returns

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

## See Also

**exchanges, exec, nice, plock, ptrace, semop, shmop, signal, times, ulimit, umask, wait.**

# fstat

## Name

**fstat** - get file status

## Format

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

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

## Description

See **stat**.

# getegid

## Name

**getegid** - get effective group ID

## Format

```
unsigned short getegid ()
```

## Description

See **getuid**.

# geteuid

## Name

**geteuid** - get effective user ID

## Format

```
unsigned short geteuid ()
```

## Description

See **getuid**.

# getgid

## Name

**getgid** - get real group ID

## Format

```
unsigned short getgid ()
```

## Description

See **getuid**.

## getpgrp

### Name

**getpgrp** - get process group ID

### Format

```
int getpgrp ()
```

### Description

See **getpid**.

# getpid

## Name

**getpid**, **getpgrp**, **getppid** - get process, process group, and parent process IDs

## Format

```
int getpid ()

int getpgrp ()

int getppid ()
```

## Description

The **getpid** system call 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**, **fork**, **intro**, **setpgrp**, **signal**.

# getppid

## Name

**getppid** - get parent process ID

## Format

```
int getppid ()
```

## Description

See **getpid**.

# getuid

## Name

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

## Format

```
unsigned short getuid ()

unsigned short geteuid ()

unsigned short getgid ()

unsigned short getegid ()
```

## Description

The **getuid** call returns the real user ID of the calling process.

The **geteuid** call 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, setuid.**

# ioctl

## Name

**ioctl** - control device

## Format

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

## Description

The **ioctl** system call performs a variety of functions on character special files (devices). The write-ups of various devices in Section 6 discuss how **ioctl** applies to them.

**ioctl** will fail 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 character special device. |
| [EINVAL] | *Request* or *arg* is not valid. See Section 6. |
| [EINTR] | A signal was caught during the **ioctl** system call. |

## Returns

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

## See Also

**termio** in Section 6.

# kill

## Name

**kill** - send a signal to a process or a group of processes

## Format

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

## Description

The **kill** system call 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*, which can be 0 or one from the list given in **signal**.

If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

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

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

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

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

If *pid* is -1 and the effective user ID of the sender is not that of the super-user, *sig* is sent to all processes (excluding *proc0* and *proc1*) whose real user IDs are equal to the effective user ID of the sender.

## kill

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

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

The **kill** call fails and no signal is 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 super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process. |

### Returns

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** in Section 1; **getpid, setpgrp, signal.**

# link

## Name

**link** - link to a file

## Format

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

## Description

*Path1* points to a path name of an existing file. *Path2* points to a path name of the new directory entry to be created. The **link** system call creates a new link (directory entry) for the existing file.

The **link** call fails and no link is 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. |

# link

## Returns

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

## See Also

**unlink**.

# locking

## Name

**locking** - exclusive access to regions of a file

## Format

```
int locking (filedes, mode, size);
int filedes, mode;
long size;
```

## Description

The **locking** system call places or removes an advisory lock on a region of a file.

Parameters specify the file to be locked or unlocked, the kind of lock or unlock, and the region affected:

- *Filedes* specifies the file to be locked or unlocked. It is a file descriptor returned by an **open, creat, pipe, fcntl,** or **dup** system call
- *Mode* specifies the action: 0 for lock removal, 1 for blocking lock, 2 for checking lock. Blocking and checking locks differ only if the attempted lock is itself locked out: a blocking lock waits until the existing lock or locks are removed; a checking lock immediately returns an error.
- The region affected begins at the current file offset associated with *filedes* and is *size* bytes long. If *size* is zero, the region affected ends at the end of the file (*size* should be positive).

Locking imposes no structure on a CENTIX file. A process can arbitrarily lock any unlocked byte and unlock any locked byte. However, creating a large number of noncontiguous locked regions can fill up the system's lock table and make further locks impossible. It is advisable that a program's use of the **locking** call segment the file in the same way as does the program's use of a **read** and **write**.

## locking

A process is said to be deadlocked if it is sleeping until an unlocking which is indirectly prevented by that same sleeping process. The kernel will not permit a blocking **locking** if such a call would deadlock the calling process. **Errno** is set to EDEADLOCK. The standard response to such a situation is for the program to release all its existing lock areas and try again. If a **locking** call fails because the kernel's table of locked areas is full, again, **errno** is set to EDEADLOCK and, again, the calling program should release its existing locked areas.

Special files and pipes can be locked, but no input/output is blocked.

Locks are automatically removed if the process that placed the lock terminates or closes the file descriptor used to place the lock.

### Returns

A return value of -1 indicates an error, with the error value in **errno**.

| | |
|---|---|
| [EACCES] | A checking lock on a region already locked. |
| [EDEADLOCK] | A lock that would cause deadlock or overflow the system's lock table. |

### Caution

Do not apply any standard input/output library function to a locked file: this library does not know about **locking**.

The lock is purely advisery. Users who wish to can still **read**, **write**, **creat**, and **open** the file. The **locking** system call is the only system call that checks locks.

### See Also

**creat**, **close**, **dup**, **open**, **read**, **write**.

# lseek

## Name

**lseek** - move read/write file pointer

## Format

```
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. The **lseek** system call sets the file pointer associated with *fildes* as follows:

- If *whence* is 0, the pointer is set to *offset* bytes.
- If *whence* is 1, the pointer is set to its current location plus *offset*.
- If *whence* is 2, the pointer is set to the size of the file plus *offset*.

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

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

| | |
|---|---|
| [EBADF] | *Fildes* is not an open file descriptor. |
| [ESPIPE] | *Fildes* is associated with a pipe or fifo. |
| [EINVAL and SIGSYS signal] | *Whence* is not 0, 1 or 2. |
| [EINVAL] | The resulting file pointer would be negative. |

## Returns

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.

## Caution

Not all block devices support **lseek**.

## See Also

**creat**, **dup**, **fcntl**, **open**.

# mknod

## Name

**mknod** - makes a directory, or a special or ordinary file

## Format

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

## Description

The **mknod** system call creates a new file named by the path name pointed to by *path*. The mode of the new file is initialized from *mode*. 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 000000 ordinary file
0004000 set user ID on execution
0002000 set group ID on execution
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 is set to the process' effective user ID. The group ID is set to the process' effective group ID.

Values of *mode* other than those presented are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process' file mode creation mask: all bits set in the process file mode creation mask are cleared. See **umask**. 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

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

The **mknod** command fails and the new file is not created if one or more of the following are true:

| | |
|---|---|
| [EPERM] | The process's effective user ID is not super-user. |
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [ENOENT] | A component of the path prefix does not exist. |
| [EROFS] | The 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. |

## Returns

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** in Section 1; **chmod, exec, umask**; **fs** in Section 4.

# mount

## Name

**mount** - mount a file system

## Format

```
int mount (spec, dir, rwflg)
char *spec, *dir;
int rwflg;
```

## Description

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

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

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

**mount** may be invoked only by the super-user. Note that the system call **mount** (as well as **umount**) does not update the mount table file /etc/mnttab. This means that the system calls can not be used interchangeably with the **mount** and **umount** shell commands.

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

| | |
|---|---|
| [EPERM] | The effective user ID is not super-user. |
| [ENOENT] | Any of the named files do not exist. |
| [ENOTDIR] | A component of a path prefix is not a directory. |
| [ENOTBLK] | *Spec* is not a block special device. |
| [ENXIO] | The device associated with *spec* does not exist. |
| [ENOTDIR] | *Dir* is not a directory. |

# mount

| | |
|---|---|
| [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] | The low-order bit of *rwflag* is zero and the volume containing the file system is physically write-protected. |

## Returns

The **mount** command returns an integer. Upon successful completion, a value of 0 is returned and references to the file *dir* refer to the root directory on the mounted file system. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

## See Also

**umount**.

# msgctl

## Name

**msgctl** - message control operations

## Format

```
#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

The **msgctl** system call provides a variety of message control operations as specified by *cmd*. The following *cmds* 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**. {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 super user or to the value of msg_perm.uid in the data structure associated with *msqid*. Only super user can raise the value of msg_qbytes.

# msgctl

| | |
|---|---|
| **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.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.** |
| [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 and it is not equal to the value of *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. |

## Returns

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, msgget, msgop.**

# msgget

## Name

**msgget** - get message queue

## Format

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

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

## Description

The **msgget** system call returns the message queue identifier associated with *key*.

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

10 *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*, and *msg_rtime* are set to 0.

*Msg_ctim* is set to the current time.

*Msg_qbytes* is set equal to the system limit.

# msgget

**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**) 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." |

## Returns

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.

## See Also

**intro, msgctl, msgop.**

# msgop

## Name

**msgop** - message operations

## Format

```
#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

The **msgsnd** call is used to send a message to the queue associated with the message queue identifier specified by *msqid.* {WRITE} *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.

## msgop

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

These actions are as follows:

> If (*msgflg* & **IPC_NOWAIT**) is "true," the message will not be sent and the calling process will return immediately.
>
> If (*msgflg* & **IPC_NOWAIT**) is "false," the calling process will suspend 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**). 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**.

**msgsnd** will fail and no message will be 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**). |
| [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. |

## msgop

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

*Msg_qnum* is incremented by 1.

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

*Msg_stime* is set equal to the current time.

The **msgrcv** call 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*. {READ} 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 will return immediately with a return value of -1 and **errno** set to ENOMSG.

## msgop

If (*msgflg* & **IPC_NOWAIT**) is "false," the calling process will suspend 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**.

**msgrcv** will fail and no message will be 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 mesage 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**).

*Msg_qnum* is decremented by 1.

*Msg_lrpid* is set to the current time.

*Msg_rtime* is set to the current time.

# msgop

## Returns

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.

## See Also

**intro, msgctl, msgget, signal.**

# nice

## Name

**nice** - change priority of a process

## Format

```
int nice (incr)
int incr;
```

## Description

The **nice** system call adds the value of *incr* to the nice value of the calling process. A process' nice value is a positive number. A higher nice value results in a lower CPU priority.

The system allows nice values only from -8 to 39. The **nice** system call grants nice values from -8 to -1 only to super-user processes. These negative nice values cause the CPU priority of the process to be fixed independently of CPU usage of the process. nice values from 0 to 39 allow the system to adjust dynamically the actual CPU priority of the process, temporarily lowering it in proportion to the process' recent level of CPU usage. If a super-user process requests a nice value below -8, or if any other process requests a nice value below 0, the system imposes a nice value of 0. If any process requests a nice value above 39, the system imposes a nice value of 39.

| | |
|---|---|
| [EPERM] | The **nice** call fails and does not change the nice value if *incr* is negative and the effective user ID of the calling process is not super-user. |

## Returns

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. To receive the current **nice** value, use 0 as *incr*.

## See Also

**nice** in Section 1; **exec**.

# open

## Name

**open** - open a file for reading or writing

## Format

```
#include <fcntl.h>

int open (path, oflag[, mode])
char *path;
int oflag, mode;
```

## Description

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

| | |
|---|---|
| O_RDONLY | Open for reading only. |
| O_WRONLY | Open for writing only. |
| O_RDWR | Open for reading and writing. |
| O_NDELAY | This flag may affect subsequent reads and writes. See **read** and **write**. |

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

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

If O_NDELAY is clear:

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

## open

When opening a file associated with a communication line:

If O_NDELAY is set:

The **open** will return without waiting for a carrier.

If O_NDELAY is clear:

The **open** will block until carrier is present. If you want to configure an RS-232 modem entry, you must declare the async line in the CP or TP configuration file (for example, CP00.cnf) as a modem. To do this, enter:

```
Async 1, speed=9600, modem
```

When an OpenTerminal request is received, the code in the TP or CP raises data terminal ready (DTR) and asserts request to send (RTS). It then waits for data set ready (DSR) to be asserted. BTOS will wait up to 3 seconds for DSR. If DSR is not received in that time, the request is rejected with an erc 11010 (ercDSRNotDetected). Assuming that DSR is recognized, the processor will then begin to wait for data carrier detect (DCD). There is no timeout waiting for DCD. The OpenTerminal will return only when DCD has been asserted.

After a successful OpenTerminal, ReadTerminal requests will return the ercCarrierLoss error if the carrier has dropped since the last request. The fCarrierDetect flag in the terminal output structure will follow the current value of the DCD RS-232 signal.

CloseTerminal drops RTS and DTR and pauses 0.25 second to allow DSR time to drop on mechanically switched modems.

| | |
|---|---|
| O_APPEND | If set, the file pointer is set to the end of the file prior to each write. |
| O_NODIRECT | Do not perform direct I/O for this file, even if a transfer satisfies the system default criteria. |

## open

| | |
|---|---|
| O_SYNC | If set, all writes will be synchronous. This option applies only to regular files. |
| O_CREAT | If the file exists, this flag has no effect. Otherwise, the file owner ID is set to the process's effective user ID, the file group ID is set to the process's effective group ID, and the low-order 10 bits of the file mode are set to the value of mode, modified as follows (see **creat**):<br><br>All bits set in the process file mode creation mask are cleared. See **umask.**<br><br>The "save text image after execution" bit of the mode is cleared. See **chmod**.<br><br>The process's default cluster size exponent determines the cluster size of files created on PILF file systems. |
| 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 and the file exists, **open** fails. O_EXCL has no meaning unless it is used with O_CREAT. |

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**.

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

| | |
|---|---|
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [ENOENT] | O_CREAT is not set and the named file does not exist. |
| [EACCES] | A component of the path prefix denies search permission. |
| [EACCES] | *Oflag* permission is denied for the named file. |
| [EISDIR] | The named file is a directory and *oflag* is write or read/write. |
| [EROFS] | The named file resides on a read-only file system and *oflag* is write or read/write. |

# open

| | |
|---|---|
| [EMFILE] | Twenty file descriptors are currently open. |
| [ENXIO] | The named file is a character special or block special file, and the device associated with this special file does not exist. |
| [ETXTBSY] | The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write. |
| [EFAULT] | *Path* points outside the allocated address space of the process. |
| [EEXIST] | O_CREAT and O_EXCL are set, and the named file exists. |
| [ENXIO] | O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading. |
| [EINTR] | A signal was caught during the **open** system call. |
| [ENFILE] | The system file table is full. |

## Returns

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

## See Also

**chmod, close, creat, dup, fcntl, locking, lseek, pipe, read, umask, write; pilf** in Section 5.

## pause

### Name

**pause** - suspend process until signal

### Format

```
pause ()
```

### Description

The **pause** system call 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**), 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, kill, signal, wait.**

# pipe

## Name

**pipe** - create an interprocess channel

## Format

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

## Description

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

The call will fail if one or both are true:

| | |
|---|---|
| [EMFILE] | 19 or more file descriptors are currently open. |
| [ENFILE] | The system file table is full. |

## Returns

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

## See Also

**sh** in Section 1; **read, write.**

# plock

## Name

**plock** - lock process, text, or data in memory

## Format

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

int plock (op)
int op;
```

## Description

The **plock** system call allows the calling process to lock its text segment (text lock), its data and stack segments (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. For 407 object modules, TXTLOCK and DATLOCK are identical. 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. |

Shared regions (that is, text) may be locked by anyone using the text, but they may be unlocked only if the caller is the last one using the region. Note that sticky-bit text that is not explicitly unlocked will remain locked in core even after the last process using it terminates.

## plock

The **plock** call fails and does 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. |

## Returns

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.

## See Also

**exec**, **exit**, **fork**.

# profil

## Name

**profil** - execution time profile

## Format

```
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 tic, (60th second); *offset* is subtracted from it, and the result is multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left; 0177777 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words 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.

## Returns

Not defined.

## See Also

**prof** in Section 1; **monitor** in Section 3.

## ptrace

### Name

**ptrace** - process trace

### Format

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

### Description

The **ptrace** system call 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** in Section 1. The child process behaves normally until it encounters a signal (see **signal** for the list), at which time it enters a stopped state and its parent is notified by **wait**. 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**. The *pic*, *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.

## ptrace

**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 (as on PDP-11s), 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 (as on Burroughs 68000-family processors, the 3B 20S computer, and VAX-11/780), either request **1** or request **2** may be used with equal results. The *data* argument is ignored. These two requests will fail if *addr* is not the start address of a word, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

**3** With this request, the word at location *addr* in the child's USER area in the system's address space (see <sys/user.h>) is returned to the parent process. Addresses in this area range from 0 to 8192 on Burroughs 68000-family processors, 0 to 1024 on the PDP-11s and 0 to 2048 on the 3B 20 computer and VAX. The *data* argument is ignored. This request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of -1 is returned to the parent process and the parent's **errno** is set to EIO.

**4, 5** With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated (as on PDP-11s), request 4 writes a word into I space, and request 5 writes a word in to D space. If I and D space are not separated (as on Burroughs 68000-family processors, the 3B 20 computer, and VAX), either request 4 or request 5 may be used with equal results. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests will fail if *addr* is a location in a pure procedure space and another process is executing in that space, or *addr* is not the start address of a word. Upon failure a value of -1 is returned to the parent process and the parent's **errno** is set to EIO.

**6** With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are:

## ptrace

The general registers (that is, registers 0-15 on Burroughs 68000-family processors, registers 0-11 on the 3B 20S computer, registers 0-7 on PDP-11s, and registers 0-15 on the VAX).

The condition codes of the Processor Status Word on the 3B 20 computer.

The floating point status register and six floating point registers on PDP-11s.

certain bits of the Processor Status Word on PDP-11s (that is, bits 0-4, and 8-11).

Certain bits of the Processor Status Longword on the VAX (that is, bits 0-7, 16-20, and 30-31).

Burroughs 68000-family processors: all processor status bits except 8, 9, 10, and 13.

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**.

9 This request sets the trace bit in the Processor Status Word of the child (i.e., bit 15 on Burroughs 68000-family processors, bit 4 on PDP-11s; bit 30 on the VAX) 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. On the 3B 20S computer there is no trace bit and this request returns an error. Note that the trace bit remains set after an interrupt on PDP-11s but is turned off after an interrupt on the VAX.

## ptrace

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

**ptrace** will in fail if one or more of the following are true:

| | |
|---|---|
| [EIO] | *Request* is an illegal number. |
| [ESRCH] | *Pid* identifies a child that does not exist or has not executed a **ptrace** with request 0. |

## See Also

**exec, signal, wait.**

## read

### Name

**read** - read from a file

### Format

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

### Description

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

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

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

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

Upon successful completion, **read** returns the number of bytes actually read (a non-negative integer) and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see **ioctl**; see **termio** in Section 6), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

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

- If O_NDELAY is set, the read returns a 0.
- If O_NDELAY is clear, the read blocks until data is written to the file or the file is no longer open for writing.

## read

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

- If O_NDELAY is set, the read returns a 0.
- If O_NDELAY is clear, the read blocks until data becomes available.

The **read** call fails 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. |
| [EDEADLOCK] | A side effect of a previous **locking** call. |

## Returns

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

## Caution

Large data reads that are 4K (4096 bytes), or exact multiples of 4K, will pad the remaining portion of the read buffer with zeros when the data transferred does not completely fill the buffer. A data read that is not a multiple of 4K, however, will not pad the read buffer with zeros.

## See Also

**create, dup, fcntl, ioctl, locking, open, pipe**; **termio** in Section 6.

# sbrk

## Name

**sbrk** - change data segment space allocation

## Format

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

## Description

See **brk**.

# semctl

## Name

**semctl** - semaphore control operations

## Format

```
#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;

 } arg;
```

## Description

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

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

| | |
|---|---|
| **GETVAL** | Return the value of **semval** (see **intro**). {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} |

## semctl

The following *cmds* 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 *cmds* 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**. {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 super-user or to the value of 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 super-user or to the value of sem_perm.uid in the data structure associated with *semid.*

**semctl** will fail 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. |

## semctl

| | |
|---|---|
| [EACCES] | Operation permission is denied to the calling process (see **intro**). |
| [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 super-user and it is not equal to the value of *sem_perm.uid* in the data structure associated with *semid.* |
| [EFAULT] | *Arg.buf* points to an illegal address. |

## Returns

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

| | |
|---|---|
| **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.

## See Also

**intro, semget, semop**.

# semget

## Name

**semget** - get set of semaphores

## Format

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

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

## Description

The **semget** system call returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores (see **intro**) are created for *key* if one of the following are 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* is set equal to the current time.

## semget

**semget** will fail 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**) 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." |

## Returns

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.

## See Also

**intro, semctl, semop.**

# semop

## Name

**semop** - semaphore operations

## Format

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

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

## Description

The **semop** system call is used to automically 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 include 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*.

## semop

*Sem_op* specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following will occur: {ALTER}

If semval (see **intro**) 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**) for the specified semaphore. All processes suspended waiting for *semval* are rescheduled.

If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & **IPC-NOWAIT**) is "true," *semop* will return immediately.

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

*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, and all the operations are tried again.

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

## semop

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**.

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. {ALTER}

If *sem_op* is zero, one of the following will occur: {READ}

If *semval* is zero, *semop* will return immediately.

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

If *semval* is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false," *semop* will increment 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**.

## semop

**semop** will fail if one or more of the following are true for any of the seamphore 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**). |
| [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 a **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.

## Returns

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

## semop

### See Also

**exec, exit, fork, intro, semctl, semget.**

# setgid

## Name

**setgid** - set group ID

## Format

```
int setgid (gid)
int gid;
```

## Description

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

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

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

If the effective user ID of the calling process is not super-user, but the saved set-group ID from **exec** is equal to *gid*, the effective group ID is set to *gid*.

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

## Returns

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

## See Also

**getuid, intro, setuid.**

# setpgrp

## Name

**setpgrp** - set process group ID

## Format

```
int setpgrp ()
```

## Description

**Setpgrp** sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

## Returns

**Setpgrp** returns the value of the new process group ID.

## See Also

**exec, fork, getpid, intro, kill, signal.**

# setuid

## Name

**setuid** - set user ID

## Format

```
int setuid (uid)
int uid;
```

## Description

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

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

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

If the effective user ID of the calling process is not super-user, but the saved set-user ID from **exec** is equal to *uid*, the effective user ID is set to *uid*.

**setuid** will fail if the real user ID of the calling process is not equal to *uid* and its effective user ID is not super-user [EPERM], or if the *uid* is out of range [EINVAL].

## Returns

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

## See Also

**getuid**, **intro**.

# shmctl

## Name

**shmctl** - shared memory control operations

## Format

```
#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

The **shmctl** system call provides a variety of shared memory control operations as specified by *cmd*. The following *cmds* 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**. {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 either that of super-user or to the value of *shm_perm.uid* in the data structure associated with *shmid*.

# shmctl

**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 either that of super-user or to the value of *shm_perm.uid* in the data structure associated with *shmid.*

**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**). |
| [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 super-user and it is not equal to the value of *shm_perm.uid* in the data structure associated with *shmid.* |
| [EFAULT] | *Buf* points to an illegal address. |

## Returns

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, shmget, shmop.**

# shmget

## Name

**shmget** - get shared memory segment

## Format

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

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

## Description

The **shmget** system call returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of size *size* bytes (see **intro**) 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 follows:

*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 to 0.

*Shm_ctime* is set equal to the current time.

## shmget

**shmget** will fail 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**) 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. |
| [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 physical 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." |

### Returns

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.

### See Also

**intro, shmctl, shmop.**

# shmop

## Name

**shmop** - shared memory operations

## Format

```
#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*.

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

# shmop

| | |
|---|---|
| [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. With the initial resease, this value is 64K. This value can be changed by customizing the CENTIX kernel. |
| [EMFILE] | The number of shared memory segments attached to the calling process would exceed the system-imposed limit. |
| [EINVAL] | **shmdt** detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*. |
| [EINVAL] | **shmdt** will fail and not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment. |

## Returns

Upon successful completion, the return value is as follows:

**shmat** returns the data segment start address of the attached shared memory segment.

**shmdt** returns a value of 0.

Otherwise, -1 is returned and **errno** is set to indicate the error.

## Caution

Processes that share a segment of memory on an Application Processor must be executing on that AP. Memory cannot be shared across Application Processors.

## See Also

**exec, exit, fork, intro, shmctl, shmget.**

# signal

## Name

**signal** - specify what to do upon receipt of a signal

## Format

```
#include <signal.h>

int (*signal (sign, func))()
int sig;
void (*func)();
```

## Description

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

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

| | | |
|---|---|---|
| SIGHUP | 01 | Hangup. |
| SIGINT | 02 | Interrupt. |
| SIGQUIT | 03 | Quit. |
| SIGILL | 04 | Illegal instruction (not reset when caught). |
| SIGTRAP | 05 | Trace trap (not reset when caught). |
| SIGIOT | 06 | IOT instruction. |
| SIGEMT | 07 | EMT instruction. |
| SIGFPE | 08 | Floating point exception. |
| SIGKILL | 09 | Kill (cannot be caught or ignored). |
| SIGBUS | 10 | Bus error. |
| SIGSEGV | 11 | Segmentation violation. |
| SIGSYS | 12 | Bad argument to system call. |
| SIGPIPE | 13 | Write on a pipe with no one to read it. |
| SIGALRM | 14 | Alarm clock. |
| SIGTERM | 15 | Software termination signal. |
| SIGUSR1 | 16 | User-defined signal 1. |
| SIGUSR2 | 17 | User-defined signal 2. |
| SIGCLD | 18 | Death of a child (reset when caught). |
| SIGPWR | 19 | Power fail (not reset when caught). |

See below for the significance of the asterisk (*) in the above list.

## signal

*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**

Terminate process upon receipt of a signal.

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in **exit**. In addition a "core image" will be 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 read 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.

**SIG_IGN**

Ignore signal.

The signal *sig* is to be ignored. Note that the signal **SIGKILL** cannot be ignored.

*function address*

Catch signal.

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL**. **SIGTRAP**, or **SIGPWR**.

# signal

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

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

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

**signal** will fail if sig is an illegal signal number, including **SIGKILL**. [EINVAL]

## Returns

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

## Caution

Two other signals that behave differently than the signals described above exist in this release of the system. They are:

| | | |
|---|---|---|
| **SIGCLD** | 18 | Death of a child (reset when caught). |
| **SIGPWR** | 19 | Power fail (not reset when caught). |

There is no guarantee that, in future release of the CENTIX system, these signals will continue to behave as described below; they are included only for compatibility with some versions of the UNIX system. Their use in new programs is strongly discouraged.

## signal

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.<br>The signal is to be ignored. |
| **SIG_IGN** | Ignore signal.<br>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**. |
| *function address* | Catch signal.<br>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** except that while the process is executing the signal-catching function, any received **SIGCLD** signals will be queued and the signal-catching function will be continually reentered until the queue is empty. |

The **SIGCLD** affects two other system calls (**wait**, and **exit**) in the following ways:

| | |
|---|---|
| **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. |

## signal

### Known Problems

A user process cannot catch a signal caused by an invalid memory reference during a partially completed instruction. Thus **SIGSEGV** can be ignored or be allowed to terminate the process, but cannot be caught. This bug is due to a temporary implementation problem.

### See Also

**kill** in Section 1; **kill**, **pause**, **ptrace**, **wait**, **setjmp** in Section 3.

# stat

## Name

**stat, fstat** - get file status

## Format

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

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

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

## Description

*Path* points to a path name naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. **stat** obtains information about the named file. **stat** works with all files, but does not obtain information peculiar to PILF files (see **syslocal**; see **pilf** in Section 5).

Similarly, **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 about the file is placed.

The contents of the structure pointed to by *buf* include the following members:

```
ushort    st_mode;       /*File mode; see mknod*/
ino_t     st_ino;        /*I-node number*/
dev_t     st_dev;        /*ID of device containing*/
                         /*a directory entry for this file*/
```

## stat

```
dev_t     st_rdev;      /*ID of device*/
                        /*This entry is defined only for*/
                        /*character special or block*/
                        /*special files*/
short     st_nlink;     /*Number of links*/
ushort    st_uid;       /*User ID of the file's owner*/
ushort    st_gid;       /*Group ID of the file's group*/
off_t     st_size;      /*File size in bytes*/
time_t    st_atime;     /*Time of last access*/
time_t    st_mtime;     /*Time of last data modification*/
time_t    st_ctime:     /*Time of last file status change*/
                        /*Times measured in seconds since*/
                        /*00:00:00 GMT, Jan. 1, 1970*/
```

| | |
|---|---|
| *st_atime* | Time when file data was last accessed. Changed by the following system calls: **creat**, **mknod**, **pipe**, and **read**. |
| *st_mtime* | Time when data was last modified. Changed by the following system calls: **creat**, **mknod**, **pipe**, and **write**. |
| *st_ctime* | Time when file status was first changed. Changed by the following system calls: **chmod**, **chown**, **creat**, **link**, **mknod**, **pipe**, **unlink**, and **write**. |

Note that when recreating a file that already exists and the existing file is more than zero bytes in length, only the modification time (*st_mtime*) and the file status time (*st_ctime*) are updated. The file data access time is not updated since this field in the buffer changes only when data from the file is actually accessed. If you recreate an existing file that is zero bytes in length, the modification time, file status time, and file data access time will not be updated.

The **stat** call fails 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. |

# stat

The **fstat** call fails if one or more of the following are true:

| | |
|---|---|
| [EBADR] | *Fildes* is not a valid open file descriptor. |
| [EFAULT] | *Buf* points to an invalid address. |

## Returns

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**, **chown**, **creat**, **link**, **mknod**, **pipe**, **read**, **syslocal**, **time**, **unlink**, **utime**, **write**.

## stime

### Name

**stime** - set time

### Format

```
int stime (tp)
long *tp;
```

### Description

The **stime** system call 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.

**stime** fails if:

| | |
|---|---|
| [EPERM] | The effective user ID of the calling process is not super-user. |

### Returns

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

**time**.

## swrite

### Name

**swrite** - synchronous write on a file

### Format

```
int swrite (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

### Description

The **swrite** system call has the same purpose and conventions as **write**. The two differ solely in the handling of disk input/output. **swrite**, unlike **write**, does not give a normal return before physical output is complete. A program that executes an **swrite** can assume that the data is on the disk, not waiting in a buffer pool.

### See Also

**creat**, **dup**, **lseek**, **open**, **pipe**, **write**.

## sync

### Name

**sync** - update super-block

### Format

```
void sync ()
```

### Description

The **sync** system call 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** and **df**. It is mandatory for a boot.

The writing, although scheduled, is not necessarily complete upon return from **sync**.

# syslocal

## Name

**syslocal** - special system requests

## Format

```
#include <syslocal.h>

int syslocal (cmd[, arg] ...)
int cmd;
```

## Description

The **syslocal** system call executes certain special system calls. The specific call is indicated by the first argument.

### System Type

```
int syslocal(SYSL_SYSTEM);
```

Return SYSL_XE for the XE 500.

### Superblock Synchronization

```
int syslocal(SYSL_RESYNC, devnum)
short devnum
```

Preserve current contents of the superblock. *Devnum* specifies the file system: the high order byte contains the major device number of the character special device; the low order byte contains the minor device number. The superblock is reread, replacing the current in-RAM copy of the superblock. Both actions have the effect of preventing the system from writing out the superblock, undoing, for example, the effects of file system repair.

### Application Processor Number

```
syslocal(SYSL_APNUM)
```

Return the processor number of the Application Processor on which this process is executing.

# syslocal

## Total Application Processors

```
syslocal(SYSL_TOTAPS)
```

Return the total number of Application Processors currently running.

## Console Control

```
syslocal(SYSL_CONSOLE, type, action)
int type, action;
```

Manage Application Processor console. Affects AP on which this process is running. *Type* specifies the type of action, *action* the specific action. Values of *type* are: 0 to query console status, 1 to associate the terminal with a terminal, 2 to control kernel prints, and 3 to control entry to the kernel debugger.

If *type* is 0 and *action* is 1, the return value indicates the terminal association of the console: a positive value is the terminal number of the associated terminal; -1 indicates that no terminal is associated with the console.

If *type* is 0 and *action* is 2, the return value gives the status of kernel diagnostic prints: 0 for off, 1 for on.

If *type* is 0 and *action* is 3, the return value tells whether entry to the kernel debugger is enabled: 0 for no, 1 for yes.

If *type* is 0 and *action* is 4, the contents of the console's circular buffer are written to standard output.

If *type* is 1, *action* indicates a new terminal association for the console. If *action* is 0, terminal association is removed. If *action* is -1, the console is associated with the UART kludge port. If *action* is positive, it must be the file descriptor for an open terminal special file; the console is associated with that terminal. If the terminal is under window management, then the file descriptor refers to one of the windows in that terminal; the console is associated with that particular window. A return value of 0 indicates a successful association, a -1 indicates an unsuccessful association, with the error value set in **errno**.

## syslocal

If *type* is 2, *action* controls diagnostic prints: 0 disables, any other value enables.

If *type* is 3, *action* controls access to the kernel debugger: 0 disables, 1 enables, and any other value must be a process group whose terminal/window is to have kernel prints enabled. When access to the kernel debugger is enabled, entering a CTRL-B or CODE-B on the console terminal enters the kernel debugger.

### Maximum Number of Users

```
syslocal(SYSL_MAXUSERS)
```

Returns maximum number of concurrent logins on the processor on which this process is executing.

### PILF File Status

Note that the following calls must be compiled with the **-D PILF** option.

```
#include <prof.h>
#include <stat.h>
#include <types.h>

syslocal(SYSL_PSTAT, name, st_buf)
char *name
struct p_stat *st_buf;

syslocal(SYSL_PFSTAT, fd, st_buf)
int fd;
struct p_stat *st_buf;

struct p_stat
{
        dev_t    st_dev;
        ino_t    st_ino;
        ushort   st_mode;
        short    st_nlink;
        ushort   st_uid;
        ushort   st_gid;
        dev_t    st_rdev;
```

# syslocal

```
        off_t     st_size;
        time_t    st_atime;
        time_t    st_mtime;
        time_t    st_ctime;
        char      st_cluster;
}
```

These calls work exactly like **stat** and **fstat** (see **stat**), except that the status structure has one additional field, *st_cluster,* which gives the cluster size exponent of the file.

## Get Process's Cluster Size Exponent

```
syslocal(SYSL_GETCLUS)

syslocal(SYSL_SETCLUS, cluster)
int cluster;
```

A process's cluster size exponent sets the cluster size exponent of any files the process creates on PILF file systems. A process's cluster size exponent can be -1, indicating that the new file's cluster size exponent should be taken from the file system's default cluster size exponent. A new process inherits its parent's exponent.

**syslocal** SYSL_GETCLUS returns a positive value if a previous SYSL_SETCLUS was issued; otherwise, -1 is returned.

**syslocal** SYSL_SETCLUS sets the process's cluster size exponent to *cluster.*

## Caution

Kernel prints and the kernel debugger **syslocal** calls that support them may disappear without notice. Use of kernel prints degrades system performance. Use of the kernel debugger halts normal processing.

## See Also

**apnum, fsck** in Section 1; **openi; pilf** in Section 5; **console** in Section 6.

# time

## Name

**time** - get time.

## Format

```
long time((long *) 0)

long time (tloc)
long *tloc;
```

## Description

The **time** system call returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

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

| | |
|---|---|
| [EFAULT] | **time** will fail if *tloc* points to an illegal address. |

## Returns

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

## See Also

**stime**.

# times

## Name

**times** - get process and child process times

## Format

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

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

## Description

The **times** system call fills the structure indicated by *buffer* with time-accounting information. The structure takes the following form:

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

The time accounting information comes from the calling process and each of its terminated child processes for which it has executed a **wait**. Times are in 60ths of a second.

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

## times

*Tms_stime* is the CPU time used by the system on behalf of the calling process.

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

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

The **times** call will fail if

| | |
|---|---|
| [EFAULT] | *Buffer* points to an illegal address. |

### Returns

Upon successful completion, **times** returns the elapsed real time, in 60ths of a second, since an arbitrary point in the past (such as 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.

### See Also

**exec, fork, time, wait.**

# ulimit

## Name

**ulimit** - get and set user limits

## Format

```
long ulimit (cmd, newlimit)
int cmd;
long newlimit;
```

## Description

This system call provides for control over process limits. The *cmd* values available are:

| | |
|---|---|
| 1 | Get the 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 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** will fail and the limit will be unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit. [EPERM] |
| 3 | Get the maximum possible break value. See **brk**. |

## Returns

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

## See Also

**brk, write**.

# umask

## Name

**umask** - set and get the file creation mask

## Format

```
int umask (cmask)
int cmask;
```

## Description

The **umask** system call sets the process file mode creation mask to *cmask*. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

## Returns

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

## See Also

**mkdir**, **sh** in Section 1; **chmod**, **creat**, **mknod**, **open**.

# umount

## Name

**umount** - unmount a file system

## Format

```
int umount (spec)
char *spec;
```

## Description

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

**umount** may be invoked only by the super-user.

**umount** fails if one or more of the following are true:

| | |
|---|---|
| [EPERM] | The process's effective user ID is not super-user. |
| [ENXIO] | *Spec* does not exist. |
| [ENOTBLK] | *Spec* is not a block special device. |
| [EINVAL] | *Spec* is not mounted. |
| [EBUSY] | A file on *spec* is busy. |
| [EFAULT] | *Spec* points to an illegal address. |

## Returns

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

## See Also

**mount**.

## uname

### Name

**uname** - get name of current CENTIX system

### Format

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

int uname (name)
struct utsname *name;
```

### Description

**uname** stores information identifying the current CENTIX system in the structure pointed to by *name*.

**uname** uses the structure defined in <sys/utsname.h> whose members are:

```
char        sysname[9];
char        nodename[9];
char        release[9];
char        version[9];
char        machine[9];
```

**uname** returns a null-terminated character string naming the current CENTIX 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 CENTIX system is running on.

[EFAULT] **uname** will fail if *name* points to an invalid address.

## uname

### Returns

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

### See Also

**uname** in Section 1.

# unlink

## Name

**unlink** - remove directory entry

## Format

```
int unlink (path)
char *path;
```

## Description

The **unlink** system call 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. |
| [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 allocated address space of the process. |

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.

# unlink

## Returns

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

## See Also

**rm** in Section 1; **close**, **link**, **open**.

# ustat

## Name

**ustat** - get file system statistics

## Format

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

int ustat (dev, buf)
int dev;
struct ustat *buf;
```

## Description

The **ustat** system call returns information about a mounted file system. *Dev* is a device number that identifies the 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*/
no_t       f_tinode;      /*Number of free i-nodes*/
char       f_fname[6];    /*Filsys name*/
char       f_fpack[6];    /*Filsys pack name*/
```

**ustat** fails if one or both 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 allocated address space of the process. |

## Returns

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

## See Also

**stat**; **fs** in Section 4.

# utime

## Name

**utime** - set file access and modification times

## Format

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

int utime (path, times)
char *path;
struct times *tm;
```

## Description

*Path* points to a path name naming a file. The **utime** system call sets the access and modification times of the named file.

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

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

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

```
        struct     times {
                   time_t   acc_time;    /* access time */
                   time_t   mod_time;    /* modification time */

        };
```

# utime

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

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

## Returns

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

## See Also

**stat.**

# wait

## Name

**wait** - wait for a child process to stop or terminate

## Format

```
int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)
```

## Description

The **wait** system call suspends the calling process until one of the immediate children terminates or until a child that is being traced stops because it has hit a break point. The call will return prematurely if a signal is received; if a child process stopped or terminated prior to the call on **wait**, return is immediate.

If *stat_loc* (taken as an integer) is non-zero, 16 bits of information called status are stored in the low order 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 has 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 contain 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 contain the low order 8 bits of the arguments that the child process passed to **exit**. See **exit**.

# wait

If the child process terminated due to a signal, the high order 8 bits of status are zero and the low order 8 bits contain the number of the signal that caused the termination. In addition, if the low order seventh bit (for example, bit 200) is set, a core image is produced.

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 that the initialization process inherits the child processes. See **intro**.

**wait** fails and returns immediately if one or both of the following are true:

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

## Returns

If **wait** returns due to 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.

## Caution

SIGCLD (termination of a created process) affects **wait**. If the *func* of SIGCLD is set to SIG_IGN (ignore signal) and a **wait** is executed, the **wait** blocks until all created processes of the calling process terminate. It then returns a value of -1 with **errno** set to ECHILD.

## See Also

**exec, exit, fork, intro, pause, ptrace, signal.**

# write

## Name

**write** - write on a file

## Format

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

## Description

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

The **write** system call attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from **write**, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

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

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

| | |
|---|---|
| [EBADF] | *Fildes* is not a valid file descriptor open for writing. |
| [EPIPE and SIGPIPE signal] | An attempt is made to write to a pipe that is not open for reading by any process. |
| [EFBIG] | An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See **ulimit**. |
| [EFAULT] | *Buf* points outside the process's allocated address space. |
| [EINTR] | A signal was caught during the **write** system call. |

# write

| | |
|---|---|
| [ENOSPC] | Additional blocks cannot be allocated to the file because the file system has no free blocks or because a PILF file's cluster size exceeds the size of the unallocated clusters. |
| [EDEADLOCK] | A side effect of a previous **locking** call. |

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

If the file being written is a pipe (or FIFO), no partial writes are permitted. Thus, the write fails if a write of nbyte bytes exceeds a limit.

If the file being written is a pipe (or FIFO), and the O_NDELAY flag of the file flag word is set, then a write to a full pipe returns a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe block until space becomes available.

## Returns

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

## See Also

**creat, dup, lseek, locking, open, pipe.**

# Section 3

# Library Functions

## intro

### Name

**intro** - introduction to libraries and subroutines

### Description

This section describes functions found in various libraries (other than those functions that directly invoke CENTIX system primitives, which are described in Section 2). The functions are divided into four major categories:

- The Standard C Library functions. These functions, along with those in Section 2 and those in the Standard I/O Package (below), constitute the Standard C Library, libc. The libc library is automatically loaded by the C compiler, **cc** (see Section 1). The link editor **ld** (Section 1) searches this library under the **-lc** option. Declarations for some of these functions may be obtained from #include files indicated on the appropriate pages.
- The Math Library functions. These functions constitute the Math Library, libm. They are not automatically loaded by the C compiler, **cc**; however, the link editor searches this library under the **-lm** option. Declarations for these functions may be obtained from the #include file <math.h>.
- The Standard I/O Package functions. These functions are in the library libc, mentioned earlier. Declarations for these functions may be obtained from the #include file <stdio.h>.
- Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

Two groups of entries represent direct communication with BTOS. Functions whose names begin with **of** (outside file system) provide BTOS-style input/output. Functions whose names begin with **qu** (queue) provide access to BTOS queue management.

## intro

For convenience, many of the functions in this section are grouped under single headings. The following table lists all functions in each of the above categories, along with the entries under which the functions should be referenced.

Table 3-1 **Library Functions**

In the C Library:

| **Function** | **Reference** | **Description** |
|---|---|---|
| a64l | a64l | Convert between long integer and base-64 ASCII string. |
| abort | abort | Generate an IOT fault. |
| abs | abs | Return integer absolute value. |
| asctime | ctime | Convert date and time to string. |
| atof | atof | Convert ASCII string to floating-point number. |
| atof | strtod | Convert string to double-precision number. |
| atoi | strtol | Convert string to integer. |
| atol | strtol | Convert string to integer. |
| bsearch | bsearch | Binary search a sorted table. |
| calloc | malloc | Main memory allocator. |
| clock | clock | Report CPU time used. |
| crypt | crypt | Generate DES encryption. |
| ctime | ctime | Convert date and time to string. |
| dial | dial | Establish an out-going terminal line connection. |
| drand48 | drand48 | Generate uniformly distributed pseudo-random numbers. |
| ecvt | ecvt | Convert floating-point number to string. |
| edata | end | Last locations in programs. |
| encrypt | crypt | Generate DES encryption. |
| end | end | Last locations in programs. |

## intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| endgrent | getgrent | Close group file entry. |
| endpwent | getpwent | Close password file entry. |
| endutent | getut | Close utmp file entry. |
| erand48 | drand48 | Generate uniformly distributed pseudo-random numbers. |
| errno | perror | System error messages. |
| etext | end | Last locations in programs. |
| fcvt | ecvt | Convert floating-point number to string. |
| fgetc | getc | Get character from a stream. |
| fgetgrent | getgrent | Get group file entry. |
| fgetpwent | getpwent | Get password file entry. |
| free | malloc | Main memory allocator. |
| frexp | frexp | Manipulate parts of floating-point numbers. |
| ftw | ftw | Walk a file tree. |
| gcvt | ecvt | Convert floating-point number to string. |
| getc | getc | Get character from a stream. |
| getchar | getc | Get character from a stream. |
| getcwd | getcwd | Get the path-name of the current working directory. |
| getenv | getenv | Return value for environment name. |
| getgrent | getgrent | Get group file entry. |
| getgrgid | getgrent | Get group file id. |
| getgrnam | getgrent | Get group file name. |
| getlogin | getlogin | Get login name. |
| getopt | getopt | Get option letter from argument vector. |
| getpass | getpass | Read a password. |

## intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| getpw | getpw | Get name from UID. |
| getpwent | getpwent | Get password file entry. |
| getpwnam | getpwent | Get password file name. |
| getpwuid | getpwent | Get password file user id. |
| getutent | getut | Access utmp file entry. |
| getutid | getut | Access utmp file entry. |
| getutline | getut | Access utmp file entry. |
| getw | getc | Get word from a stream. |
| gmtime | ctime | Convert date and time to string. |
| gsignal | ssignal | Software signals. |
| hcreate | hsearch | Create hash tables. |
| hdestroy | hsearch | Destroy hash tables. |
| hsearch | hsearch | Search hash tables. |
| isalnum | ctype | Determine if a character is alphanumeric. |
| isalpha | ctype | Determine if a character is alphabetic. |
| isascii | ctype | Determine if an integer is an ASCII character. |
| isatty | ttyname | Find name of a terminal. |
| iscntrl | ctype | Determine if a character is a control character. |
| isdigit | ctype | Determine if a character is a decimal digit. |
| isgraph | ctype | Determine if a character is printable. |
| islower | ctype | Determine if a character is a lower case letter. |
| isprint | ctype | Determine if a character is printable. |
| ispunct | ctype | Determine if a character is a punctuation character. |
| isspace | ctype | Determine if a character is a white space character. |

# intro

Table 3-1 **Library Functions (Cont.)**

| Function | Reference | Description |
|---|---|---|
| isupper | ctype | Determine if a character is an upper case letter. |
| isxdigit | ctype | Determine if a character is a hexadecimal digit. |
| jrand48 | drand48 | Generate uniformly distributed pseudo-random numbers. |
| l3tol | l3tol | Convert between 3-byte integers and long integers. |
| l64a | a64l | Convert between long integer and base-64 ASCII string. |
| lcong48 | drand48 | Generate uniformly distributed pseudo-random numbers. |
| ldexp | frexp | Manipulate parts of floating-point numbers. |
| lfind | lsearch | Linear search and update. |
| localtime | ctime | Convert date and time to string. |
| longjmp | setjmp | Non-local goto. |
| lrand48 | drand48 | Generate uniformly distributed pseudo-random numbers. |
| lsearch | lsearch | Linear search and update. |
| ltol3 | l3tol | Convert between 3-byte integers and long integers. |
| malloc | malloc | Main memory allocator. |
| memccpy | memory | Copy characters from one memory area to another. |
| memchr | memory | Search for specified character in a block of memory. |
| memcmp | memory | Compare blocks of memory. |
| memcpy | memory | Copy one block of memory to another. |
| memset | memory | Set a block of memory to a specified value. |
| mktemp | mktemp | Make a unique file name. |

# intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| modf | frexp | Manipulate parts of floating-point numbers. |
| monitor | monitor | Prepare execution profile. |
| mrand48 | drand48 | Generate uniformly distributed pseudo-random numbers. |
| nlist | nlist | Get entries from the name list. |
| nrand48 | drand48 | Generate uniformly distributed pseudo-random numbers. |
| perror | perror | System error messages. |
| putenv | putenv | Change or add value to environment. |
| putpwent | putpwent | Write password file entry. |
| pututline | getut | Write out supplied utmp structure into the utmp file. |
| qsort | qsort | Quicker sort. |
| rand | rand | Random number generator. |
| realloc | malloc | Main memory allocator. |
| seed48 | drand48 | Seed uniformly distributed pseudo-random number generator. |
| setgrent | getgrent | Reset group file to allow repeated searches. |
| setjmp | setjmp | Non-local goto. |
| setkey | crypt | Generate DES encryption. |
| setpwent | getpwent | Reset password file to allow repeated searches. |
| setutent | getut | Reset input stream to beginning of utmp file. |
| sleep | sleep | Suspend execution for interval. |
| srand | rand | Simple random number generator. |
| srand48 | drand48 | Generate uniformly distributed pseudo-random numbers. |
| ssignal | ssignal | Software signals. |

## intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| stdipc | stdipc | Standard interprocess communication package (ftok). |
| strcat | string | Concatenate two strings. |
| strchr | string | Search a string for a character. |
| strcmp | string | Compare two strings. |
| strcpy | string | Copy a string over another string. |
| strcspn | string | Determine the length of an initial segment of a string. |
| strlen | string | Determine the length of a string. |
| strncat | string | Append one string to another. |
| strncmp | string | Compare two strings. |
| strncpy | string | Copy one string over another string. |
| strpbrk | string | Search a string for a specified set of characters. |
| strrchr | string | Search a string in reverse order for a specified character. |
| strspn | string | Determine the length of an initial string. |
| strtod | strtod | Convert string to double-precision number. |
| strtok | string | Search a string for a token. |
| strtol | strtol | Convert string to long integer. |
| swab | swab | Swap bytes. |
| sys_errlist | perror | System error messages. |
| sys_nerr | perror | System error messages. |
| tdelete | tsearch | Delete a node from a binary search tree. |
| tfind | tsearch | Search for data in a binary search tree. |
| toascii | conv | Translate characters to ASCII. |
| tolower | conv | Convert a character to lower case. |
| _tolower | conv | Convert an upper case letter to lower case. |

# intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| toupper | conv | Convert a character to upper case. |
| _toupper | conv | Convert a lower case letter to upper case. |
| tsearch | tsearch | Build and access binary search tree. |
| twalk | tsearch | Walk a binary search tree. |
| ttyname | ttyname | Find name of a terminal. |
| ttyslot | ttyslot | Find the slot in the utmp file of the current user. |
| tzset | ctime | Convert date and time to string. |
| undial | dial | Release an out-going terminal line connection. |
| utmpname | getut | Change utmp file name. |

In the Math Library:

| | | |
|---|---|---|
| acos | trig | Arccosine function. |
| asin | trig | Arcsine function. |
| atan | trig | Arctangent function. |
| atan2 | trig | Arctangent function. |
| ceil | floor | Ceiling function. |
| cos | trig | Cosine function. |
| cosh | sinh | Hyperbolic cosine function. |
| erf | erf | Error function. |
| erfc | erf | Complementary error function. |
| exp | exp | Exponential function. |
| fabs | floor | Absolute value function. |
| floor | floor | Floor function. |
| fmod | floor | Remainder function. |
| gamma | gamma | Log gamma function. |
| hypot | hypot | Euclidean distance function. |

## intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| j0 | Bessel | Bessel function. |
| j1 | Bessel | Bessel function. |
| jn | Bessel | Bessel function. |
| log | exp | Logarithm function. |
| log10 | exp | Logarithm base ten function. |
| matherr | matherr | Error handling function. |
| pow | exp | Power function. |
| sin | trig | Sine function. |
| sinh | sinh | Hyperbolic sine function. |
| sqrt | exp | Square root function. |
| tan | trig | Tangent function. |
| tanh | sinh | Hyperbolic tangent function. |
| y0 | Bessel | Bessel function. |
| y1 | Bessel | Bessel function. |
| yn | Bessel | Bessel function. |

In the Standard I/O Package:

| | | |
|---|---|---|
| clearerr | ferror | Stream status inquiry. |
| ctermid | ctermid | Generate file name for terminal. |
| cuserid | cuserid | Get character login name of the user. |
| fclose | fclose | Close a stream. |
| fdopen | fopen | Open a stream. |
| feof | ferror | Stream status inquiry. |
| ferror | ferror | Stream status inquiry. |
| fflush | fclose | Flush a stream. |
| fgetc | getc | Get a character from a stream. |
| fgets | gets | Get a string from a stream. |

## intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| fileno | ferror | Stream status inquiry. |
| fopen | fopen | Open a stream. |
| fprintf | printf | Print formatted output. |
| fputc | putc | Put a character on a stream. |
| fputs | puts | put a string on a stream. |
| fscanf | scanf | Convert formatted input. |
| fseek | fseek | Reposition a file pointer in a stream. |
| fread | fread | Read from binary input. |
| freopen | fopen | Open a stream. |
| ftell | fseek | Reposition a file pointer in a stream. |
| fwrite | fread | Write to binary output. |
| getc | getc | Get character from a stream. |
| getchar | getc | Get character from a stream. |
| gets | gets | Get a string from a stream. |
| getw | getc | Get a word from a stream. |
| pclose | popen | Close a stream opened by popen. |
| popen | popen | Initiate pipe to/from a process. |
| printf | printf | Print formatted output. |
| putc | putc | Put character on a stream. |
| putchar | putc | Put character on a stream. |
| puts | puts | Put a string on a stream. |
| putw | putc | Put a word on a stream. |
| rewind | fseek | Reposition a file pointer in a stream. |
| scanf | scanf | Convert formatted input. |
| setbuf | setbuf | Assign buffering to a stream. |
| setubuf | setbuf | Assign buffering to a stream. |
| sprintf | printf | Print formatted output. |

## intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| sscanf | scanf | Convert formatted output. |
| stdio | stdio | Standard buffered input/output package. |
| system | system | Issue a shell command. |
| tempnam | tmpnam | Create a name for a temporary file. |
| tmpfile | tmpfile | Create a temporary file. |
| tmpnam | tmpnam | Create a name for a temporary file. |
| ungetc | ungetc | Push character back into input stream. |
| vfprintf | vprintf | Print formatted output of a varargs argument list. |
| vprintf | vprintf | Print formatted output of a varargs argument list. |
| vsprintf | vprintf | Print formatted output of a varargs argument list. |

In Various Specialized Libraries:

| | | |
|---|---|---|
| assert | assert | Verify program assertion. |
| calloc | malloc (fast version) | Fast main memory allocator. |
| curses | curses | CRT screen handling and optimization package. |
| free | malloc (fast version) | Fast main memory allocator. |
| ldaclose | ldclose | Close a common object file. |
| ldahread | ldahread | Read the archive header of a member of an archive file. |
| ldaopen | ldopen | Open a common object file for reading. |
| ldclose | ldclose | Close a common object file. |
| ldfhread | ldfhread | Read the file header of a common object file. |
| ldgetname | ldgetname | Retrieve symbol name for common object file symbol table entry. |
| ldlinit | ldlread | Manipulate line number entries of a common object file function. |

## intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| ldlitem | ldlread | Manipulate line number entries of a common object file function. |
| ldlread | ldlread | Manipulate line number entries of a common object file function. |
| ldlseek | ldlseek | Seek to line number entries of a section of a common object file. |
| ldnlseek | ldlseek | Seek to line number entries of a section of a common object file. |
| ldnrseek | ldrseek | Seek to relocation entries of a section of a common object file. |
| ldnshread | ldshread | Read an indexed/named section header of a common object file. |
| ldnsseek | ldsseek | Seek to an indexed/named section of a common object file. |
| ldohseek | ldohseek | Seek to the optional file header of a common object file. |
| ldopen | ldopen | Open a common object file for reading. |
| ldrseek | ldrseek | Seek to relocation entries of a section of a common object file. |
| ldshread | ldshread | Read an indexed/named section header of a common object file. |
| ldsseek | ldsseek | Seek to an indexed/named section of a common object file. |
| ldtbindex | ldtbindex | Compute the index of a symbol table entry of a common object file. |
| ldtbread | ldtbread | Read an indexed symbol table entry of a common object file. |
| ldtbseek | ldtbseek | Seek to the symbol table of a common object file. |
| logname | logname | Return login name of user. |
| mallinfo | malloc (fast version) | Provide instrumentation describing space usage for malloc (fast version). |
| malloc | malloc (fast version) | Fast main memory allocator. |

## intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| mallopt | malloc (fast version) | Provide for control over the malloc(1) allocation algorithm. |
| ocurse | ocurse | Optimized screen functions. |
| ofChangeFileLength | ofCreate | Reset length of a BTOS file. |
| ofCloseAllFiles | ofOpenFile | Close all BTOS files. |
| ofCloseFile | ofOpenFile | Close a BTOS file. |
| ofCrDir | ofDir | Create a BTOS directory. |
| ofCreate | ofCreate | Create a BTOS file. |
| ofDelete | ofCreate | Delete a BTOS file. |
| ofDlDir | ofDir | Delete an empty BTOS directory. |
| ofGetFileStatus | ofStatus | Get BTOS file information. |
| ofOpenFile | ofOpenFile | Open a BTOS file. |
| ofRead | ofRead | Input one or more sectors from a BTOS file. |
| ofReadDirSector | ofDir | Read a single BTOS 512-byte directory sector. |
| ofRename | ofRename | Rename a BTOS file. |
| ofSetFileStatus | ofStatus | Set BTOS file information. |
| ofWrite | ofRead | Output one or more sectors to a BTOS file. |
| quAdd | quAdd | Add a new entry to a BTOS queue. |
| quReadKeyed | quRead | Examine a BTOS queue. |
| quReadNext | quRead | Examine a BTOS queue. |
| quRemove | quRemove | Take back a BTOS queue request. |
| regcmp | regcmp | Compile a regular expression. |
| regex | regcmp | Execute a regular expression. |
| sgetl | sputl | Access long integer data in a machine-dependent fashion. |

## intro

Table 3-1 **Library Functions (Cont.)**

| **Function** | **Reference** | **Description** |
|---|---|---|
| spawnlp | spawn | Execute a process on a specific Application Processor. |
| spawnvp | spawn | Execute a process on a specific Application Processor. |
| sputl | sputl | Access long integer data in a machine-dependent fashion. |
| spwait | spwait | Wait for a spawned process to terminate. |
| swaplong | swapshort | Translate byte orders to Motorola/Intel. |
| swapshort | swapshort | Translate byte orders to Motorola/Intel. |
| tgetent | termcap | Get terminal entry. |
| tgetflag | termcap | Determine if a terminal has boolean capability. |
| tgetnum | termcap | Get value of terminal numeric capability. |
| tgetstr | termcap | Interpret value of terminal string capability. |
| tgoto | termcap | Move cursor. |
| tputs | termcap | Direct output of string returned by tgetstr or tgoto. |
| wmgetid | wmgetid | Get window ID. |
| wmlayout | wmlayout | Get terminal's window layout. |
| wmop | wmop | Window management operations. |
| wmsetid | wmsetid | Associate a file descriptor with a window. |
| wmsetids | wmsetids | Associate a file descriptor with a window. |

# intro

## Definitions

| | |
|---|---|
| character | Any bit pattern able to fit into a byte on the machine. |
| null character | A character with value 0, represented in the C language as '\0.' |
| character array | A sequence of characters. |
| null-terminated character array | A sequence of characters, the last of which is the null character. |
| string | a designation for a null-terminated character array. |
| null string | A character array containing only the null character. |
| NULL pointer | 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 do so to indicate errors. |
| NULL | Defined as 0 in <stdio.h>; you can include your own definition if you are not using <stdio.h>. |

## Files

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

## Diagnostics

Functions in the Math Library may return the conventional values 0 or HUGE (the largest single-precision floating-point number) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable **errno** (see **intro** in Section 2) is set to the value EDOM or ERANGE.

## intro

### Caution

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 inadvertantly 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** program checker (see Section 1) reports name conflicts of this kind as "multiple declarations" of the names in question. Definitions for Section 2 and for Standard C Library and Standard I/O functions of Section 3 are checked automatically. Other definitions can be included by using the **-l** option (for example, **-lm** includes definitions for the Math Library, libm). Use of **lint** is highly recommended.

### See Also

**ar**, **cc**, **ld**, **nm** in Section 1; **intro** in Section 2; **stdio**.

# a64l

## Name

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

## Format

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

## Description

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

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

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

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

## Known Problems

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

# abort

## Name

**abort** - generate an IOT fault

## Format

```
int abort ()
```

## Description

The **abort** function first closes all open files, if possible, then causes an IOT signal to be sent to the process. This usually results in termination with a core dump.

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

## Diagnostics

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

## See Also

**adb** in Section 1; **exit, kill, signal** in Section 2.

## abs

### Name

**abs** - return integer absolute value

### Format

```
int abs (i)
int i;
```

### Description

The **abs** function returns the absolute value of its integer operand.

### Known Problems

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

### See Also

**floor**.

# assert

## Name

**assert** - verify program assertion

## Format

```
#include <assert.h>

assert (expression)
int expression;
```

## Description

This function is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), **assert** prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* is the source line number of the **assert** statement.

Compiling with the preprocessor option **-DNDEBUG** (see **cpp** in Section 1), or with the preprocessor control statement "#define NDEBUG" ahead of the "#include <assert.h>" statement, will stop assertions from being compiled into the program.

## See Also

**cpp** in Section 1; **abort**.

## atof

### Name

**atof** - convert ASCII string to floating-point number

### Format

```
double atof (nptr)
char *nptr;
```

### Description

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

### Diagnostics

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

### See Also

**scanf**.

# Bessel

## Name

**j0**, **j1**, **jn**, **y0**, **y1**, **yn** - Bessel functions

## Format

```
#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.

## Bessel

### 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**.

### See Also

**matherr**.

## bsearch

### Name

**bsearch** - binary search a sorted table

### Format

```
#include <search.h>

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

### Description

The **bsearch** function 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 following example 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.

## bsearch

```
#include <stdio.h>
#include <search.h>

#define TABSIZE     1000

struct node {                   /*these are stored in 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("%", 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.string);

                }
        }
}
/*
        This routine compares two nodes based on an
        alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
struct node *node1, *node2;
{
        return strcmp(node1->string, node2->string);
}
```

## bsearch

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.

### Diagnostics

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

### See Also

**hsearch**, **lsearch**, **qsort**, **tsearch**.

# clock

## Name

**clock** - report CPU time used

## Format

```
long clock ();
```

## Description

The **clock** function returns the amount of CPU time (in microseconds) used since the first call to **clock**. The time reported is the sum of user and system times of the calling process and its terminated child processes for which it has executed a **wait** system call or **system** library function. The return value will vary based on system usage.

The resolution of the clock is 16.667 microseconds on CENTIX processors.

## Known Problems

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

## See Also

**times**, **wait** in Section 2; **system**.

## conv

### Name

**toupper, tolower, _toupper, _tolower, toascii** - translate characters

### Format

```
#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

The **toupper** and **tolower** functions have as domain the range of the **getc** library function: the integers from -1 through 255. If the argument of **toupper** represents a lower case letter, the result is the corresponding upper case letter. If the argument of **tolower** represents an upper case letter, the result is the corresponding lower case letter. All other arguments in the domain are returned unchanged.

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

## conv

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

### See Also

**ctype, getc.**

# crypt

## Name

**crypt, setkey, encrypt** - generate DES encryption

## Format

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

void setkey (key)
char *key;

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

## Description

The **crypt** function is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES of key hardware implementations of the DES for key search.

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

The **setkey** and **encrypt** entries provide (rather primitive) access to the actual DES algorithm. The argument of **setkey** is a character array of length 64 containing only the characters with numerical value 0 and 1. If the 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 to the machine. This is the key that will be used with the above mentioned algorithm to encrypt or decrypt the string *block* with the function **encrypt**.

## crypt

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

### Known Problems

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

### See Also

**login**, **passwd** in Section 1; **getpass**; **passwd** in Section 4.

## ctermid

### Name

**ctermid** - generate file name for terminal

### Format

```
#include <stdio.h>

char *ctermid (s)
char *s;
```

### Description

The **ctermid** function generates the path name of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string containing the path name is stored in an internal static area, the contents of which are overwritten by the next call to **ctermid**, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least L_ctermid elements; the path name is placed in this array and the value of *s* is returned. The constant L_ctermid is defined in the <stdio.h> header file.

The difference between **ctermid** and **ttyname** is that **ttyname** must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while **ctermid** returns a string (/dev/tty) that will refer to the terminal if used as a file name. Thus, **ttyname** is useful only if the process already has at least one file open to a terminal.

### See Also

**ttyname**.

# ctime

## Name

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

## Format

```
#include <time.h>

char *ctime (clock)
long *clock;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

extern long timezone

extern int daylight

extern char *tzname[2];

void tzset ()
```

## Description

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

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

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

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

## ctime

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

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

*Tm_isdst* is nonzero if Daylight Savings Time is in effect.

The external long variable *timezone* contains the difference in seconds between GMT and local standard time (in EST, *timezone* is 5*60*60). The external variable *daylight* is nonzero if and only if the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

If an environment variable named TZ (time zone) is present, **asctime** uses the contents of the variable to override the default timezone. The value of TZ must be a three-letter timezone name, followed by a number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional three-letter name for a daylight time zone. For example, the setting for New Jersey would be EST5EDT. Setting TZ changes the value of the external variables *timezone* and *daylight;*in addition, the time zone names contained in the external variable

    char *tzname[2] = {"EST","EDT"};

are set from the environment variabel TZ. The function **tzset** sets these external variables from TZ; **tzset** is called by **asctime** and may also be called explicitly by the user.

Note that in most installations, TZ is set by default when the user logs in, to a value in the local /etc/profile file (see **profile** in section 4).

## ctime

### Known Problems

The return values point to static data whose content is overwritten by each call.

### See Also

**time** in Section 2; **getenv**; **profile** in Section 4; **environ** in Section 5.

# ctype

## Name

**isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii** - classify characters

## Format

```
#include <ctype.h>

int isalpha (c)
int c;

. . .
```

## Description

These macro functions classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. The function **isascii** is defined on all integer values; the rest are defined only where **isascii** is true and on the single non-ASCII value EOF (-1 - see **stdio**).

| | |
|---|---|
| **isalpha** | *c* is a letter. |
| **isupper** | *c* is an upper case letter. |
| **islower** | *c* is a lower case letter. |
| **isdigit** | *c* is a digit [from 0 to 9]. |
| **isxdigit** | *c* is a hexadecimal digit [0-9], [A-F] and [a-f]. |
| **isalnum** | *c* is an alphanumeric (letter or digit). |
| **isspace** | *c* is a space, tab, carriage return, new-line, vertical tab, or form-feed. |
| **ispunct** | *c* is a punctuation character (neither control nor alphanumeric). |
| **isprint** | *c* is a printing character, ASCII octal code 040 (space) through 0176 (tilde). |
| **isgraph** | *c* is a printing character, like **isprint** except it is false for space. |

# ctype

| | |
|---|---|
| **iscntrl** | *c* is a delete character (0177) or an ordinary control character (less than 040). |
| **isascii** | *c* is an ASCII character code less than 0200. |

## Diagnostics

If the argument to any of these macros is not in the domain of the function, the result is undefined.

## curses

### Name

**curses** - CRT screen handling and optimization package

### Format

```
#include <curses.h>
cc [flags] files -lcurses [libraries]
```

### Description

These routines give the user a method of updating screens with reasonable optimization. In order to initialize the routines, the routine **initscr()** must be called before any of the other routines that deal with windows and screens are used. The routine **endwin()** should be called before exiting. To get character-at-a-time input without echoing (most interactive, screen-oriented programs want this), after calling **initscr()** you should call "**nonl(); cbreak(); noecho();**"

The full curses interface permits manipulation of data structures called windows, which can be thought of as two dimensional arrays of characters representing all or part of a CRT screen. A default window called *stdscr* is supplied, and others can be created with **newwin**. Windows are referred to by variables declared "WINDOW *," the type WINDOW is defined in curses.h to be a C structure. These data structures are manipulated with functions described below, among which the most basic are **move** and **addch**. (More general versions of these functions are included with names beginning with w, allowing you to specify a window. The routines not beginning with w affect *stdscr*.) Then **refresh()** is called, telling the routines to make the user's CRT screen look like *stdscr*.

Mini-Curses is a subset of curses which does not allow manipulation of more than one window. To invoke this subset, use -DMINICURSES as a **cc** option. This level is smaller and faster than full curses.

## curses

If the environment variable TERMINFO is defined, any program using curses will check for a local terminal definition before checking in the standard place. For example, if the standard place is /usr/lib/terminfo, and TERM is set to "vt100," then normally the compiled file is found in /usr/lib/terminfo/v/vt100. (The "v" is copied from the first letter of "vt100" to avoid creation of huge directories.) However, if TERMINFO is set to /usr/mark/myterms, curses will first check /usr/mark/myterms/v/vt100, then, if that fails, /usr/lib/terminfo/v/vt100. This is useful for developing experimental definitions or when write permission in /usr/lib/terminfo is not available.

### Functions

Routines listed in Table 3-2 may be called when using the full curses. Those marked with an asterisk may be called when using Mini-Curses

Table 3-2 **Curses Routines**

| | |
|---|---|
| addstr(ch)* | Add a character to *stdscr* (like **putchar**) (wraps to next line at end of line). |
| addstr(str)* | Calls **addch** with each character in *str.* |
| attroff(attrs)* | Turns off attributes named. |
| attron(attrs)* | Turns on attributes named. |
| attrset(attrs)* | Set current attributes to *attrs.* |
| baudrate()* | Current terminal speed. |
| beep()* | Sound beep on terminal. |
| box(win, vert, hor) | Draw a box around the edges of *win. Vert* and *hor* are chars to use for vert. and horiz. edges of box. |
| clear() | Clear *stdscr.* |
| clearok(win, bf) | Clear screen before next redraw of *win.* |
| clrtobot() | Clear to bottom of *stdscr.* |
| clrtoeol() | Clear to end of line on *stdscr.* |
| cbreak()* | Set break mode. |
| delay_output(ms)* | Insert *ms* millisecond pause in output. |

## curses

Table 3-2 **Curses Routines (Cont.)**

| | |
|---|---|
| delch() | Delete a character. |
| deleteln() | Delete a line. |
| delwin(win) | Delete *win*. |
| doupdate() | Update screen from all **wnoutrefresh**. |
| echo()* | Set echo mode. |
| endwin()* | End window modes. |
| erase() | Erase *stdscr*. |
| erasechar() | Return user's erase character. |
| fixterm() | Restore tty to "in curses" state. |
| flash() | Flash screen or beep. |
| flushinp()* | Throw away any typeahead. |
| getch()* | Get a character from tty. |
| getstr(str) | Get a string through *stdscr*. |
| gettmode() | Establish current tty modes. |
| getyx(win, y, x) | get (*y*, *x*) co-ordinates. |
| has_ic() | True if terminal can do insert character. |
| has_il() | True if terminal can do insert line. |
| idlok(win, bf)* | Use terminal's insert/delete line if *bf* != 0. |
| inch() | get char at current (*y*, *x*) co-ordinates. |
| initscr()* | Initialize screens. |
| insch(c) | Insert a char. |
| insertln() | Insert a line. |
| intrflush(win, bf) | Interrupts flush output if *bf* is TRUE. |
| keypad(win, bf) | Enable keypad input. |
| killchar() | Return current user's kill character. |
| leaveok(win, flag) | OK to leave cursor anywhere after refresh if *flag* != 0 for *win*, otherwise cursor must be left at current position. |
| longname() | Return verbose name of terminal. |
| meta(win, flag)* | Allow meta characters on input if *flag* != 0. |
| move(y, x)* | Move to (*y*, *x*) on *stdscr*. |
| mvaddch(y, x, ch) | move(*y*, *x*) then **addch(ch)**. |
| mvaddstr(y, x, str) | Similar... |
| mvcur(oldrow, oldcol, newrow, newcol) | Low level cursor motion. |

# curses

Table 3-2 **Curses Routines (Cont.)**

| | |
|---|---|
| mvdelch(y, x) | Like **delch**, but **move(y, x)** first. |
| mvgetch(y, x) | And so on... |
| mvgetstr(y,x) | |
| mvinch(y,x) | |
| mvinsch(y, x, c) | |
| mvprintw(y, x, fmt, args) | |
| mvscanw(y, x, fmt, args) | |
| mvwaddch(win, y, x, ch) | |
| mvwaddstr(win, y, x, str) | |
| mvwdelch(win, y, x) | |
| mvwgetch(win, y, x) | |
| mvwgetstr(win, y, x) | |
| mvwin(win, by, bx) | |
| mvwinch(win, y, x) | |
| mvwinsch(win, y, x, c) | |
| mvwprintw(win, y, x, fmt, args) | |
| mvwscanw(win, y, x, fmt, args) | |
| newpad(nlines, ncols) | Create a new pad with given dimensions. |
| newterm(type, fd) | Set up a new terminal of given *type* to output on *fd*. |
| newwin(lines, cols, begin_y, begin_x) | Create a new window. |
| nl()* | Set newline mapping. |
| nocbreak()* | Unset cbreak mode. |
| nodelay(win, bf) | Enable nodelay input mode through **getch**. |
| noecho()* | Unset echo mode. |
| nonl()* | Unset newline mapping. |
| noraw()* | Unset raw mode. |
| overlay(win1, win2) | Overlay *win1* on *win2*. |
| overwrite(win1, win2) | Overwrite *win1* on top of *win2*. |
| pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol) | Like **prefresh**, but with no output until **doupdate** called. |
| prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol) | Refresh from pad starting with given upper left corner of pad with output to given portion of screen. |
| printw(fmt, arg1, arg2, ...) | **printf** on *stdscr*. |
| raw()* | Set raw mode. |
| refresh()* | Make current screen look like *stdscr*. |
| resetterm()* | Set tty modes to "out of curses" state. |
| resetty()* | Reset tty flags to stored value. |
| saveterm()* | Save current modes as "in curses" state. |

## curses

Table 3-2 **Curses Routines (Cont.)**

| | |
|---|---|
| savetty()* | Store current tty flags. |
| scanw(fmt, arg1, arg2, ...) | **scanf** through *stdscr.* |
| scroll(win) | Scroll *win* one line. |
| scrollok(win, flag) | Allow terminal to scroll if *flag* != 0. |
| set_term(new) | Now talk to terminal *new.* |
| setscrreg(t, b) | Set user scrolling region to lines *t* through *b.* |
| setterm(type) | Establish terminal with given *type.* |
| setupterm(term, filenum, errret) | Set up terminal. |
| standend()* | Clear standout mode attribute. |
| standout()* | Set standout mode attribute. |
| subwin(win, lines, cols, begin_y, begin_x) | Create a subwindow. |
| touchwin(win) | Change all of *win.* |
| traceoff() | Turn off debugging trace output. |
| traceon() | Turn on debugging trace output. |
| typeahead(fd) | Use file descriptor *fd* to check typeahead. |
| unctrl(ch)* | Printable version of *ch.* |
| waddch(win, ch) | Add char to *win.* |
| waddstr(win, str) | Add string to *win.* |
| wattroff(win, attrs) | Turn off *attrs* in *win.* |
| wattron(win, attrs) | Turn on *attrs* in *win.* |
| wattrset(win, attrs) | Set attributes in *win* to *attrs.* |
| wclear(win) | Clear *win.* |
| wclrtobot(win) | Clear to bottom of *win.* |
| wclrtoeol(win) | Clear to end of line on *win.* |
| wdelch(win, c) | Delete char from *win.* |
| wdeleteln(win) | Delete line from *win.* |
| werase(win) | Erase *win.* |
| wgetch(win) | Get a char through *win.* |
| wgetstr(win, str) | Get a string through *win.* |
| winch(win) | Get a char at current (*y, x*) in *win.* |
| winsch(win, c) | Insert char into *win.* |
| winsertln(win) | Insert line into *win.* |
| wmove(win, y, x) | Set current (*y, x*) co-ordinates on *win.* |
| wnoutrefresh(win) | Refresh but no screen output. |
| wprintw(win, fmt, arg1, arg2, ...) | **printf** on *win.* |
| wrefresh(win) | Make screen look like *win.* |
| wscanw(win, fmt, arg1, arg2, ...) | **scanf** through *win.* |
| wsetscrreg(win, t, b) | Set scrolling region of *win.* |
| wstandend(win) | Clear standout attribute in *win.* |
| wstandout(win) | Set standout attribute in *win.* |

# curses

## Terminfo Level Routines

The routines in Table 3-3 should be called by programs wishing to deal directly with the terminfo database. Due to the low level of this interface, it is discouraged. Initially, **setupterm** should be called. This will define the set of terminal dependent variables defined in **terminfo** (see Section 4). The include files <curses.h> and <term.h> should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm** to instantiate them. All terminfo strings (including the output of **tparm**) should be printed with **tputs** or **putp**. Begore exiting, **resetterm** should be called to restore the tty modes. (Programs desiring shell escapes or suspending with control Z can call **resetterm** before the shell is called, and **fixterm** after returning from the shell.)

Table 3-3 **Terminfo Level Routines**

| | |
|---|---|
| fixterm() | Restore tty modes for terminfo use (called by **setupterm**). |
| resetterm() | Reset tty modes to state before program entry. |
| setupterm(term, fd, rc) | Read in database. Terminal type is the character string *term*, all output is to CENTIX file descriptor *fd*. A status value is returned in the integer pointed to by *rc*: 1 is normal. The simplest call would be **setupterm(0, 1, 0)**, which uses all defaults. |
| tparm(str, p1, p2, ..., p9) | Instantiate string *str* with parms *pi*. |
| tputs(str, affcnt, putc) | Apply padding info to string *str*. *Affcnt* is the number of lines affected, or 1 if not applicable. *Putc* is a **putchar**-like function to which the characters are passed, one at a time. |
| putp(str) | Handy function that calls **tputs(str, 1, putchar)**. |
| vidputs(attrs, putc) | Output the string to put terminal in video attribute mode *attrs*, which is any combination of the attributes listed below. Chars are passed to **putchar**-like function **putc**. |
| vidattr(attrs) | Like **vidputs** but outputs through **putchar**. |

# curses

## Termcap Compatibility Routines

These routines were included as a conversion aid for programs that use termcap. Their parameters are the same as for termcap. They are emulated using the terminfo database. They may go away at a later date.

Table 3-4 **Termcap Compatibility Routines**

| | |
|---|---|
| tgetent(bp, name) | Look up termcap entry for *name.* |
| tgetflag(id) | Get boolean entry for *id.* |
| tgetnum(id) | Get numeric entry for *id.* |
| tgetstr(id, area) | Get string entry for *id.* |
| tgoto(cap, col, row) | Apply parms to given *cap.* |
| tputs(cap, affcnt, fn) | Apply padding *cap* calling *fn* as **putchar**. |

## Attributes

The video attributes in Table 3-5 can be passed to the functions **attron, attroff, attrset.**

Table 3-5 **Video Attributes**

| | |
|---|---|
| A_STANDOUT | Terminal's best highlighting mode. |
| A_UNDERLINE | Underlining. |
| A_REVERSE | Reverse video. |
| A_BLINK | Blinking. |
| A_DIM | Half bright. |
| A_BOLD | Extra bright or bold. |
| A_BLANK | Blanking (invisible). |
| A_PROTECT | Protected. |
| A_ALTCHARSET | Alternate character set. |

## curses

### Function Keys

The function keys in Table 3-6 might be returned by **getch** if **keypad** has been enabled. Note that not all of these are currently supported, due to lack of definitions in terminfo or the terminal not transmitting a unique code when the key is pressed.

Table 3-6 **Curses Function Keys**

| **Name** | **Value** | **Key Name** |
|---|---|---|
| KEY_BREAK | 0401 | Break key (unreliable). |
| KEY_DOWN | 0402 | The four arrow keys... |
| KEY_UP | 0403 | |
| KEY_LEFT | 0404 | |
| KEY_RIGHT | 0405 | |
| KEY_HOME | 0406 | Home key (upward+left arrow). |
| KEY_BACKSPACE | 0407 | Backspace (unreliable). |
| KEY_F0 | 0410 | Function keys. Space for 64 is reserved. |
| KEY_F(n) | (KEY_F0+(n)) | Formula for f*n*. |
| KEY_DL | 0510 | Delete line. |
| KEY_IL | 0511 | Insert line. |
| KEY_DC | 0512 | Delete character. |
| KEY_IC | 0513 | Insert char or enter insert mode. |
| KEY_EIC | 0514 | Exit insert char mode. |
| KEY_CLEAR | 0515 | Clear screen. |
| KEY_EOS | 0516 | Clear to end of screen. |
| KEY_EOL | 0517 | Clear to end of line. |
| KEY_SF | 0520 | Scroll 1 line forward. |
| KEY_SR | 0521 | Scroll 1 line backwards (reverse). |
| KEY_NPAGE | 0522 | Next page. |
| KEY_PPAGE | 0523 | Previous page. |
| KEY_STAB | 0524 | Set tab. |
| KEY_CTAB | 0525 | Clear tab. |
| KEY_CATAB | 0526 | Clear all tabs. |
| KEY_ENTER | 0527 | Enter or send (unreliable). |
| KEY_SRESET | 0530 | Soft (partial) reset (unreliable). |
| KEY_RESET | 0531 | Reset or hard reset (unreliable). |
| KEY_PRINT | 0532 | Print or copy. |
| KEY_ILL | 0533 | Home down or bottom left. |

## curses

### Caution

The plotting library **plot** and the curses library **curses** both use the names erase() and move(). The curses versions are macros. If you need both libraries, put the **plot** code in a different source file than the **curses** code, and/or #undef move() and erase() in the **plot** code.

### See Also

**terminfo** in Section 4; *XE 500 CENTIX System Programming Guide*.

## cuserid

### Name

**cuserid** - get character login name of the user

### Format

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

### Description

The **cuserid** function gets the user's login name as found in /etc/utmp. If the login name cannot be found, **cuserid** gets the login name corresponding to the user ID of the process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least L_cuserid characters; the representation is left in this array. The constant L_cuserid is defined in the <stdio.h> header file.

### Diagnostics

If the login name cannot be found and the process's owner lacks a password file entry, **cuserid** returns a NULL pointer; if *s* is not a NULL pointer, a NULL character (\0) will be placed at *s*[0].

### See Also

**getlogin**, **getpwent**.

# dial

## Name

**dial, undial** - establish and release an out-going terminal line connection.

## Format

```
#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;
```

## Description

The **dial** library function 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 telno digits
                                     string*/
        int           modem;     /*specify modem control for
                                     direct lines*/
        char          *device;   /*will hold the name of the
                                     device used to make
                                     a connection*/
        int           dev_len;   /*the length of the device
                                     used to make
                                     connection*/
} CALL;
```

## dial

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 L-devices file. In this case, the value of the *baud* element need not be specified as it will be determined from the L-devices file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. The number must consist of the following codes:

| | |
|---|---|
| **0-9** | Dial 0-9. |
| * or : | Dial *. |
| # or ; | Dial #. |
| - | 4 second delay for second dial tone. |
| w or = | Wait for secondary dial tone. |
| f | Flash off hook for 1 second. |

On a smart modem, these symbols are translated to modem commands using the modem description in /usr/lib/uucp/modemcap.

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 will be set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

# dial

The CALL element *device* is used to hold the device name (cul..) that establishes the connection.

The CALL element *dev len* is the length of the device name that is copied into the array device.

## Files

/usr/lib/uucp/modemcap
/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK *tty-device*

## Diagnostics

On failure, a negative value indicating the reason for the failure will be 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    /*cannot open LDEVS 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 request baud*/
NO_BD_K    -11   /*no device known at request baud*/
```

## Cautions

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.

## See Also

**uucp** in Section 1; **alarm, read, write** in Section 2; **modemcap** in Section 5; **termio** in Section 6.

# drand48

## Name

**drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48** - generate uniformly distributed pseudo-random numbers

## Format

```
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 library functions generate pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

The **drand48** and **erand48** functions return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

**lrand48** and **nrand48** return non-negative long integers uniformly distributed over the interval [0, $2^{31}$).

# drand48

**mrand48** and **jrand48** return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

The **srand48**, **seed48**, and **lcong48** functions 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

$$X_{n+1} = (aX_n + c)_{\text{mod} m} \text{ where } n >= 0$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless **lcong48** has been invoked, the multiplier value $a$ and the addend value $c$ are given by

$$a = 5DEECE66D_{16} = 273673163155_8$$
$$c = B_{16} = 13_8$$

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 (leftmost) 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; that is why they must be initialized prior to being invoked. The functions **erand48**, **nrand48**, and **jrand48** require the calling program to provide storage for the successive $X_i$ values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of $X_i$ 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, that is, 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.

## drand48

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 be ignored if it is not needed, is useful when restarting a program from a given point at some future time. For example, use the pointer to get 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 element *param[0-2]* specifies $X_i$, *param[3-5]* specifies 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 previously.

### See Also

**rand.**

# ecvt

## Name

**ecvt**, **fcvt**, **gcvt** - convert floating-point number to string

## Format

```
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

The **ecvt** function 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 return 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*.

The **gcvt** function 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.

# ecvt

## Known Problems

The values returned by **ecvt** and **fcvt** point to a single static data array whose content is overwritten by each call.

## See Also

**printf**.

# end

## Name

**end**, **etext**, **edata** - last locations in programs

## Format

```
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** (see Section 2), **malloc**, standard input/output (**stdio**), the profile (**-p**) option of **cc** (see Section 1), and so on. Thus, the current value of the program break should be determined by **sbrk(0)** (see **brk** in Section 2).

## See Also

**brk** in Section 2; **malloc**.

# erf

## Name

**erf**, **erfc** - error function and complementary error function

## Format

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

## Description

The **erf** function 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 (for example, for *x* - 5, 12 places are lost).

## See Also

**exp**.

# exp

## Name

**exp, log, log10, pow, sqrt** - exponential, logarithm, power, square root functions

## Format

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

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

double sqrt (x)
double x;
```

## Description

**exp** returns the exponential function ($e_x$).

**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.

## exp

### 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 +/-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**.

### See Also

**hypot, matherr, sinh.**

# fclose

## Name

**fclose, fflush** - close or flush a stream

## Format

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

## Description

The **fclose** function causes any buffered data for the named *stream* to be written out, and the *stream* to be closed. **fclose** is performed automatically for all open files upon calling the **exit** system call.

The **fflush** function causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

## Diagnostics

These functions return 0 for success, and EOF if any error (such as trying to write to a file that has not been opened for writing) is detected.

## See Also

**close, exit** in Section 2; **fopen, setbuf.**

## ferror

### Name

**ferror, feof, clearerr, fileno** - stream status inquiries

### Format

```
#include <stdio.h>

int ferror (stream)
FILE *stream;

int feof (stream)
FILE *stream;

void clearerr (stream)
FILE stream;

int fileno (stream)
FILE *stream;
```

### Description

The **ferror** function returns non-zero when I/O error has previously occurred reading from or writing to the named *stream*; otherwise, it returns zero.

The **feof** function returns non-zero when EOF has previously been detected reading the named input *stream*; otherwise, it returns zero.

**clearerr** resets the error indicator and EOF indicator to zero on the named *stream*.

**fileno** returns the integer file descriptor associated with the named *stream*; see **open** in Section 2.

All of these functions are implemented as macros; they cannot be declared or redeclared.

### See Also

**open** in Section 2; **fopen**

# floor

## Name

**floor, ceil, fmod, fabs** - floor, ceiling, remainder, absolute value functions

## Format

```
#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

The **floor** function 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*: zero 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**.

## fopen

### Name

**fopen, freopen, fdopen** - open a stream

### Format

```
#include <stdio.h>

FILE *fopen (file-name, type)
char *file-name, *type;

FILE *freopen (file-name, type, stream)
char *file-name, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char type;
```

### Description

The **fopen** function opens the file named by *file-name* and associates a stream with it. **fopen** returns a pointer to the FILE structure associated with the *stream*.

*File-name* points to a character string that contains the name of the file to be opened.

*Type* is a character string having one of the following values:

| | |
|---|---|
| "r" | Open for reading. |
| "w" | Truncate or create for writing. |
| "a" | Append; open for writing at end of file, or create for writing. |
| "r+" | Open for update (reading and writing). |
| "w+" | Truncate or create for update. |
| "a+" | Append; open or create for update at end-of-file. |

The **freopen** function substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether open ultimately succeeds. **freopen** returns a pointer to the FILE structure associated with *stream*.

# fopen

The **fdopen** function associates a *stream* with a file descriptor obtained from **open**, **dup**, **creat**, or **pipe** system calls, which will open files but not return pointers to a FILE structure *stream*, which is necessary input for many of the standard I/O library functions. The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening **fseek** or **rewind**, and input may not be directly followed by output without an intervening **fseek**, **rewind**, or an inpur operation that encounters end-of-file.

When a file is opened for append (that is, when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. **fseek** may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written to by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

## Diagnostics

**fopen** and **freopen** return a NULL pointer on failure.

## See Also

**open** in Section 2; **fclose**.

# fread

## Name

**fread, fwrite** - binary input/output

## Format

```
#include <stdio.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

## Description

The **fread** function copies, into an array pointed to by *ptr, nitems* of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. **fread** stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* have been read. **fread** leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read, if there is one.

The **fwrite** function appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. **fwrite** stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. **fwrite** does not change the contents of the array pointed to by *ptr*.

The argument *size* is typically **sizeof(*ptr)**, where the pseudo-function **sizeof** specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

# fread

## Diagnostics

**fread** and **fwrite** return the number of items read or written. If *size* or *nitems* is non-positive, no characters are read or written and 0 is returned by both **fread** and **fwrite**.

## See Also

**read, write** in Section 2; **fopen, getc, gets, printf, putc, puts, scanf.**

# frexp

## Name

**frexp, ldexp, modf** - manipulate parts of floating-point numbers

## Format

```
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 $0.5 =< |x| < 1.0$, and the "exponent" $n$ is an integer. The **frexp** function 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.

The **ldexp** function returns the quantity $value * 2^{exp}$.

The **modf** function returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

## Diagnostics

If **ldexp** would cause overflow, +/-HUGE is returned (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.

## fseek

### Name

**fseek**, **rewind**, **ftell** - reposition a file pointer in a stream

### Format

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

### Description

The **fseek** function sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the bginning, from the current position, or from the end of the file, according as *ptrname* has the value 0, 1, or 2.

**rewind**(*stream*) is equivalent to **fseek**(*stream*, 0L, 0), except that no value is returned.

**fseek** and **rewind** undo any effects of the **ungetc** function.

After **fseek** or **rewind**, the next operation on a file opened for update may be either input or output.

The **ftell** function returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

# fseek

## Diagnostics

**fseek** returns non-zero for improper seeks; otherwise, zero. An improper seek can be, for example, an **fseek** done on a file that has not been opened via **fopen**; in particular, **fseek** may not be used on a terminal, or on a file opened via **popen**.

## Caution

On CENTIX, the value returned by **ftell** is a number of bytes, and a program can use this value to seek relative to the current offset. Such programs are not portable to systems where file offsets are not measured in bytes.

## See Also

**lseek** in Section 2; **fopen**.

# ftw

## Name

**ftw** - walk a file tree

## Format

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn)();
int depth;
```

## Description

The **ftw** function 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** in Section 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, that **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) permissions.

The **ftw** function visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of **fn** returns a non-zero value, or some error is detected within **ftw** (such as an I/O error). If the tree is exhausted, **ftw** returns zero. If **fn** returns a non-zero 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

**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 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.

### Known Problems

Because **ftw** is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

It could be made to run faster and use less storage on deep structures at the cost of considerable complexity.

**ftw** uses the **malloc** function to allocate dynamic storage during its operation. If **ftw** is forcibly terminated, such as by **longjmp** being executed by **fn** or an attempted 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 non-zero value at its next invocation.

### See Also

**stat** in Section 2; **malloc**.

# gamma

## Name

**gamma** - log gamma function

## Format

```
#include <math.h>

double gamma (x)
double x;

extern int signgam;
```

## Description

The **gamma** function 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 $\Gamma$ :

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
        error();
y = signgam * exp(y);
```

where LN_MAXDOUBLE is the least value that causes the **exp** function to return a range error, and is defined in the <values.h> header file.

## Diagnostics

For non-negative integer arguments, HUGE is returned and **errno** is set to EDOM. A message indicating SIGN error is printed on the standard error output.

If the correct value would overflow, **gamma** returns HUGE and sets **errno** to ERANGE.

## gamma

These error-handling procedures may be changed with the function **matherr**.

### See Also

**exp**, **matherr**; **values** in Section 5.

## getc

### Name

**getc**, **getchar**, **fgetc**, **getw** - get character or word from a stream

### Format

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

### Description

The **getc** function returns the next character (or byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. The **getchar** function is defined as **getc**(*stdin*). **getc** and **getchar** are macros.

**fgetc** behaves like **getc** but is a genuine function. **fgetc** runs more slowly than **getc**, but it takes less space per invocation and its name can be passed as an argument to a function.

**getw** returns the next word (integer) from the named input *stream*. **getw** increments the associated file pointer, if defined, to point to the next word. The size of the word is the size of an integer and varies from machine to machine. **getw** assumes no special alignment in the file.

### Diagnostics

These functions return the constant EOF at end-of-file or upon an error. Because EOF is a valid integer, the **ferror** function should be used to detect **getw** errors.

## getc

### Caution

If the integer value returned by **getc**, **getchar**, or **fgetc** is stored into a character variable and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a character on widening to an integer is machine-dependent.

### Known Problems

Because it is implemented as a macro, **getc** incorrectly treats a *stream* argument with side effects. In particular, **getc(*f++)** does not work sensibly. **fgetc** should be used instead. Because of possible differences in word length and byte ordering, files written using the **putw** function are machine-dependent, and may not be read using **getw** on a different processor.

### See Also

**fclose**, **ferror**, **fopen**, **fread**, **gets**, **putc**, **scanf**.

# getcwd

## Name

**getcwd** - get the path-name of the current working directory

## Format

```
char *getcwd (buf, size)
char *buf;
int size;
```

## Description

The **getcwd** function 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 the **malloc** function. 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** to pipe the output of the **pwd** shell command into the specified string space.

## Example

```
char *cwd, getcwd();
.
.
.
if ((cwd = getcwd((char*)NULL, 64)) == NULL) {
      perror("pwd");
      exit(1);
}
printf("%s\n",cwd);
```

## Diagnostics

Returns NULL with **errno** set if *size* is not large enough, or if an error occurs in a lower-level function.

## See Also

**pwd** in Section 1; **malloc**, **popen**.

# getenv

## Name

**getenv** - return value for environment name

## Format

```
char *getenv (name)
char *name;
```

## Description

The **getenv** function searches the environment list (see **environ** in Section 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** in Section 2; **putenv**; **environ** in Section 5.

# getgrent

## Name

**getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent** - get group file entry

## Format

```
#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

The **getgrent, getgrgid,** and **getgrname** functions each return pointers to objects 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;        /* name of the group */
        char     *gr_passwd;      /* encrypted group passwd */
        int      gr_gid;          /* numerical group ID */
        char     **gr_mem;        /* vector of pointers to
                                     member names */
};
```

When first called, **getgrent** returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. **getgrgid** searches from the beginning

## getgrent

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 a returns a pointer to a 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.

The **fgetgrent** function returns a pointer to the next group structure in the stream *f*, which matches the format of /etc/group.

### Files

/etc/group

### Diagnostics

A NULL pointer is returned on EOF or error.

### Caution

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.

### Known Problems

All information is contained in a static area, so it must be copied if it is to be saved.

### See Also

**getlogin**, **getpwent**; **group** in Section 4.

# getlogin

## Name

**getlogin** - get login name

## Format

```
char *getlogin ();
```

## Description

The **getlogin** function 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, call **getpwuid**.

## Files

/etc/utmp

## Diagnostics

Returns the NULL pointer if *name* is not found.

## Known Problems

The return values point to static data whose content is overwritten by each call.

## See Also

**cuserid**, **getgrent**, **getpwent**; **utmp** in Section 4.

# getopt

## Name

**getopt** - get option letter from argument vector

## Format

```
int getopt (argc, argv, optstring)
int argc;
char **argv, optstring;

extern char *optarg;
extern int optind, opterr;
```

## Description

The **getopt** function returns the next option letter in *argv* that matches a letter in *optstring*. *Opstring* is a string of recognized option letters. If a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *Optarg* is set to point to the start of the option argument on return from **getopt**.

The **getopt** function places in *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to zero automatically before the first call to **getopt**.

When all options have been processed (that is, up to the first non-option arguments), **getopt** returns EOF. The special option - may be used to delimit the end of the options. EOF will be returned and - will be skipped.

## Diagnostics

**getopt** prints an error message on the stderr file and returns a question mark ('?') when it encounters an option letter not included in *optstring*. This error message may be disabled by setting *opterr* to a non-zero value.

## getopt

### Example

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern char *optarg;
        extern int optind;
        .
        .
        .
        while ((c = getopt(argc, argv, "abf:o:")) != EOF)
              switch (c) {
              case 'a' :
                    if (bflg)
                          errflg++;
                    else
                          aflg++;
                    break;
              case 'b' :
                    if (aflg)
                          errflg++;
                    else
                          bproc();
                    break;
              case 'f' :
                    ifile = optarg;
                    break;
              case 'o' :
                    ofile = optarg;
                    break;
              case '?' :
                    errflg++;
              }
        if (errflg) {
              fprintf(stderr, "usage:    ");
              exit (2);
        }
        for ( ; optind < argc; optind++) {
              if (access(argv[optind], 4)) {
        .
        .
        .
}
```

## getopt

### See Also

**getopt** in Section 1.

# getpass

## Name

**getpass** - read a password

## Format

```
char *getpass (prompt)
char *prompt;
```

## Description

The **getpass** function reads up to a new-line or EOF from the file /dev/tty, after prompting 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

## Caution

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.

## Known Problems

The return value points to static data whose content is overwritten by each call.

## See Also

**crypt**.

## getpw

### Name

**getpw** - get name from UID

### Format

```
int getpw (uid, buf)
int uid;
char *buf;
```

### Description

The **getpw** function 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 ponted to by *buf*, and returns 0. **getpw** returns non-zero if *uid* cannot be found.

This routine is included only for compatibility with older systems and should not be used; see **getpwent** for routines to use instead.

### Files

/etc/passwd

### Diagnostics

**getpw** returns non-zero on error.

### Caution

The above routine uses <stdio>, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

### See Also

**getpwent**; **passwd** in Section 4.

# getpwent

## Name

**getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent** - get password file entry

## Format

```
#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

The **getpwent, getpwuid**, and **getpwnam** functions return 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.

## getpwent

The *pw_comment* field is unused; the others have meanings described in **passwd** (see Section 4).

The **getpwent** function, 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, then 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, then returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to **setpwent** has the effect of rewinding the password file to allow repeated searches. **endpwent** may be called to close the password file when processing is complete.

The **fgetpwent** function returns a pointer to the next passwd structure in the stream *f*, which matches the format of /etc/passwd.

### Files

/etc/passwd

### Diagnostics

A NULL pointer is returned on EOF or error.

### Caution

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.

## getpwent

### Known Problems

All information is contained in a static area, so it must be copied if it is to be saved.

### See Also

**getlogin**, **getgrent**; **passwd** in Section 4.

# gets

## Name

**gets, fgets** - get a string from a stream

## Format

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

## Description

The **gets** function reads characters from the standard input stream, *stdin*, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated by a null character.

The **fgets** function reads characters from *stream* into the array pointed to by *s*, until *n*-1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a NULL character.

## Diagnostics

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise, *s* is returned.

## See Also

**ferror, fopen, fread, getc, scanf.**

# getut

## Name

**getutent, getutid, getutline, pututline, setutent, endutent, utmpname** - access utmp file entry

## Format

```
#include <utmp.h>

struct utmp *getutent ()

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getuline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ()

void endutent ()

void utmpname (file)
char *file;
```

# getut

## Description

The **getutent**, **getutid**, and **getutline** functions 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 number)*/
        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 term. 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*/

};
```

The **getutent** function reads in the next entry from a utmp-like file. If the file is not already open, **getutent** 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** returns 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 the file is reached without a match, it fails.

The **getutline** function 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 the file is reached without a match, it fails.

## getut

**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. Do this before each search for a new entry if the entire file must 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. Most often, this file will be /etc/utmp. If the file does not exist, this will be apparent after the first attempt to reference it, not on the **utmpname** call. This function does not open the file, it just closes the old utmp file, if currently open, and saves the new file name.

### Files

/etc/utmp, /etc/wtmp

### Diagnostics

A NULL pointer is returned upon failure to read, whether due to permissions or to having reached the end of the file, or upon failure to write.

# getut

## Known Problems

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 is 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 unbeffered non-standard write to avoid race conditions between processes trying to modify the utmp and wtmp files.

## See Also

**ttyslot**; **utmp** in Section 4.

# hsearch

## Name

**hsearch, hcreate, hdestroy** - manage hash search tables

## Format

```
#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ()
```

## Description

The **hsearch** function is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer to 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 associated with that key. (Pointers to types other than character should be cast to type 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 the appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

The **hcreate** function 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.

## hsearch

The **hdestroy** function destroys the search table, freeing the memory used by the table. It may be followed by another call to **hcreate**.

**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 **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 manner similar to **strcmp** (see **string**). |
| **CHAINED** | Use a linked list to resolve collisions. If this option is selected, the following options become available. |

| | |
|---|---|
| **START** | Place new entries at the beginning of the linked list (default is at the end). |
| **SORTUP** | Keep the linked list sorted by key in ascending order. |
| **SORTDOWN** | Keep the linked list sorted by key in descending order. |

Additionally, there are preprocessor flags for obtaining debugging printout (**-DDEBUG**) and for including a test driver in the calling routine (**-DDRIVER**). The source code should be consulted for further details.

### Example

The following example will read in strings followed by two numbers and store them in a hash table, discarding the duplicates. It will then read in strings and find the matching entry in the hash table, then print it out.

## hsearch

```
#include <stdio.h>
#include <search.h>

struct info {              /*this is the info stored in 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 available space in string_space*/
        char *str_ptr = string_space;
        /*next available 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)
              }
        }
}
```

# hsearch

## 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.

## Caution

**hsearch** and **hcreate** use the **malloc** function to allocate space.

## Known Problems

Only one hash search table may be active at any given time.

## See Also

**bsearch, lsearch, malloc, string, tsearch.**

## hypot

### Name

**hypot** - Euclidean distance function

### Format

```
#include <math.h>

double hypot (x, y)
double x, y;
```

### Description

The **hypot** function returns

sqrt(x * x + y * y),

taking precautions against unwarranted overflows.

### Diagnostics

When the correct value would overflow, **hypot** returns HUGE and sets **errno** to ERANGE.

These error-handling procedures may be changed with the **matherr** function.

### See Also

**matherr**, **exp**.

# l3tol

## Name

**l3tol, ltol3** - convert between 3-byte integers and long integers

## Format

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

## Description

The **l3tol** function 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*.

The **ltol3** function 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.

## Known Problems

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

## See Also

**fs** in Section 4.

# ldahread

## Name

**ldahread** - read the archive header of a member of an archive file

## Format

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

## Description

If TYPE(*ldptr*) is the archive file magic number, the **ldahread** function reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

**ldahread** returns SUCCESS or FAILURE. The function will fail if TYPE(*ldptr*) does not represent an archive file, or if it cannot read the archive header.

The program must be loaded with the object file access routine library libld.a.

## Files

/usr/lib/libld.a

## See Also

**ldclose**, **ldopen**; **ldfcn**, **ar** in Section 4.

# ldclose

## Name

**ldclose, ldaclose** - close a common object file

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

## Description

The **ldopen** and **ldclose** functions are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If TYPE(*ldptr*) does not represent an archive file, **ldclose** will close the file and free the memory allocated to the LDFILE structure associated with *ldptr*. If TYPE(*ldptr*) is the magic number of an archive file, and if there are any more files in the archive, **ldclose** will reinitialize OFFSET(*ldptr*) to the file address of the next archive member and return FAILURE. The LDFILE structure is prepared for a subsequent **ldopen**. In all other cases, **ldclose** returns SUCCESS.

**ldaclose** closes the file and frees the memory allocated to the LDFILE structure associated with *ldptr* regardless of the value of TYPE(*ldptr*). **ldaclose** always returns SUCCESS. The function is often used in conjunction with **ldaopen**.

The program must be loaded with the object file access routine library libld.a.

# ldclose

## Files

/usr/lib/libld.a

## See Also

**fclose**, **ldopen**; **ldfcn** in Section 4.

# ldfhread

## Name

**ldfhread** - read the file header of a common object file

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

## Description

The **ldfhread** function reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

**ldfhread** returns SUCCESS or FAILURE. The function will fail if it cannot read the file header.

In most cases, the use of **ldfhread** can be avoided by using the macro HEADER(*ldptr*), defined in ldfcn.h (see **ldfcn** in Section 4). The information in any field, *fieldname*, of the file header may be accessed using HEADER(*ldptr*).*fieldname*.

The program must be loaded with the object file access routine library libld.a.

## Files

/usr/lib/libld.a

## See Also

**ldclose, ldopen; ldfcn** in Section 4.

# ldgetname

## Name

**ldgetname** - retrieve symbol name for common object file symbol table entry

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;
```

## Description

The **ldgetname** function returns a pointer to the name associated with *symbol* as a string. The string is contained in a static buffer local to **ldgetname** that is overwritten by each call to **ldgetname**, and therefore must be copied by the caller if the name is to be saved.

As of UNIX System release 5.0, which corresponds with the first release of CENTIX, the common object file format has been extended to handle arbitrary length symbol names with the addition of a "string table." **ldgetname** will return the symbol name associated with a symbol table entry for either a pre-UNIX System 5.0 object file or a UNIX System 5.0 object file. Thus, **ldgetname** can be used to retrieve names from object files without any backward compatibility problems. **ldgetname** will return NULL (defined in **stdio.h**) for a UNIX System 5.0 object file if the name cannot be retrieved. This situation can occur:

- if the "string table" cannot be found,
- if not enough memory can be allocated for the string table,

## ldgetname

- if the string table appears not to be a string table (for example, if an auxiliary entry is handed to **ldgetname** that looks like a reference to a name in a non-existent string table, or
- if the name's offset into the string table is past the end of the string table.

Typically, **ldgetname** will be called immediately after a successful call to **ldtbread** to retrieve the name associated with the symbol table entry filled by **ldtbread**.

The program must be loaded with the object file access routine library libld.a.

### Files

/usr/lib/libld.a

### See Also

**ldclose**, **ldopen**, **ldtbread**, **ldtbseek**; **ldfcn** in Section 4.

# ldlread

## Name

**ldlread, ldlinit, ldlitem** - manipulate line number entries of a common object file function

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>

int ldlread (ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO linent;

int ldlinit (ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem (ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO linent;
```

## Description

The **ldlread** function searches the line number entries of the common object file currently associated with *ldptr*. **ldlread** begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. **ldlread** reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

# ldlread

**ldlinit** and **ldlitem** together perform exactly the same function as **ldlread**. After an initial call to **ldlread** or **ldlinit**, **ldlitem** may be used to retrieve a series of line number entries associated with a single function. **ldlinit** simply locates the line number entries for the function identified by *fcnindx*. **ldlitem** finds and reads the entry with the smallest line number equal to or greater than *linenum* to *linent*.

**ldlread**, **ldlinit**, and **ldlitem** each return either SUCCESS or FAILURE. **ldlread** will fail if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. **ldlinit** will fail if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. **ldlitem** will fail if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library libld.a.

## Files

/usr/lib/libld.a

## See Also

**ldclose**, **ldopen**, **ldtbindex**; **ldfcn** in Section 4.

# ldlseek

## Name

**ldlseek, ldnlseek** - seek to line number entries of a section of a common object file

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

## Description

The **ldlseek** function seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr.*

**ldnlseek** seeks to the line number entries of the section specified by *sectname*.

**ldlseek** and **ldnlseek** return SUCCESS or FAILURE. **ldlseek** will fail if *sectindx* is greater than the number of sections in the object file; **ldnlseek** will fail if there is no section name corresponding with **sectname*. Either function will fail if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library libld.a.

# ldlseek

## Files

/usr/lib/libld.a

## See Also

**ldclose, ldopen, ldshread**; **ldfcn** in Section 4.

# ldohseek

## Name

**ldohseek** - seek to the optional file header of a common object file

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

## Description

The **ldohseek** function seeks to the optional file header of the common object file currently associated with *ldptr.*

**ldohseek** returns SUCCESS or FAILURE. It will fail if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library libld.a.

## Files

/usr/lib/libld.a

## See Also

**ldclose, ldopen, ldfhread; ldfcn** in Section 4.

## ldopen

### Name

**ldopen, ldaopen** - open a common object file for reading

### Format

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

### Description

The **ldopen** and **ldclose** functions are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.

If *ldptr* has the value NULL, then **ldopen** will open *filename* and allocate and initialize the LDFILE structure, and return a pointer to the structure to the calling program.

If *ldptr* is valid and if TYPE(*ldptr*) is the archive magic number, **ldopen** will reinitialize the LDFILE structure for the next archive member of *filename*.

**ldopen** and **ldclose** are designed to work together. **ldclose** will return FAILURE only when TYPE(*ldptr*) is the archive magic number and there is another file in the archive to be processed. Only then should **ldopen** be called with the current value of *ldptr*. In all other cases, in particular whenever a new *filename* is opened, **ldopen** should be called with a NULL *ldptr* argument.

## ldopen

The following is a prototype for the use of **ldopen** and **ldclose**.

```
/*for each filename to be processed*/

ldptr = NULL;
do
{
        if ((ldptr = ldopen(filename, ldptr)) != NULL)
        {
                /*check magic number*/
                /*process the file*/
        }
} while (ldclose(ldptr) == FAILURE);
```

If the value of *oldptr* is not NULL, **ldaopen** will open a new *filename* and allocate and initialize a new LDFILE structure, copying the TYPE, OFFSET, and HEADER fields from *oldptr*. **ldaopen** returns a pointer to the new LDFILE structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both **ldopen** and **ldaopen** open *filename* for reading. Both functions return NULL if *filename* cannot be opened, or if memory for the LDFILE structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

The program must be loaded with the object file access routine library libld.a.

## Files

/usr/lib/libld.a

## See Also

**fopen**, **ldclose**; **ldfcn** in Section 4.

# ldrseek

## Name

**ldrseek, ldnrseek** - seek to relocation entries of a section of a common object file

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

## Description

The **ldrseek** function seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

**ldnrseek** seeks to the relocation entries of the section specified by *sectname*.

**ldrseek** and **ldnrseek** return SUCCESS or FAILURE. **ldrseek** will fail if *sectindx* is greater than the number of sections in the object file; **ldnrseek** will fail if there is no section name corresponding with *sectname*. Either function will fail if the specified section has no relocation entries or if it cannot seek to the specific relocation entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library libld.a.

# ldrseek

## Files

/usr/lib/libld.a

## See Also

**ldclose, ldopen, ldshread**; **ldfcn** in Section 4.

# ldshread

## Name

**ldshread, ldnshread** - read an indexed/named section header of a common object file

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;
```

## Description

The **ldshread** function reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead.*

**ldnshread** reads the section header specified by *sectname* into the area of memory beginning at *secthead.*

**ldshread** and **ldnshread** return SUCCESS or FAILURE. **ldshread** will fail if *sectindx* is greater than the number of sections in the object file; **ldnshread** will fail if there is no section name corresponding with *sectname*. Either function will fail if it cannot read the specified section header.

Note that the first section header has an index of *one*.

The program must be loaded with the object file access routine library libld.a.

# ldshread

## Files

/usr/lib/libld.a

## See Also

**ldclose, ldopen; ldfcn** in Section 4.

# ldsseek

## Name

**ldsseek, ldnsseek** - seek to an indexed/named section of a common object file

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

## Description

The **ldsseek** function seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

The **ldnsseek** function seeks to the section specified by *sectname*.

**ldsseek** and **ldnsseek** return SUCCESS or FAILURE. **ldsseek** will fail if *sectindx* is greater than the number of sections in the object file; **ldnsseek** will fail if there is no section name corresponding with *sectname*. Either function will fail if there is no section data for the specified section or if it cannot seek to the specified section.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library libld.a.

# ldsseek

## Files

/usr/lib/libld.a

## See Also

**ldclose, ldopen, ldshread; ldfcn** in Section 4.

# ldtbindex

## Name

**ldtbindex** - compute the index of a symbol table entry of a common object file

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

## Description

The **ldtbindex** function returns the (long) index of the symbol table entry at the current position of the common object file associated with *ldptr.*

The index returned by **ldtbindex** may be used in subsequent calls to the **ldtbread** function. However, since **ldtbindex** returns the index of the symbol table entry that begins at the current position of the object file, if **ldtbindex** is called immediately after a perticular symbol table entry has been read, it will return the index of the next entry.

**ldtbindex** will fail if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library libld.a.

# ldtbindex

## Files

/usr/lib/libld.a

## See Also

**ldclose**, **ldopen**, **ldtbread**, **ldtbseek**; **ldfcn** in Section 4.

# ldtbread

## Name

**ldtbread** - read an indexed symbol table entry of a common object file

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

## Description

The **ldtbread** function reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at *symbol.*

**ldtbread** returns SUCCESS or FAILURE. It will fail if *symindex* is greater than the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library libld.a.

## Files

/usr/lib/libld.a

## See Also

**ldclose, ldopen, ldtbseek; ldfcn** in Section 4.

# ldtbseek

## Name

**ldtbseek** - seek to the symbol table of a common object file

## Format

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

## Description

The **ldtbseek** function seeks to the symbol table of the object file currently associated with *ldptr*.

**ldtbseek** returns SUCCESS or FAILURE. It will fail if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library libld.a.

## Files

/usr/lib/libld.a

## See Also

**ldclose, ldopen, ldtbread; ldfcn** in Section 4.

## lockf

### Name

**lockf** - record locking on files

### Format

```
#include <unistd.h>

lockf (fildes, function, size)
long size;
int fildes, function;
```

### Description

The **lockf** function will allow sections of a file to be locked (advisory write locks). (Mandatory or enforcement mode record locks are not currently available.) Locking calls from other processes that 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 a process terminates. (See **fcntl** in Section 2 for more information about record locking.)

*Fildes* is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission on order to establish lock with this function call.

*Function* is a control value that specifies the action to be taken. The permissible values for *function* are defined in <unistd.h> as follows:

```
#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.

## lockf

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_UNLOCK 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. If *size* is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked, as such locks may exist past the end-of-file.

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 [EACCESS] 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 **lock** 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.

## lockf

Sleeping on a resource is interrupted with any signal. The **alarm** system call may be used to provide a timeout facility in applications that require this facility.

The **lockf** utility fails if one or more of the following are true:

| | |
|---|---|
| [EBADF] | *Fildes* is not a valid open descriptor. |
| [EACCESS] | *Cmd* is F_TLOCK or F_TEST and the section is already locked by another process. |
| [EDEADLK] | *Cmd* is F_LOCK or F_TLOCK and a deadlock would occur. |
| [ENOLCK] | The *cmd* is F_LOCK, F_TLOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system. (Note that this differs from EDEADLOCK.) |

### Returns

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

### Cautions

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data that is or was locked. The standard I/O package is the most common source of unexpected buffering.

### See Also

**close, creat, fcntl, intro, open, read, write** in Section 2.

# logname

## Name

**logname** - return login name of user

## Format

```
char *logname ()
```

## Description

The **logname** function returns a pointer to the null-terminated login name; it extracts the $LOGNAME variable from the user's environment.

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

## Files

/etc/profile

## Known Problems

The return values point to static data whose content is overwritten by each call.

This method of determining a login name is subject to forgery.

## See Also

**env, login** in Section 1; **profile** in Section 4; **environ** in Section 5.

# lsearch

## Name

**lsearch, lfind** - linear search and update

## Format

```
#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

The **lsearch** function is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer to a table indicating where a datum can be found. If the datum is not found, it is added to 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 if the comparison function that 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; 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.

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.

# lsearch

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 =< TABSIZE strings of length =< ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.h>

#define TABSIZE 50
#define ELSIZE 120

        char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch();
        unsigned nel = 0;
        int strcmp();
        . . .
        while (fgets(line, ELSIZE, stdin) != NULL &&
           nel < TABSIZE)
              (void) lsearch(line, (char *)tab, &nel,
                 ELSIZE, strcmp);
        . . .
```

## 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.

## Known Problems

Undefined results can occur if there is not enough room in the table to add a new item.

## See Also

**bsearch**, **hsearch**, **tsearch**.

# malloc (fast version)

## Name

**malloc, free, realloc, calloc, mallopt, mallinfo** - fast main memory allocator

## Format

```
#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 (max)
int max;
```

## Description

The **malloc** and **free** functions provide a simple general-purpose memory allocation package, which runs considerably faster than the slower **malloc** package. It is found in the library "malloc," and is loaded if the option **-lmalloc** is used with **cc** or **ld** (see Section 1).

**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 are destroyed (see **mallopt**, below, for a way to change this behavior).

## malloc (fast version)

Undefined results will occur if the space assigned by **malloc** is overrun, or if some random number is handed to **free**.

The **realloc** function changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

**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 and then doles them out very quickly. The default value for *maxfast* is 0. |
| 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 to the nearest multiple of *grain*. *Grain* must be greater than 0. The default value of *grain* is the smallest number of bytes that will allow alignment of any data type. *Value* will be 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.

## malloc (fast version)

**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 in penalty if keep option*/
                            /*is used*/
}
```

This structure is defined in the <malloc.h> header file.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

### Diagnostics

**malloc**, **realloc**, and **calloc** return a NULL pointer if there is 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, non-zero is returned. Otherwise, it returns zero.

### Cautions

This package usually uses more data space than the slower version of **malloc**.

The code size is also bigger than the slower **malloc**.

Note that unlike the slower version of **malloc**, this package does not preserve the contents of a block when it is freed, unless the M_KEEP option of **mallopt** is used.

Undocumented features of the slower **malloc** have not been duplicated.

## malloc (fast version)

### See Also

**brk** in Section 2; **malloc**

# malloc

## Name

**malloc, free, realloc, calloc** - main memory allocator

## Format

```
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;
```

## Description

The **malloc** and **free** functions 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, but its contents are left undisturbed.

Undefined results will occur if the space assigned by **malloc** is overrun or if some random number is handed to **free**.

**malloc** allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls **sbrk** (see Section 2) to get more memory from the system when there is no suitable space already free.

## malloc

**realloc** changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, then **realloc** will ask **malloc** to enlarge the arena by *size* bytes and will then move the data to the new space.

**realloc** also works if *ptr* points to a block freed since the last call of **malloc, realloc,** or **calloc**; thus sequences of **free, malloc,** and **realloc** can exploit the search strategy of **malloc** to do storage compaction.

**calloc** allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

Note that search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer. For an alternate, more flexible implementation, see the description for the fast version of **malloc**.

## Diagnostics

**malloc, realloc,** and **calloc** return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens, the block pointed to by *ptr* may be destroyed.

## See Also

**brk** in Section 2; **malloc** (fast version).

## matherr

### Name

**matherr** - error-handling function

### Format

```
#include <math.h>

int matherr (x)
struct exception *x;
```

### Description

The **matherr** function 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.

## matherr

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 Table 3-7, below. In every case, **errno** is set to EDOM or ERANGE and the program continues.

Table 3-7 **Default Error Handling Procedures**

| | ***Types of Errors*** | | | | | |
|---|---|---|---|---|---|---|
| **type** | **DOMAIN** | **SING** | **OVERFLOW** | **UNDERFLOW** | **TLOSS** | **PLOSS** |
| ***errno*** | **EDOM** | **EDOM** | **ERANGE** | **ERANGE** | **ERANGE** | **ERANGE** |
| BESSEL: y0, y1, yn | - | - | - | - | M,0 | * |
| (arg -< 0) | M,-H | - | - | - | - | - |
| EXP: | - | - | H | 0 | - | - |
| LOG,LOG10: | | | | | | |
| (arg < 0) | M,-H | - M,-H | - | - | - | - |
| (arg = 0) | - | | - | - | - | - |
| POW: | | | | | | |
| neg**non-int | - | - | +/-H | 0 | - | - |
| 0**non-pos | M,0 | - | - | - | - | - |
| SQRT: | M,0 | - | - | - | - | - |
| GAMMA: | - | M,H | H | - | - | - |
| HYPOT: | - | - | H | - | - | - |
| SINH: | - | - | +/-H | - | - | - |
| COSH: | - | - | H | - | - | - |
| SIN, COS, TAN: | - | - | - | - | M,0 | * |
| ASIN, ACOS, ATAN2: | M,0 | - | - | - | - | - |

# matherr

## 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. |

## 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*/
}
```

# memory

## Name

**memccpy**, **memchr**, **memcmp**, **memcpy**, **memset** - memory operations

## Format

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1. *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

## Description

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

**memccpy** copies characters from memory area *s1* into *s2*, 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*.

## memory

**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 of these functions are declared in the optional <memory.h> header file.

### Known Problems

**memcmp** uses native character comparison, which is signed on some machines, but not on others. ASCII values are always positive, so programs that compare only ASCII values are portable.

Overlapping moves may yield surprises.

# mktemp

## Name

**mktemp** - make a unique file name

## Format

```
char *mktemp (template)
char *template;
```

## Description

The **mktemp** function replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of the *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.

## Known Problems

It is possible to run out of letters.

## See Also

**getpid** in Section 2; **tmpfile, tmpnam.**

# monitor

## Name

**monitor** - prepare execution profile

## Format

```
#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** (see Section 1) 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 the **profil** system call (see Section 2). *Lowpc* and *highpc* are the addresses of the two functions; *buffer* is an 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 profiled with the profiling option **-p** of **cc** are recorded. (The C Library and Math Library supplied when **cc -p** is used also have call counts 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(main, etext, buffer, bufsize, nfunc);
```

## monitor

*Etext* lies just above all the program text; see **end**, earlier in this section.

To stop execution monitoring and write the results on the file mon.out, use

```
monitor(0);
```

The **prof** command (see Section 1) can be used to examine the results.

### Files

mon.out
/lib/libp/libc.a
lib/libp/libm.a

### See Also

**cc, prof** in Section 1; **profil** in Section 2; **end**.

# nlist

## Name

**nlist** - get entries from the name list

## Format

```
#include <nlist.h>

int nlist (file-name, nl)
char *file-name;
struct nlist *nl;
```

## Description

The **nlist** function examines the name list in the executable file whose name is pointed to by *file-name*, 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** in Section 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.

The <nlist.h> header file is automatically included by <a.out.h> for compatibility. 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.

# nlist

## 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.

## See Also

**a.out** in Section 4.

# ocurse

## Name

**ocurse** - optimized screen functions

## Format

```
#include <ocurse.h>
```

## Description

**ocurse** is the old Berkeley **curses** library that uses **termcap** (see Section 4).

These functions optimally update the screen.

Each **curses** program begins by calling **initscr** and ends by calling **endwin**.

Before a program can change a screen, it must specify the changes. It stores changes in a variable of type WINDOW by calling **curses** functions with the variable as argument. Once the variable contains all the changes desired, the program calls **wrefresh** to write the changes to the screen.

Most programs need only a single WINDOW variable. **ocurse** provides a standard WINDOW variable for this case and a group of functions that operate on it. The variable is called *stdscr*; its special functions have the same name as the general functions minus the initial w.

### Functions

| | |
|---|---|
| addch(*ch*) | Add a character to *stdscr.* |
| addstr(*str*) | Add a string to *stdscr.* |
| box(*win,vert,hor*) | Draw a box around a window. |
| crmode() | Set cbreak mode. |
| clear() | Clear *stdscr.* |
| clearok(*scr,boolf*) | Set clear flag for *scr.* |
| clrtobot() | Clear to bottom on *stdscr.* |
| clrtoeol() | Clear to end of line on *stdscr.* |
| delch() | Delete a character. |
| deleteln() | Delete a line. |

## ocurse

| | |
|---|---|
| delwin(*win*) | Delete *win*. |
| echo() | Set echo mode. |
| endwin() | End window modes. |
| erase() | Erase *stdscr*. |
| getch() | Get a char through *stdscr*. |
| getcap(*name*) | Get terminal capability *name*. |
| getstr(*str*) | Get a string through *stdscr*. |
| gettmode() | Get tty modes. |
| getyx(*win,y,x*) | Get (*y,x*) coordinates. |
| inch() | Get char at current (*x,y*) coordinates. |
| initscr() | Initialize screens. |
| insch(*c*) | Insert a character. |
| insertln() | Insert a line. |
| leaveok(*win,boolf*) | Set leave flag for *win*. |
| longname(*termbuf,name*) | Get long name from *termbuf*. |
| move(*y,x*) | Move to (*y,x*) on *stdscr*. |
| mvcur(*lasty,lastx,newy,newx*) | Actually move cursor. |
| newwin(*lines,cols,begin_y,begin_x*) | Create a new window. |
| nl() | Set newline mapping. |
| nocrmode() | Unset cbreak mode. |
| noecho() | Unset echo mode. |
| nonl() | Unset newline mapping. |
| noraw() | Unset raw mode. |
| overlay(*win1,win2*) | Overlay *win1* on *win2*. |
| overwrite(*win1,win2*) | Overwrite *win1* on top of *win2*. |
| printw(*fmt,arg1,arg2,...*) | Printf on *stdscr*. |
| raw() | Set raw mode. |
| refresh() | Make current screen look like *stdscr*. |
| resetty() | Reset tty flags to stored value. |
| savetty() | Stored current tty flags. |
| scanw(*fmt,arg1,arg2,...*) | Scanf through *stdscr*. |
| scroll(*win*) | Scroll *win* one line. |
| scrollok(*win,boolf*) | Set scroll flag. |
| setterm(*name*) | Set term variables for *name*. |
| standend() | End standout mode. |
| standout() | Start standout mode. |
| subwin(*win,lines,cols,begin_y,begin_x*) | Create a subwindow. |
| touchwin(*win*) | Change all of *win*. |
| unctrl(*ch*) | Printable version of *ch*. |

# ocurse

| | |
|---|---|
| waddch(*win,ch*) | Add character to *win*. |
| waddstr(*win,str*) | Add string to *win*. |
| wclear(*win*) | Clear *win*. |
| wclrtobot(*win*) | Clear to bottom of *win*. |
| wclrtoeol(*win*) | Clear to end of line on *win*. |
| wdelch(*win,c*) | Delete char from *win*. |
| wdeleteln(*win*) | Delete line from *win*. |
| werase(*win*) | Erase *win*. |
| wgetch(*win*) | Get a char through *win*. |
| wgetstr(*win,str*) | Get a string through *win*. |
| winch(*win*) | Get char at current (*y,x*) in *win*. |
| winsch(*win,c*) | Insert char into *win*. |
| winsertln(*win*) | Insert line into *win*. |
| wmove(*win,y,x*) | Set current (*y,x*) coordinates on *win*. |
| wprintw(*win,fmt,arg1,arg2,...*) | Printf on *win*. |
| wrefresh(*win*) | Make screen look like *win*. |
| wscanw(*win,fmt,arg1,arg2,...*) | Scanf through *win*. |
| wstandend(*win*) | End standout mode on *win*. |
| wstandout(*win*) | Start standout mode on *win*. |

## Files

/usr/include/ocurse.h - header file
/usr/lib/libocurse.a - curses library
/usr/lib/libtermcap.a - termcap library, used by curses

## See Also

**stty** in Section 2; **setenv**; **termcap** in Section 4.

# ofCreate

## Name

**ofCreate, ofChangeFileLength, ofDelete** - allocate BTOS files

## Format

```
ofCreate (pbFileSpec, cbFileSpec, pbPassword, cbPassword,
    lfaFileSize)
char *pbFileSpec;
short cbFileSpec;
char *pbPassword;
short cbPassword;
long lfaFileSize;

ofChangeFileLength(fh, lfaNewFileSize)
short fh;
long lfaNewFileSize;

ofDelete(fh)
short fh;
```

## Description

The **ofCreate** function calls the BTOS **CreateFile** service, which creates a BTOS file. Arguments are:

- *PbFileSpec* and *cbFileSpec* specify the location and size of the new file's name. CENTIX processes lack a BTOS default path, so the name must begin with a volume name in square brackets [...], and a directory name in angle brackets <...>. The specified volume and directory must already exist. The file name that follows the volume and directory specifications can be up to 50 characters: upper case and lower case letters, digits, periods (.), hyphens (-), and right angle brackets (>). Here is an example with everything:

  [sys]<sys>Big1.subd>doc-Old

  **ofCreate** fails if the specified directory already has a file with the specified name. BTOS does not consider two file names distinct if they differ only in the case of their letters. However, a BTOS directory preserves the case of letters as specified by **ofCreate**.

## ofCreate

- *PbPassword* and *cbPassword* specify the location and size of the password that authorizes creation of the file. This password must match the volume or directory password. If the volume or directory lacks a password, no password is needed; set *cbPassword* to 0 and *pbPassword* to anything. (To give the file itself a password, see **ofstatus**.)
- *LfaFileSize* is the initial size of the file. The size must be a multiple of 512.

See **ofOpenFile** to provide a path handle for a newly created file.

The **ofChangeFileLength** function calls the BTOS **ChangeFileLength** service, which resets the length of a file. Arguments are:

- *Fh* is a file handle returned by **ofOpen**.
- *LfaNewFileSize* is the new size of the file. The size must be a multiple of 512.

The **ofDelete** function calls the BTOS **DeleteFile** service, which deletes a file. *Fh* is a file handle returned by an **ofOpen** in modify mode.

The program must be loaded with the library flag **-lctos**.

## Diagnostics

0 indicates success. **ofCreate** returns 224 if the file already exists.

## Caution

Frequent calls to **ofOpen** and **CloseFile** on a nearly full volume result in files whose contents are scattered about the disk. BTOS must add additional header blocks to the disk to keep track of the fragments. Frequent calls to **ofChangeFileLength** can have the same effect.

## See Also

**ofOpenFile, ofRead, ofDir, ofStatus, ofRename.**

## ofDir

### Name

**ofCrDir, ofDlDir, ofReadDirSector** - BTOS directory functions

### Format

```
ofCrDir (pbDirSpec, cbDirSpec, pbVolPassword, cbVolPassword,
      pbDirPassword, cbDirPassword, cSectors,
      defaultFileProtectionLevel)
char *pbDirSpec;
short cbDirSpec;
char *pbVolPassword;
short cbVolPassword;
char *pbDirPassword;
short cbDirPassword;
short cSectors;
short defaultFileProtectionLevel;

ofDlDir (pbDirSpec, cbDirSpec, pbPassword, cbPassword)
char *pbDirSpec;
short cbDirSpec;
char *pbPassword;
short cbPassword;

ofReadDirSector (pbDirSpec, cbDirSpec, pbPassword,
      cbPassword, iSector, pBufferRet)
char *pbDirSpec;
short cbDirSpec;
char *pbPassword;
short cbPassword;
short iSector;
char *pBufferRet;
```

# ofDir

## Description

The **ofDir** function calls the BTOS **CreateDir** service, which creates a BTOS directory. It takes the following arguments:

- *PbDirSpec* and *cbDirSpec* specify the location and size of the directory name. CENTIX processes lack a BTOS file path, sp the name must begin with a volume name in square brackets [...]. Angle brackets around the directory name (<...>) are optional. The specified volume must already exist. The directory name that follows the volume specification can be up to 12 characters: upper case and lower case letters, digits, periods (.), and hyphens (-). Here is an example with everything:

  [sys]<DH.1-Changes>

  **ofCrDir** fails if the specified volume already has a directory with the specified name. BTOS does not consider two directory names as distinct if they differ only in the case of their letters. However, the BTOS volume control structures preserve the case of letters as specified by **ofCrDir**.

- *PbVolPassword* and *cbVolPassword* specify the location and size of a password to be compared with the volume password. If the volume lacks a password, set *cbVolPassword* to 0 and *pbVolPassword* to anything.

- *PbDirPassword* and *cbDirPassword* specify the location and size of the password to be assigned to the directory. If the directory is to have no password, set *cbDirPassword* to 0 and *pbDirPassword* to anything.

- *Csectors* is the size of the directory in sectors. In general, one sector can store information on 15 files, but this depends on the length of the file names.

- *DefaultFileProtectionLevel* indicates the initial protection of files in the directory

## ofDir

The **ofDlDir** function calls the BTOS **DeleteDir** service, which deletes an empty directory. Delete or move all files from a directory before deleting the directory. **ofDlDir** takes the following arguments:

- *PbDirSpec* and *cbDirSpec* specify the location and size of the directory name. This name follows the same conventions used by **ofCrDir**.
- *PbPassword* and *cbPassword* specify the location and size of the password that authorizes the deletion of the directory. This password must match the volume password or the directory password. If volume or directory lack a password, no password is required to delete the directory.: set *cbPassword* to 0 and *pbPassword* to anything.

The **ofReadDirSector** function calls the BTOS **ReadDirSector** service, which reads a single 512-byte directory sector. It takes the following arguments:

- *PbDirSpec* and *cbDirSpec* specify the location and size of the directory name. This name follows the same conventions used by **ofCrDir**.
- *PbPassword* and *cbPassword* specify the location and size of the password that authorizes access of the directory. This password must match the volume password or the directory password. If the volume or directory lack a password, no password is required to delete the directory: set *cbPassword* to 0 and *pbPassword* to anything.
- *lSector* specifies which sector to read. Sectors are numbered from 0.
- *PBufferRet* points to a 512-byte area that will receive the sector.

The program must be loaded with the library flag **-lctos**.

# ofDir

## Diagnostics

0 indicates success. **ofCrDir** returns 240 ("Directory already exists") if the specified volume already has a directory with the specified name. **ofDlDir** returns 241 ("Directory not empty") if the directory still has files in it.

## See Also

**ofCreate**, **ofOpenFile**, **ofRead**, **ofStatus**, **ofRename**.

# ofOpenFile

## Name

**ofOpenFile, ofCloseFile, ofCloseAllFiles** - access BTOS files

## Format

```
ofOpenFile (pFhRet, pbFileSpec, cbFileSpec,
     pbPassword, cbPassword, mode)
short *pFhRet;
char *pbFileSpec;
short cbFileSpec;
char *pbPassword;
short cbPassword;
short mode;

ofCloseFile (fh)
short fh;

ofCloseAllFiles ()
```

## Description

The **ofOpenFile** function calls the BTOS **OpenFile** service, which opens an existing BTOS file. **ofOpenFile** takes the following arguments:

- *PFhRet* specifies where **ofOpenFile** is to return the file handle. This value is similar in use to a CENTIX file descriptor. Functions that do I/O, reallocate, and delete files require a valid file handle.
- *PbFileSpec* and *cbFileSpec* specify the location and length of the file name. CENTIX processes lack a BTOS default path, so the name must begin with a volume name in square brackets [...], and a directory name in angle brackets <...>. The remainder of the name must match a name in the specified directory, except that letters in the two names can differ in case. See **ofCreate**.
- *PbPassword* and *cbPassword* specify the location and size of a password that authorizes access to the file. The password required depends on the protection level of the file.

## ofOpenFile

- *Mode* specifies the access mode: 'm'*256+'r' for reading, 'm'*256+'m' for modifying.

A process that has file open in modify mode is the only process that can have the file open at all. An attempt to open a file in modify mode will fail if any other process already has that file open. An attempt to open a file in any mode will fail if another process already has that file open in modify mode.

Suppose you want to open for reading a file on volume sys and directory sys called danno.user. The following example works if no password is required:

```
fnmp="[sys]<sys>danno.user";
if((erc=ofOpenFile(&jhandle, fnmp, strlen(fnmp), 0, 0,
     'm'*256+'r')) != 0))
printf("BTOS open error %d\n", erc);
```

The **ofCloseFile** function calls the BTOS **CloseFile** service, which closes a file. *Fh* is a file handle previously provided by **ofOpenFile**.

**ofCloseAllFiles** closes all the process's BTOS files.

### Diagnostics

0 indicates success. If a modify mode **ofOpenFile** returns 220 ("File in use"), some other process has the file open for reading or modifying. If a read mode **ofOpenFile** returns 220, some other process has the file open for modifying.

### See Also

**ofCreate**, **ofRead**, **ofDir**, **ofStatus**, **ofRename**, **ofDir**.

## ofRead

### Name

**ofRead, ofWrite** - input/output on a BTOS file

### Format

```
ofRead (fh, pBufferRet, sBufferMax, lfa, psDataRet)
short fh;
char *pBufferRet;
short sBufferMax;
long lfa;
union {
        char *psDataRet
        short *DataRet
        };

ofWrite (fh, pBuffer, sBuffer, lfa, psDataRet)
short fh;
char *pBuffer;
short sBuffer;
long lfa;
union {
        char *psDataRet
        short *DataRet
        };
```

### Description

The **ofRead** function calls the BTOS **Read** service, which inputs one or more sectors from a BTOS file. It takes the following arguments:

- *Fh* is a file handle previously returned by **ofOpen**.
- *pBufferRet* points to a region large enough to hold the sector(s) read. The region must be on an even address; a union with a "short int" will force this.
- *sBufferMax* is the number of bytes desired. This must be a multiple of 512.
- *Lfa* is the offset, from the beginning of the file, of the first byte to be read. This must be a multiple of 512.

## ofRead

- *psDataRet* indicates where **ofRead** is to return the number of bytes actually read. This should point to a short word to work.

Note that you must read or write in multiples of 512 bytes.

The **ofWrite** function calls the BTOS **Write** service, which outputs one or more sectors. It takes the following arguments:

- *Fh* is a file handle previously returned by **openFile**.
- *PBuffer* points to the data to be output. The data must begin at an even address.
- *SBuffer* indicates the number of bytes to be output. This must be a multiple of 512.

  *Lfa* indicates the offset, from the beginning of the file, to which the data is to be written. This must be a multiple of 512.
- *PsDataRet* indicates where **ofWrite** is to return the number of bytes actually written.

The program must be loaded with the library flag **-lctos**.

### Diagnostics

0 indicates success. **ofWrite** returns 2 ("End of medium") if you attempt to write past the end of the file.

### Caution

If a BTOS process has written (or will read) binary integers to (from) the file, it stored (expects) them with Intel-byte ordering. See **swapshort**.

### See Also

**ofCreate**, **ofOpen**, **ofDir**, **ofStatus**, **ofRename**, **swapshort**.

# ofRename

## Name

**ofRename** - rename a BTOS file

## Format

```
ofRename (fh, pbNewFileSpec, cbNewFileSpec, pbPassword,
        cbPassword)
short fh;
char *pbNewFileSpec;
short cbNewFileSpec;
char *pbPassword;
short cbPassword;
```

## Description

The **ofRename** function calls the BTOS **RenameFile** service, which renames a BTOS file. It takes the following arguments:

- *Fh* is a file handle returned by an **openFile** in modify mode. This indicates the file to be renamed.
- *PbNewFileSpec* and *cbNewFileSpec* specify the location and size of the file's new name. The file name must include the volume and directory names. The file name conventions are the same as those for **ofCreate**.
- *PbPassword* and *cbPassword* specify the location and size of a password that authorizes the insertion of a file in the specified directory. This password must match the volume or directory password. If volume or directory lacks a password, no password is needed; set *cbPassword* to 0 and *pbPassword* to anything.

The program must be loaded with the library flag **-lctos**.

## Diagnostics

0 indicates success.

## ofRename

### Caution

A rename to a new directory is meaningful; a rename to a new volume is not.

### See Also

**ofCreate, ofOpenFile, ofRead, ofDir, ofStatus**.

## ofStatus

### Name

**ofGetFileStatus, ofSetFileStatus** - BTOS file status

### Format

```
ofGetFileStatus (fh, statusCode, pStatus, sStatus)
short fh;
short statusCode;
char *pStatus;
short sStatus;

ofSetFileStatus (fh, statusCode, pStatus, sStatus)
short fh;
short statusCode;
char *pStatus;
short sStatus;
```

### Description

The **ofGetFileStatus** and **ofSetFileStatus** functions call the BTOS **GetFileStatus** and **SetFileStatus** services, which get and set file information. They take the following arguments:

- *Fh* is a handle returned by a BTOS **OpenFile** in modify mode. *StatusCode* specifies the information to be obtained or changed. *StatusCode* must be one of the codes shown in Table 3-8. **ofSetFileStatus** only sets the items marked as settable.

Table 3-8 **BTOS File Status Codes**

| Code | Item | Size | Settable? |
|---|---|---|---|
| 0 | File Length | 4 | No |
| 1 | File Type | 1 | Yes |
| 2 | File protection level | f1 | Yes |
| 3 | Password | 13 | Yes |
| 4 | Date/time of creation | 4 | Yes |
| 5 | Date/time last modified | 4 | Yes |
| 6 | End-of-file pointer | 4 | Yes |
| 7 | File Header Block | 512 | No |
| 8 | Volume Home Block | 256 | No |
| 9 | Device Control Block | 100 | No |
| 10 | FHB Application Field | 64 | Yes |

## ofStatus

- *Pstatus* and *sStatus* specify the location and size of the area that holds, or is to receive, the data. If the area is not big enough, **ofGetFileStatus** right truncates the data to fit. When setting the password, use *sStatus* to indicate the password length. When getting the password, get the password length from the first byte in the data area.

A BTOS time is represented by the following formula:

(d * 0x20000) + (m * 0x10000) + s

where *d* is the number of days since the beginning of March, 1952 (in the local time zone); *m* is 0 for midnight/AM, 1 for noon/PM; *s* is the number of secondes since the last midnight or noon.

The program must be loaded with the library flag **-lctos**.

### Diagnostics

0 indicates success.

### See Also

**ofCreate, ofOpenFile, ofRead, ofDir, ofRename.**

# perror

## Name

**perror, errno, sys_errlist, sys_nerr** - system error messages

## Format

```
void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[];

extern int sys_nerr;
```

## Description

The **perror** function produces a message to the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable **errno**, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings **sys_errlist** is provided; **errno** can be used as an index in this table to get the message string without the new-line. **sys_nerr** is the largest number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

## See Also

**intro**.

# popen

## Name

**popen, pclose** - initiate pipe to/from a process

## Format

```
#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;
```

## Description

The arguments to **popen** are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either **r** for reading or **w** for writing. **popen** creates a pipe between the calling program and the command to be executed. The value returned is a stream pointer such that you can write to the standard input of the command, if the I/O mode is **w**, by writing to the file *stream*; and you can read from the standard output of the command, if the I/O mode is **r**, by reading from the file *stream*.

A stream opened by **popen** should be closed by **pclose**, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter and a type **w** command as an output filter.

## Diagnostics

**popen** returns a NULL pointer if files or processes cannot be created, or if the shell cannot be accessed.

**pclose** returns -1 if *stream* is not associated with a **popened** command.

# popen

## Known Problems

If the original and **popen**ed processes concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Problems with an output filter may be forestalled by careful buffer flushing, such as with **fflush** (see **fclose**).

## See Also

**pipe**, **wait** in Section 2; **fclose**, **fopen**, **system**.

# printf

## Name

**printf, fprintf, sprintf** - print formatted output

## Format

```
#include <stdio.h>

int printf (format[, arg]...)
char *format;

int fprintf (stream, format[, arg]...)
FILE *stream;
char *format;

int sprintf (s, format[, arg]...)
char *s, format;
```

## Description

The **printf** function places output on the standard output stream *stdout*. **fprintf** places output on the named output *stream*. **sprintf** places "output," followed by the null character (\0), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the \0 in the case of **sprintf**), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- Zero or more *flags*, which modify the meaning of the conversion specification.

## printf

- An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '-,' described below, has been given) to the field width. If the field width for an **s** conversion is preceded by a 0, the string is right-adjusted with zero-padding on the left.
- A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in an **s** conversion. The precision takes the form of a period (.), followed by a decimal digit string; a null digit string is treated as zero.
- An optional **l** specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*. An **l** before any other conversion character is ignored.
- A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear before the *arg* (if any) to be converted.

The flag characters and their meanings are:

| | |
|---|---|
| - | The result of the conversion will be left-justified within the field. |
| + | The result of a signed conversion will always begin with a sign (+ or -). |
| blank | If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored. |

## printf

| | |
|---|---|
| # | This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For an **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** and **X** conversions, a non-zero result will have 0x or 0X prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeros will not be removed from the result (they normally are removed). |

The conversion characters and their meanings are:

| | |
|---|---|
| **d, o, u, x, X** | The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. (For compatibility with older versions, padding with leading zeros may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string. |
| **f** | The float or double *arg* is converted to decimal notation in the style "[-]ddd.ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly zero, no decimal point appears. |
| **e,E** | The float or double *arg* is converted in the style "[-]d.ddd+/-dd" where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. |

# printf

| | |
|---|---|
| g,G | The float or double *arg* is printed in style **f** or **e** (or in style E in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result; a decimal point appears only if it is followed by a digit. |
| c | The character *arg* is printed. |
| s | The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results. |
| % | Print a %; no argument is converted. |

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by **printf** and **fprintf** are printed as if **putc** had been called.

## Examples

To print the date and time in the form "Sunday, July 3, 10:02," where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d", weekday,month,day,hour,min);
```

To print pi to 5 decimal places:

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

## See Also

**ecvt, putc, scanf, stdio**.

# putc

## Name

**putc, putchar, fputc, putw** - put character or word on a stream

## Format

```
#include <stdio.h>

int putc (c, stream)
int c;
FILE *stream;

int putchar (c)
int c;

int fputc (c, stream)
int c;
FILE *stream;

int putw (w, stream)
int w;
FILE stream;
```

## Description

**putc** writes the character *c* onto the output *stream* (at the position where the file pointer, if defined, is pointing). **putchar**(*c*) is defined as **putc**(*c, stdout*). **putc** and **putchar** are macros.

The **fputc** function behaves like **putc**, but is a function rather than a macro. **fputc** runs more slowly than **putc**, but it takes less space per invocation and its name can be passed as an argument to a function.

**putw** writes the word (or integer) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of the integer and varies from machine to machine. **putw** neither assumes nor causes special alignment in the file.

## putc

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of **freeopen** (see **fopen**) will cause it to become buffered or line-buffered. When the output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). **setbuf** may be used to change the stream's buffering strategy.

### Diagnostics

On success, these functions each return the value they have written. On failure, they return the constant EOF. This will occur if the file *stream* is not open for writing or if the output file cannot be grown. Because EOF is a valid integer, **ferror** should be used to detect **putw** errors.

### Known Problems

Because it is implemented as a macro, **putc** treats incorrectly a *stream* argument with side effects. In particular, **putc** (*c*, *f++); doesn't work sensibly. **fputc** should be used instead. Because of possible differences in word length and byte ordering, files written using **putw** are machine-dependent, and may not be read using **getw** on a diffrent processor.

### See Also

**fclose, ferror, fopen, fread, printf, puts, setbuf.**

## putenv

### Name

**putenv** - change or add value to environment

### Format

```
int putenv (string)
char *string;
```

### Description

*String* points to a string of the form *"name=value."* The **putenv** function 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**.

### Diagnostics

**putenv** returns non-zero if it is unable to obtain enough space via **malloc** for an expanded environment; otherwise, the function returns zero.

### Cautions

**putenv** manipulates the environment pointed to by **environ**, and can be used in conjunction with **getenv**. However, *envp* (the third argument to *main*) is not changed.

This routine uses the **malloc** function to enlarge the environment.

After **putenv** is called, environmental variables are not in alphabetical order.

## putenv

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.

### See Also

**exec** in Section 2; **getenv**, **malloc**; **environ** in Section 5.

# putpwent

## Name

**putpwent** - write password file entry

## Format

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

## Description

The **putpwent** function is the inverse of **getpwent**. 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.

## Diagnostics

**putpwent** returns non-zero if an error is detected during its operation; otherwise, it returns zero.

## Caution

The above routine uses <stdio.h>, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

## See Also

**getpwent**.

# puts

## Name

**puts, fputs** - put a string on a stream

## Format

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

## Description

The **puts** function writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream *stdout*.

**fputs** writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

**puts** appends a new-line character while **fputs** does not.

## Diagnostics

Both routines return EOF on error. This will happen if the routines try to write on a file that has not been opened for writing.

## See Also

**ferror, fopen, fread, printf, putc.**

## qsort

### Name

**qsort** - quicker sort

### Format

```
void qsort ((char *)base, nel, sizeof(*base), compar)

unsigned int nel;
int (*compar)();
```

### Description

The **qsort** function is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

*Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero, depending on whether the first argument is to be considered less than, equal to, or greater than the second.

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

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

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

### See Also

**sort** in Section 1; **bsearch**, **lsearch**, **string**.

## quAdd

### Name

**quAdd** - add a new entry to a BTOS queue

### Format

```
quAdd (pbQueueName, cbQueueName, fQueueIfNoServer,
      priority, queueType, pEntry, sEntry, pDateTime,
      repeatTime)
char *pbQueueName;
short cbQueueName;
char fQueueIfNoServer;
char priority;
short queueType;
char *pEntry;
short sEntry;
unsigned long *pDateTime;
short repeatTime;
```

### Description

The **quAdd** function calls the BTOS **AddQueueEntry** service. A CENTIX process that wants to submit a request to a BTOS queue server creates a queue entry with **quAdd**. **quAdd** takes the following arguments:

- *PbQueueName* and *cbQueueName* describe the location and length of a queue name. This must be one of the queues mentioned in the BTOS file [sys]<sys>queue.index.
- *FQueueIfNoServer* determines the action if the queue manager finds that no servers are active for the specified queue. **0xFF** means to queue the entry anyway. **0** means abort the queue entry.
- *Priority* sets the queue entry's priority. **0** is the highest priority, **9** is the lowest.
- *QueueType* is the type of queue. This must match the number given in the fourth field of the queue's entry in the queue index file.

## quAdd

- *PEntry* and *sEntry* describe the size and location of entry data. The size and layout of this data area is conventional for each queue.
- *PDateTime* points to the service time. A server will serve the request no sooner than the service time.

  The service time must be in BTOS format:

  (d * 0x20000) + (m * 0x10000) + s

  where *d* is the number of days since the beginning of March, 1952 (in the local time zone); *m* is 0 for midnight/AM, 1 for noon/PM; *s* is the number of seconds since the last midnight or noon.

  A service time of 0 means "undated;" the queue manager provides servers for all undated requests before it provides servers for any dated requests.
- *RepeatTime* specifies a repeat interval. Unless this value is 0, the queue manager resubmits the request *repeatTime* minutes after a queue server deletes it. Thus the request repeats forever, with at least *repeatTime* minutes between repetitions. A CENTIX process can terminate this loop with the **quRemove** function.

Queue servers run under BTOS and thus expect integers to have Intel-byte ordering. **quAdd** translates *queueType*, the date, and *repeatTime*, but does nothing about entry data. To translate entry data, see **swapshort**.

The program must be loaded with the library flag **-lctos**.

### Files

[sys]<sys>queue.index - master queue index

# quAdd

## Diagnostics

0 indicates success. 254 ("Queue not served") if *fQueueIfNoServer* is 0 and no servers are active on the specified queue.

## See Also

**quRemove, quRead.**

# quRead

## Name

**quReadNext**, **quReadKeyed** - examine BTOS queue

## Format

```
structQueueStatusBlock {
      long qehRet;
      char priority;
      char padding;
      short ServerUserNumber;
      long qehNextRet;
      };

quReadNext (pbQueueName, cbQueueName, qeh, pEntryRet,
      sEntryRet, pStatusBlock, sStatusBlock)
char *pbQueueName;
short cbQueueName;
long qeh;
char *pEntryRet;
short sEntryRet;
struct QueueStatusBlock *pStatusBlock;
short sStatusBlock;

quReadKeyed (pbQueueName, cbQueueName, pbKey1, cbKey1,
      oKey1, pbKey2, cbKey2, oKey2, pEntryRet, sEntryRet,
      pStatusBlock, sStatusBlock)
char *pbQueueName;
short cbQueueName;
char *pbKey1;
short cbKey1;
short oKey1;
char *pbKey2;
short cbKey2;
short okey2;
char *pEntryRet;
short sEntryRet;
struct QueueStatusBlock *pStatusBlock;
short sStatusBlock;
```

# quRead

## Description

The **quReadNext** and **quReadKeyed** functions call the BTOS **ReadNextQueueEntry** and **ReadKeyedQueueEntry** services. A queue client uses **quReadNext** or **quReadKeyed** to examine a BTOS queue. Each call returns information on a single queue entry. **quReadNext** and **quReadKeyed** have the following arguments in common:

- *PbQueueName* and *cbQueueName* describe the location and size of a queue name.
- *PEntryRet* and *sEntryRet* describe the location and size of an area that is to receive entry data. Size and layout of entry data is specific to each queue. If the area is smaller than an area's data, the data is right–truncated to fit.
- *PStatusBlock* and *sStatusBlock* describe the location and size of an area that is to receive the entry's status block. If the area is smaller than sizeof(QueueStatusBlock), the block is right-truncated to fit.

**quReadNext** and **quReadKeyed** return the following values in the status block:

- *QehRet* is the queue entry handle. This integer value is unique for each entry in the queue.
- *Priority* is the priority of the entry.
- *ServerUserNum* is the BTOS user number of the queue server that has appropriated (marked) the request and plans to service it. If no server has appropriated the request, *serverUserNum* is -1.
- *QehNextRet* is the queue entry handle for the next entry in the queue. If the current entry is the last entry in the queue, *QehNextRet* is -1.

The following argument is specific to **quReadNext**:

- *Qeh* specifies the queue entry to be read. 0 indicates the first queue entry; any other value must be a queue entry handle.

## quRead

This example passes the data for each entry in SPL to prentry( ).

```
qnl = strlen(qns = "SPL");
for (handle = 0; handle != -1; handle = status.QehNextRet) {
    quReadNext(qnl, qns, handle, &data,
        sizeof(data), &status, sizeof(status));
    prentry(&status);
}
```

The following arguments are specific to **quReadKeyed**.

- *PbKey1* and *cbKey1* describe the location and size of the first search key. If there is no search key, set *cbKey1* to 0.
- *Okey1* is the offset of the first search string. This is the offset, from the beginning of the entry data, of a string that is to be compared with the first search key. **quReadKeyed** assumes that the first byte of this string gives the size of the remainder of the string. If there is no first search key, the function ignores *oKey1*.
- *PbKey2* and *cbKey2* describe the location and size of the second search key. If there is no second search key, set *cbKey2* to 0.
- *OKey2* is the offset of the second search string. This is the offset, from the beginning of the entry data, of a string that is to be compared with the second search key. **quReadKeyed** assumes that the first byte of this string gives the size of the remainder of the string. If there is no second search key, the function ignores *oKey2*.

The client that calls **quReadKeyed** must supply 1 or 2 search keys. **quReadKeyed** returns the first entry that matches both search keys. If only one key is given, **quReadKeyed** returns the first entry that matches that single key.

The program must be loaded with the library flag **-lctos**.

### Files

[sys]<sys>queue.index - master queue index

## quRead

### Diagnostics

0 indicates success. **quReadNext** returns 904 ("Entry deleted") if another client deletes a queue entry between the time you get the entry's handle and the time you try to read it.

### See Also

**quRemove, quAdd**.

# quRemove

## Name

**quRemove** - take back a BTOS queue request

## Format

```
quRemove (pbQueueName, cbQueueName, pbKey1, cbKey1,
     oKey1, pbKey2, cbKey2, oKey2)
char *pbQueueName;
short cbQueueName;
char *pbKey1;
short cbKey1;
short oKey1;
char *pbKey2;
short cbKey2;
short oKey2;
```

## Description

The **quRemove** function calls the BTOS **RemoveKeyedQueueEntry** service. A queue client uses **quRemove** to delete entries from a BTOS queue. **quRemove** uses search keys to identify the request. It takes the following arguments:

- *PbQueueName* and *cbQueueName* describe the location and size of a queue name.
- *PbKey1* and *cbKey1* describe the location and size of the first search key. If there is no first search key, set *cbKey1* to 0.
- *OKey1* is the offset of the first search string. This is the offset, from the beginning of the entry data, of a string that is to be compared with the first search key. **quRemove** assumes that the first byte of this string gives the size of the remainder of the string. If there is no first search key, the function ignores *oKey1*.
- *PbKey2* and *cbKey2* describe the location and size of the second search key. If there is no second search key, set *cbKey2* to 0.

## quRemove

- *OKey2* is the offset of the second search string. This is the offset, from the beginning of the entry data, of a string that is to be compared with the second search key. **quRemove** assumes that the first byte of this string gives the size of the remainder of the string. If there is no second search key, the function ignores *oKey2.*

The client that calls **quRemove** must supply 1 or 2 search keys. **quRemove** deletes the first entry that matches both search keys. If only one key is given, **quRemove** deletes the first entry that matches the single key, *oKey2.*

The program must be loaded with the library flag **-lctos**.

## Files

[sys]<sys>queue.index - master queue index

## See Also

**quAdd, quRead**.

# rand

## Name

**rand**, **srand** - simple random number generator

## Format

```
int rand ()

void srand (seed)
unsigned seed;
```

## Description

The **rand** function 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.

Note that the spectral properties of **rand** leave much to be desired. The **drand48** function provides a much better, though more elaborate, random number generator.

## See Also

**drand48**.

# regcmp

## Name

**regcmp**, **regex** - compile and execute regular expression

## Format

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

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

extern char *__loc1;
```

## Description

The **regcmp** function compiles a regular expression and returns a pointer to the compiled form. The **malloc** function is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A NULL return from **regcmp** indicates an incorrect argument. **regcmp** has been written to generally preclude the need for this routine at execution time.

The **regex** function executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. **regex** returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *__loc1* points to where the match began. **regcmp** and **regex** were mostly borrowed from the editor, **ed** (see Section 1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings:

| | |
|---|---|
| [ ]*.^ | These symbols retain their current meaning. |
| $ | atches end of a string; \n matches a new-line. |
| – | Within brackets, the minus means "through." For example, [a-z] is equivalent to [abcd...xyz], which means [a through z]. The - can appear as itself only if used as the first or last character. For example, the character class expression [ ]-] matches the characters ] and -. |

## regcmp

| | |
|---|---|
| + | A regular expression followed by + means one or more times. For example, [0-9]+ is equivalent to [0-9][0-9]*. |
| {m} {m,} {m,u} | Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number a *u* is a number, less than 256, which is the maximum. If only *m* is present (that is, {m}), it indicates the exact number of times the regular expression is to be applied. The value {m,} is analogous to {m,infinity}. The plus (+) and asterisk (*) operations are equivalent to {1,} and {0,}, respectively. |
| (...)$*n* | The value of the enclosed regular expression is to be returned. The value will be stored in the (*n*+1)th argument following the subject argument. At most ten enclosed regular expressions are allowed. **regex** makes its assignments unconditionally. |
| (...) | Parentheses are used for grouping. An operator (such as *, +, { }) can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0. |

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

### Examples

The following example will match a leading new-line in the subject string pointed at by cursor:

```
char *cursor, *newcursor, *ptr;

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

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

```
char ret0[9];
char *newcursor, *name;

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

## regcmp

The third example applies a precompiled regular expression in file.i (see **regcmp** in Section 1) against *string*:

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

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

## Known Problems

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

```
/*user's program*/
     ...
char *
malloc(n)
unsigned n;
{
        static char rebuf[512]:
        return (n <= sizeof rebuf) ? rebuf : NULL;
}
```

## See Also

**ed**, **regcmp** in Section 1; **malloc**.

# scanf

## Name

**scanf**, **fscanf**, **sscanf** - convert formatted input

## Format

```
#include <stdio.h>

int scanf (format[, pointer] ...)
char *format;

int fscanf (stream, format[, pointer] ...)
FILE *stream;
char *format;

int sscanf (s, format[, pointer] ...)
char *s, *format;
```

## Description

The **scanf** function reads from the standard input stream *stdin*. **fscanf** reads from the named input *stream*. **sscanf** reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format*, described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1 White-space characters (blanks, tabs, new-lines or form–feeds) that, except in two cases described below, cause input to be read up to the next non-white-space character.

2 An ordinary character (not %), which must match the next character of the input stream.

# scanf

3 Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional l or h indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field that is to be skipped. An input field is described as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c," white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

| | |
|---|---|
| % | A single % is expected in the input at this point; no assignment is done. |
| d | A decimal integer is expected; the corresponding argument should be an integer pointer. |
| u | An unsigned decimal integer is expected; the corresponding argument should be an integer pointer. |
| o | An octal integer is expected; the corresponding argument should be an integer pointer. |
| x | A hexadecimal integer is expected; the corresponding argument should be an integer pointer. |
| e,f,g | A floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e, followed by an optional +, -, or space, followed by an integer. |
| s | A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which will be added automatically. The input field is terminated by a white-space character. |

## scanf

| | |
|---|---|
| c | A character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use %1s. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read. |
| [ | Indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *canset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The ^, when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters not contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first-last*, thus [0123456789] may be expressed as [0–9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possible preceded by a ^) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket.. The corresponding argument must point to a character array large enough to hold the data field and the terminating \0, which will be added automatically. At least one character must match for this conversion to be considered successful. |

The conversion characters d, u, o, and x may be preceded by l or h to indicate that a pointer to long or to short, rather than to int, is in the argument list. Similarly, the conversion characters e, f, and g may be preceded by l to indicate that a pointer to double, rather than to float, is in the argument list. The l or h modifier is ignored for other conversion characters.

**scanf** conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

**scanf** returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

Note that trailing white space (including a new-line) is left unread unless matched in the control string.

# scanf

## Examples

The call

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line

```
25 54.32E-1 henry
```

will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain henry\0. Or

```
int i; float x, char name[50];
(void)scanf("%2d%f%*d%[0-9]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip 0123, and place the string 56\0 in *name*. The next call to **getchar** (see **getc**) will return a.

## Diagnostics

These functions return EOF on end of input and a short count for missing or illegal data items.

## Known Problems

The success of literal matches and suppressed assignments is not directly determinable.

## See Also

**getc, printf, strtod, strtol.**

# setbuf

## Name

**setbuf**, **setvbuf** - assign buffering to a stream

## Format

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;
```

## Description

The **setbuf** function may be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer, input/output will be completely unbuffered.

A constant, BUFSIZ, defined in the <stdio.h> header file, tells how big an array is needed:

```
char buf[BUFSIZ]
```

**setvbuf** may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type*, defined in stdio.h, are:

| | |
|---|---|
| _IOFBF | Causes input/output to be fully buffered. |
| _IOLBF | Causes output to be line buffered; the buffer will be flushed when a new-line is written, the buffer is full, or input it requested. |
| _IONBF | Causes input/output to be completely unbuffered. |

If *buf* is not the NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *Size* specifies the size of the buffer to be used. The constant BUFSIZ in <stdio.h> is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

## setbuf

By default, output to a terminal is line buffered, and all other input/output is fully buffered.

Note that a common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

## Diagnostics

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

## See Also

**fopen, getc, malloc, putc, stdio.**

# setjmp

## Name

**setjmp**, **longjmp** - non-local goto

## Format

```
#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.

The **setjmp** function 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.

The **longjmp** function restores the environment saved by the last call of **setjmp** with the corresponding *env* argument. After **longjmp** is completed, program execution continues as if the corresponding call of **setjmp** (which must not itself have returned in the interim) had just returned the value *val*. **longjmp** cannot cause **setjmp** to return the value 0. If **longjmp** is invoked with a second argument of 0, **setjmp** will return 1. All accessible data have values as of the time **longjmp** was called.

## Caution

If **longjmp** is called when *env* was never primed by a call to **setjmp**, or when the last such call is in a function that has since returned, absolute chaos is guaranteed.

## See Also

**signal** in Section 2.

## sinh

### Name

**sinh**, **cosh**, **tanh** - hyperbolic functions

### Format

```
#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

### Description

The **sinh**, **cosh**, and **tanh** functions return, respectively, the hyperbolic sine, cosine, and tangent of their arguments.

### Diagnostics

**sinh** and **cosh** return HUGE (and **sinh** may return -HUGE for negative *x*) and set **errno** to ERANGE when the correct value would overflow.

These error-handling procedures may be changed with the **matherr** function.

### See Also

**matherr**.

# sleep

## Name

**sleep** - suspend execution for interval

## Format

```
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 until such an alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is then executed just before the **sleep** routine returns, unless the **sleep** time is less than the time until the alarm, in which case the prior alarm time is reset to go off at the same time it would have without the intervening **sleep**.

## See Also

**alarm, pause, signal** in Section 2.

# spawn

## Name

**spawnlp, spawnvp** - execute a process on a specific Application Processor

## Format

```
int
spawnlp (apnum, directory, name, arg0, arg1, ..., argn, 0)
int apnum;
char *directory;
char *name, *arg0, *arg1, ..., *argn;

int
spawnvp (apnum, directory, name, argv)
int apnum;
char *directory;
char *name, *argv[];

extern char **environ;
```

## Description

The **spawn** functions, **spawnlp** and **spawnvp,** execute a file on the specified AP, creating a new process on that processor. The practical effect is that of a **fork/exec** sequence with the following differences:

- **spawn** will create the new process on any AP. **fork/exec** always creates the new process on the parent process's application processor.
- A **spawn** process is not a child of the process that called **spawn**; it is a child of the spawn server on the designated AP (see **spawnsrv** in Section 1). Thus the process that called **spawn** cannot **wait** (see Section 2) for the new process's death; use **spwait** instead. Also, not all the attributes that are inherited across a **fork** are inherited across a **spawn**.
- A **fork/exec** is less expensive than a **spawn**.

## spawn

The spawn server passes the following attributes to the new process, based partially on the attributes of the calling process:

- File descriptors 0, 1, and 2 (standard input, output, and error) of the new process are open to /dev/null. None of the calling process's file descriptors are available to the new process.
- Signals caught by the calling process terminate the new process. Other signals (ignored by or causing termination of the calling process) have the same effect on the new process they had on the calling process.
- The new process inherits the following, unchanged, from the calling process: environment parameters (variables); file creation mask (**umask**, Section 2); effective user ID and group ID.
- If the calling process's effective user ID is 0, the new process inherits the calling process's real user ID and group ID. Otherwise, the new process's real IDs are the same as its effective IDs.

The calling conventions for **spawnlp** and **spawnvp** are the same as for **execlp** and **execvp** (see Section 2), but with two additional parameters at the beginning:

| | |
|---|---|
| *apnum* | The number of the AP that is to run the new process. Application processors are numbered from 0. Viewed from behind, APs in the rightmost enclosure are counted first, working left; within an enclosure, count left to right. See the *XE 500 CENTIX Administration Guide.* |
| *directory* | A pointer to a null-terminated string identifying the new process's working directory. If *directory* is (char *)0 (NULL in <stdio.h>), the new process's working directory is the same as the calling process's. (Use of NULL is expensive: it causes a call to **pwd**; see Section 1.) |

## spawn

### Examples

The following runs myprog in the same directory as the current process, but it runs on AP01:

```
#define NULL ((char *)0)
spawnlp(01, NULL, "myprog", "myprog", "arg1", NULL);
```

The following runs a shell on the other AP:

```
    spawnlp(01, "/", "/bin/sh", "-sh", "-c",
            "cd $HOME; exec myprog", NULL);
```

### Diagnostics

Both functions return -1 on error; otherwise, they return the process number of the new process.

### See Also

**apnum, pwd, spawn** in Section 1; **apnum, fork, signal** in Section 2; **getcwd, spwait; environ** in Section 5.

# sputl

## Name

**sputl**, **sgetl** - access long integer data in a machine–dependent fashion

## Format

```
void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;
```

## Description

The **sputl** function takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

The **sgetl** function retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of **sputl** and **sgetl** provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

A program that uses these functions must be loaded with the object-file access routine library libld.a.

## spwait

### Name

**spwait** - wait for a spawned process to terminate

### Format

```
spwait (pid, status)
int pid, *status;
```

### Description

The **spwait** function suspends the calling process until a signal is received or the process specified by process ID *pid* terminates. The specified process must have been previously spawned (see **spawn**) by the calling process.

If *status* is not equal to (int *)0, the word it points to receives two data:

- The high byte gets the low byte of the specified process's **exit** (see Section 2) parameter.
- The low byte gets the specified process's termination status. If the termination status's 0200 bit is set, the process produced a core image when it terminated.

### Diagnostics

If **spwait** 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 terminated spawn process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

### See Also

**spawn** in Section 1; **exit**, **fork**, **signal** in Section 2; **spawn**.

# ssignal

## Name

**ssignal**, **gsignal** - software signals

## Format

```
#include <signal.h>

int (*ssignal (sig, action))()
int sig, (*action)();

int gsignal (sig)
int sig;
```

## Description

The **ssignal** and **gsignal** functions implement a software facility similar to **signal** in Section 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 15. A call to **ssignal** associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to **gsignal**. Raising a software signal causes the action established for that signal to be taken.

The first argument to **ssignal** is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of the (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.

## ssignal

The **gsignal** function 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 the 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.

Note that there are some additional signals with numbers outside the range 1 through 15 that are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

### See Also

**signal** in Section 2.

# stdio

## Name

**stdio** - standard buffered input/output package

## Format

```
#include <stdio.h>

FILE *stdin, *stdout, *stderr;
```

## Description

These functions, as well as the other functions whose declarations are obtained from the #include file <stdio.h>, constitute an efficient, user-level I/O buffering scheme. The in-line macros **getc** and **putc** handle characters quickly. The macros **getchar** and **putchar**, and the higher-level routines **fgetc** , **fgets**, **fprintf**, **fputc**, **fputs**, **fread**, **fscanf**, **fwrite**, **gets**, **getw**, **printf**, **puts**, **putw**, and **scanf** all use or act as if they use **getc** and **putc**; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type FILE. The **fopen** function creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

| | |
|---|---|
| **stdin** | Standard input file. |
| **stdout** | Standard output file. |
| **stderr** | Standard error file. |

A constant NULL (0) designates a non-existent pointer.

An integer-constant EOF (-1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant BUFSIZ specifies the size of the buffers used by the particular implementation.

## stdio

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

These functions and constants are declared in that header file and need no further declaration. The constants and the following functions are implemented as macros (redeclaration of these names is perilous): **getc**, **getchar**, **putc**, **putchar**, **ferror**, **feof**, **clearerr**, and **fileno**.

### Diagnostics

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

### See Also

**open**, **close**, **lseek**, **pipe**, **read**, **write** in Section 2; **intro**, **ctermid**, **cuserid**, **fclose**, **ferror**, **fopen**, **fread**, **fseek**, **getc**, **gets**, **popen**, **printf**, **putc**, **puts**, **scanf**, **setbuf**, **system**, **tmpfile**, **tmpnam**, **ungetc**.

# stdipc

## Name

**stdipc** - standard interprocess communication package (**ftok**)

## Format

```
#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**, **semget**, and **shmget** system calls (see Section 2) 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. Their 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 that 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 but different *id*s.

# stdipc

## Diagnostics

**ftok** returns (key_t)-1 if *path* does not exist or if it is not accessible to the process.

## Caution

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 returned 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.

## See Also

**intro, msgget, semget, shmget** in Section 2.

# string

## Name

**strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok** - string operations

## Format

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;

char strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s2, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

# string

## Description

The arguments *s1*, *s2*, and *s* point to strings (arrays of characters pointed to by a null character). The functions **strcat**, **strncat**, **strcpy**, and **strncpy** all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

**strcat** appends a copy of string *s2* to the end of string *s1*. **strncat** appends at most *n* characters. Each returns a pointer to the null-terminated result.

The **strcmp** function compares its arguments and returns an integer less than, equal to, or greater than 0, depending on whether *s1* is lexicographically less than, equal to, or greater than *s2*. **strncmp** makes the same comparison but looks at at most *n* characters.

**strcpy** copies strings *s2* to *s1*, stopping after the null character has been copied. **strncpy** copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

**strlen** returns the number of characters in *s*, not including the terminating null character.

**strchr (strrchr)** returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

**strpbrk** returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a NULL pointer if no character from *s2* exists in *s1*.

**strspn (strcspn)** returns the length of the initial segment of string *s1*, which consists entirely of characters from (not from) string *s2*.

## string

**strtok** considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that on subsequent calls (which must be made with the first argument a NULL pointer) will work through the string *s1* immediately following that token. In this way, subsequent calls will work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

Note that for user convenience, all of the above functions are declared in the optional <string.h> header file.

### Known Problems

**strcmp** and **strncmp** use native character comparison, which is signed on Burroughs 68000-family processors. This means that characters are 8-bit signed values; all ASCII characters have values of at least 0; non-ASCII are negative. On some machines, all characters are positive. Thus programs that only compare ASCII values are portable; programs that compare ASCII with non–ASCII values are not.

Overlapping moves may yield surprises.

# strtod

## Name

**strtod**, **atof** - convert string to double-precision number

## Format

```
double strtod (str, ptr)
char *str, **ptr;

double atof (str)
char *str;
```

## Description

The **strtod** function 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**), 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*).

## Diagnostics

If the correct value would cause overflow, plus or minus HUGE 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.

## See Also

**ctype**, **scanf**, **strtol**.

# strtol

## Name

**strtol, atol, atoi** - convert string to integer

## Format

```
long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char str;

int atoi (str)
char *str;
```

## Description

The **strtol** function 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**) 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 as follows: 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).

# strtol

## Known Problems

Overflow conditions are ignored.

## See Also

**ctype, scanf, strtod.**

# swab

## Name

**swab** - swap bytes

## Format

```
void swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

## Description

The **swab** function copies *nbytes* pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between machines. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive, **swab** uses *nbytes*-1 instead. If *nbytes* is negative, *swab* does nothing.

## swapshort

### Name

**swapshort, swaplong** - translate byte orders to Motorola/Intel

### Format

```
swapshort (s)
short s;

swaplong (l)
long l;
```

### Description

Processes that run on an XE 500 CENTIX Application Processor do not store integers the same way as do processes that run on other (BTOS) processors. CENTIX processes use Motorola ordering; BTOS processes use Intel ordering. CENTIX processes must translate integers sent to or received from BTOS processes.

Library functions do this translation whenever they know an integer value is involved. For example, **AddQueueEntry** translates integers that are supplied for all queue entries: the priority, the queue type, and the data. But **AddQueueEntry** does not translate any integers in the entry data.

**swaplong** translates to or from Intel four-byte integers. **swaplong** returns *l* with its bytes in reverse order. For example, if *l* is 4885001 (0x004A8A09), **swaplong** returns 160057856 (0x098A4A00).

**swapshort** translates to or from Intel two-byte integers. **swapshort** returns *s* with its bytes in reverse order.

The program must be loaded with the **-lctos** library flag.

# system

## Name

**system** - issue a shell command

## Format

```
#include <stdio.h>

int system (string)
char *string;
```

## Description

The **system** function causes the *string* to be given to **sh** (see Section 1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

## Files

/bin/sh

## Diagnostics

**system** forks to create a child process that in turn exec's /bin/sh in order to execute *string*. If the fork or exec fails, **system** returns -1 and sets **errno** to indicate the error.

## See Also

**sh** in Section 1; **exec** in Section 2.

# termcap

## Name

**tgetent**, **tgetnum**, **tgetflag**, **tgetstr**, **tgoto**, **tputs** - terminal independent operations

## Format

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

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

tgetnum (id)
char *id;

tgetflag (id)
char *id;

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

char *
tgoto (cmstr, destcol, destline)
char *cmstr;
int destcol, destline;

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

## Description

These functions extract and use information from terminal descriptions that follow the conventions in **termcap** (see Section 4). The functions do only basic screen manipulation: they find and output specified terminal function strings and interpret the cm string. **curses** describes a screen updating package built on **termcap**.

## termcap

**tgetent** finds and copies a terminal description. *Name* is the name of the description; *bp* points to a buffer to hold the description. **tgetent** passes *bp* to the other **termcap** functions; the buffer must remain allocated until the program is done with the **termcap** functions.

**tgetent** uses the TERM and TERMCAP environment variables to locate the terminal description.

- If TERMCAP is not set, or if it is empty, **tgetent** searches for *name* in /etc/termcap.
- If TERMCAP contains the full pathname of a file (any string that begins with /), **tgetent** searches for *name* in that file.
- If TERMCAP contains any string that does not begin with / and TERM is not set or matches *name*, **tgetent** copies the TERMCAP string.
- If TERMCAP contains any string that does not begin with / and TERM does not match *name*, **tgetent** searches for *name* in /etc/termcap.

**tgetent** returns -1 if it could not open the terminal capability file, 0 if it could not find an entry for *name*, and 1 upon success.

**tgetnum** returns the value of the numeric capability whose name is *id*. It returns -1 if the terminal lacks the specified capability or it is not a numeric capability.

**tgetflag** returns 1 if the terminal has boolean capability whose name is *id*, 0 if it does not or it is not a boolean capability.

**tgetstr** copies and interprets the value of the string capability named by *id*. **tgetstr** expands instances in the string of \ and ^. It leaves the expanded string in the buffer indirectly pointed to by *area* and leaves the buffer's direct pointer pointing to the end of the expanded string; for example:

```
    tgetstr("cl", &ptr);
```

where *ptr* is a character pointer - not an array name. **tgetstr** returns a (direct) pointer to the beginning of the string.

# termcap

**tgoto** interprets the % in a cm string. It returns *cmstr* with the % sequences changed to the position indicated by *destcol* and *destline*. This function must have the external variables BC and UP set to the values of the bc and up capabilities; if the terminal lacks the capability, set the external variable to null. If **tgoto** cannot interpret all the % sequences in cm, it returns "OOPS."

**tgoto** avoids producing characters that might be misinterpreted by the terminal interface. If expanding a % sequence would produce null or control-d, the function will, if possible, send the cursor to the next line or column and use BC or UP to move to the correct location. Note that **tgoto** does not avoid producing tabs; a program must turn off the TAB3 feature of the terminal interface (see **termio**, Section 6). This is a good idea anyway: some terminals use the tab character as a nondestructive space.

**tputs** directs the output of a string returned by **tgetstr** or **tgoto**. This function must have the external variable PC set to the value of the pc capability; if the terminal lacks the capability, set the external variable to null. **tputs** interprets any delay at the beginning of the string. *Cp* is the string output; *affcnt* is the number of lines affected by the action (1 if "number of lines affected" doesn't mean anything); and *outc* points to a function that takes a single char argument, such as **putchar**, and outputs it.

## Files

/usr/lib/libtermcap.a - library

/etc/termcap - data base

## See Also

**ex** in Section 1; **curses**; **term** in Section 5.

## tmpfile

### Name

**tmpfile** - create a temporary file

### Format

```
#include <stdio.h>
FILE *tmpfile ()
```

### Description

The **tmpfile** function creates a temporary file using a name generated by **tmpnam**, and returns a corresponding FILE pointer. If the file cannot be opened, an error message is printed using **perror**, and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("w+").

### See Also

**creat**, **unlink** in Section 2; **fopen**, **mktemp**, **perror**, **tmpnam**.

# tmpnam

## Name

**tmpnam, tempnam** - create a name for a temporary file

## Format

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

## Description

These functions generate file names that can safely be used for a temporary file.

**tmpnam** always generates a file name using the path prefix defined as P_tmpdir in the <stdio.h> header file. If *s* is NULL, **tmpnam** leaves its result in an internal static area and returns a pointer to that area. The next call to **tmpnam** will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least L_tmpnam bytes, where L_tmpnam is a constant defined in <stdio.h>; **tmpnam** places its result in the array and returns *s*.

**tempnam** allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string that is not a name for an appropriate directory, the path prefix defined as P_tmpdir in the <stdio.h> header file is used. If that directory is not accessible, /tmp will be used as a last resort. This entire sequence can be upstaged by providing an environment variable TMPDIR in the user's environment, whose value is the name of the desired temporary-file directory.

## tmpnam

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

**tempnam** uses **malloc** to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from **tempnam** may serve as an argument to **free** (see **malloc**). If **tempnam** cannot return the expected result for any reason (such as if **malloc** failed), or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

Note that these functions generate a different file name each time they are called.

Note also that files created using these functions and either an **fopen** function or a **creat** system call are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use the **unlink** system call to remove the file when its use is ended.

### Known Problems

If called more than 17,576 times in a single process, these functions will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or **mktemp**, and the file names are chosen so as to render duplication by other means unlikely.

### See Also

**creat**, **unlink** in Section 2; **fopen**, **malloc**, **mktemp**, **tmpfile**.

# trig

## Name

**sin, cos, tan, asin, acos, atan, atan2** - trigonometric functions

## Format

```
#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** respectively return the sine, cosine, and tangent of their arguments, *x*, measured in radians.

**asin** returns the arcsine of *x*, in the range -pi/2 to pi/2.

**acos** returns the arccosine of *x*, in the range 0 to pi.

**atan** returns the arctangent of *x*, in the range -pi/2 to pi/2.

**atan2** returns the arctangent of *t/x*, in the range -pi to pi, using the signs of both arguments to determine the quadrant of the return value.

# trig

## Diagnostics

**sin**, **cos**, and **tan** lose accuracy when their arguments are 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 **matherr** function.

## See Also

**matherr**.

# tsearch

## Name

**tsearch, tfind, tdelete, twalk** - manage binary search trees

## Format

```
#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 pointer 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 is 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 that will be at the route of the new tree.

## tsearch

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 route being level zero.

The pointers to the *key* and the route 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 contining 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.

## tsearch

```
#include <search.h>
#include <stdio.h>

struct node {           /*pointers to these are stored in 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, &root,
               node_compare);
          /*adjust pointers, so we don't overwrite tree*/
          strptr += nodeptr->length + 1;
          nodeptr++;
     }
     twalk(root, print_node);
}
/*
     This routine compares two nodes, based on an
     alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
struct node *node1, node2;
{
     return strcmp(node1->string, node2->string);
}
/*
     This routine prints out a node, the first time
     twalk encounters it.
*/
void
print_node(node, order, level)
struct node **node;
VISIT order;
int level;
{
     if (order == preorder   order == leaf) {
          (void)printf("string = %20s, length = %d\n",
               (*node)->string, (*node)->length);
     }
}
```

## tsearch

### Diagnostics

A NULL pointer is returned by **tsearch** if there is not enough space available to create a new node.

A NULL pointer is returned by **tsearch**, **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.

### Cautions

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 visiting 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.

### Known Problems

If the calling function alters the pointer to the root, results are unpredictable.

### See Also

**bsearch**, **hsearch**, **lsearch**.

# ttyname

## Name

**ttyname, isatty** - find name of a terminal

## Format

```
char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;
```

## Description

The **ttyname** function 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.

## Known Problems

The return value points to static data whose content is overwritten by each call.

# ttyslot

## Name

**ttyslot** - find the slot in the utmp file of the current user

## Format

```
int ttyslot ()
```

## Description

The **ttyslot** function returns the index of the current user's entry in the /etc/utmp file. This is accomplished by actually scanning the file /etc/utmp for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1, or 2).

## Files

/etc/utmp

## 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.

## See Also

**getut, ttyname.**

## ungetc

### Name

**ungetc** - push character back into input stream

### Format

```
#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;
```

### Description

The **ungetc** function inserts the character *c* into the buffer associated with an input *stream*. The character *c* will be returned by the next **getc** call on that *stream*. **ungetc** returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered. In the case that *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

If *c* equals EOF, **ungetc** does nothing to the buffer and returns EOF.

The **fseek** function erases all memory of inserted characters.

### Diagnostics

**ungetc** returns EOF if it cannot insert the character.

### See Also

**fseek**, **getc**, **setbuf**.

# vprintf

## Name

**vprintf, vfprintf, vsprintf** - print formatted output of a varargs argument list

## Format

```
#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

## Description

**vprintf, vfprintf,** and **vsprintf** are the same as **printf, fprintf,** and **sprintf,** except that instead of being called with a variable number of arguments, they are called with an argument list defined by **varargs** (see Section 5).

# vprintf

## Example

The following demonstrates how **vprintf** could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
*       error should be called like
*               error(function_name, format, arg1, arg2...);
*/
/*VARARGS0*/
void
error(va_alist)
/*Note that the function_name and format arguments cannot be
*       separately declared because of the definition of
*       varargs.
*/
va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);
        /*print out name of function causing error*/
        (void)fprintf(stderr, "ERROR in %s:",
                va_args(args, char *));
        fmt = va_arg(args, char *);
        /*print out remainder of message*/
        (void)vfprintf(fmt, args);
        va_end(args);
        (void)abort();
}
```

## See Also

**printf**; varargs in Section 5.

# wmgetid

## Name

**wmgetid** - get window ID

## Format

```
#include <oa/wm.h>

int wmgetid (fildes);
int fildes;
```

## Description

The **wmgetid** function returns the window ID associated with the file descriptor *fildes*. A window ID is a positive integer that identifies the window associated with the file descriptor. The ID is passed to other window management library functions to identify the particular window being acted upon. The only way to get a valid window ID is from a window management library call; do not use a value obtained in any other way.

To get all the window IDs for a terminal, use the layout structure written by **wmlayout** or **wmop**. To associate a file descriptor with a different window, use **wmsetid**.

**wmgetid** fails if one or more of the following are true:

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

The indicated file does not represent a terminal, or the terminal cannot support window management. [ENOTTY]

The window manager is not running on the terminal. [ENOENT]

## wmgetid

### Files

/dev/tty
/usr/lib/libwm.a - window management library

### Diagnostics

If successful, **wmgetid** returns the window ID associated with *fildes*. Otherwise, -1 is returned and **errno** is set to indicate the error.

### See Also

**wm** in Section 1; **wmop, wmlayout, wmsetid.**

# wmlayout

## Name

**wmlayout** - get terminal's window layout

## Format

```
#include <oa/wm.h>

int wmlayout (fildes, layout)
int fildes;
struct wm_layout *layout;
```

## Description

The **wmlayout** function fetches a description of the screen layout of a terminal under window management. *Fildes* is a file descriptor associated with the terminal's special file by a **creat, dup, fcntl,** or **open** system call; the association of *fildes* with a particular window is not used. *Layout* points to an area that is to receive the description. Before calling **wmlayout**, a program must set *layout->maxwcount* to indicate the number of window descriptions the area can accommodate; the constant WM_MAX gives the number of windows currently permitted. The description consists of the following data structures:

```
struct wm_layout {
        int      cwindowid;
        short    maxwcount;
        short    wcount;
        struct   wm_layoutw[WM_MAX];
        };

struct wm_wlayout {
        int      windowid;
        short    pwindowid;
        short    startrow;
        short    startcolumn;
        short    drows;
        short    dcolumns;
        short    syncrow;
        short    synccolumn;
        short    vrows;
        short    vcolumns;
        short    crow;
        short    ccolumn;
        char     reserved[6];   /*must be 0*/
        }
```

## wmlayout

Here are the meanings of the fields in a *wm_layout* structure:

| | |
|---|---|
| *cwindowid* | The window ID of the active window. |
| *maxwcount* | Number of window descriptions this structure has room for. Normally set to WM_MAX so as to get all of them. |
| *wcount* | Number of windows currently on terminal. |
| *w* | Array of individual window descriptions. |

Here are the meanings of the fields in a *wm_wlayout* structure:

| | |
|---|---|
| *windowid* | The window ID. |
| *pwindowid* | The physical window ID. Meant only for window management internal use. |
| *startrow* | Starting physical row of the window (the tag line is on the row before). |
| *startcolumn* | Starting physical column of the window. Currently this value is always 1. |
| *drows* | The number of displayed rows in the window. Note that the tag line is not counted in this value. |
| *dcolumns* | The number of displayed columns in the window. Currently this value is always 80. |
| *syncrow* | Virtual display row that corresponds to the first row of the window. |
| *synccolumn* | Virtual display column that corresponds to the first column of the window. Currently this value is always 1. |
| *vrows* | Number of rows in virtual display. |
| *vcolumns* | Number of columns in virtual display. Currently this value is always 80. |
| *crow* | The current cursor row number. |
| *ccolumn* | The current cursor column number. |
| *reserved* | Always zeros. |

Rows and columns are numbered from 1.

# wmlayout

A window ID is a positive integer that identifies the window associated with the file descriptor. The ID is passed to other window management library functions to identify the particular window being acted upon. The only way to get a valid window ID is from a window management library call; do not use a value obtained any other way.

Currently, physical windows always start in column zero and physical windows and virtual displays are always 80 columns wide.

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

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

The indicated file does not represent a terminal, or the terminal cannot support window management. [ENOTTY]

The structure pointed to by *windowreq* is invalid. [EINVAL]

The window manager is not running on the terminal. [ENOENT]

## Files

/usr/lib/libwm.a - window management library

/dev/tty*

## Diagnostics

**wmlayout** returns 0 if successful; otherwise, the function returns -1 and sets **errno** to indicate the error.

## See Also

**wm** in Section 1; **wmgetid, wmsetid, wmop.**

# wmop

## Name

**wmop** - window management operations

## Format

```
#include <oa/wm.h>

int wmop (fildes, windowreq, layout)
int fildes;
struct wm_request *windowreq;
struct wm_layout *layout;
```

## Description

**wmop** manipulates windows on a terminal under window management. It is normally used by application programs. *Fildes* is a file descriptor associated with the terminal's special file by a **creat, dup, fctntl,** or **open** system call.; the association of *fildes* with a particular window is not used. *Windowreg* is a pointer to a structure that describes the operation. *Layout* is an optional pointer to a layout structure of the type used by **wmlayout**; if present, the structure is filled with the new description of the window.

The request structure is defined as follows:

```
struct wm_request {
        int     request;
        int     windowid;
        int     (*notify)()
        short   startrow;
        short   startcolumn;
        short   drows;
        short   dcolumns;
        short   syncrow;
        short   synccolumn;
        short   vrows;
        short   vcolumns;
        short   crow;
        short   ccolumn;
};
```

## wmop

Only two fields in the request structure are used by all operations:

- *Request* specifies the operations desired. See the operation constants, described below.
- *Windowid* specifies a window, usually with a window ID returned by a previous **wmop**, **wmlayout**, or **wmgetid**. The only way to get a valid window ID is from a window management library call; do not use a value obtained any other way. If the operations do not include WM_CREATE (create a new window), *windowid* is a window ID that specifies the single window to which the operations apply. If the operations do include WM_CREATE, *windowid* must be either a window ID, indicating the window that yields space for the new window, or 0, a value with special meanings described under WM_CREATE and WM_START; the other operations apply to the new window.

| | |
|---|---|
| WM_CREATE | Create a new window. Other operations describe the new window's characteristics; if no other operations are specified with WM_CREATE, the new window has the following characteristics:<br><br>The new window occupies the bottom half of the window specified by *windowid*. If *windowid* is 0, the new window occupies the bottom half of the active window.<br><br>The new window's virtual display is 29 lines long.<br><br>The cursor is on the first line of the new window's virtual display, which is also the first line of the new window.<br><br>The user is permitted to split the new window only if the old window permitted user splits. See WM_SPLIT. |
| WM_DESTROY | Destroy the window. If the window is the top window, the destroyed window's rows go to the window below; otherwise the destroyed window's rows go to the window above. If the destroyed window was the active window, the window that gets the destroyed window's rows is activated. |

## wmpo

| | |
|---|---|
| WM_DSIZE | Change the window size. The operation can be modified by WM_DRSIZE; this description assumes it is not. The window size, which does not include the window's tag line, can vary from 0 to 26. *Drows* specifies the new window size.<br><br>If WM_DSIZE is specified with WM_CREATE, *drows* specifies the new window's size. |
| WM_DRSIZE | Modifies WM_DSIZE so the *drows* specifies an offset relative to the current value, rather than an absolute size. *Drows* can be negative.<br><br>If WM_DSIZE and WM_DRSIZE are specified with WM_CREATE, *drows* specifies the new window's size relative to the size of the old window. Thus, in this case, *drows* must be negative. |
| WM_DSTART | Set the starting row of the window (not the tag line, which is automatically on the row before). This operation may be modified by WM_DRSTART; this description assumes it is not. Rows are numbered from 1, and a window can start on any row from 2 to 28. *Startrow* specifies the new starting row.<br><br>If WM_DSTART is specified with WM_CREATE and *windowid* is 0, *startrow* specifies the new window's starting position on the screen, without reference to an existing window. |
| WM_DRSTART | Modifies WM_DSTART so the *startrow* specifies an offset relative to the current value, rather than an absolute starting row. *Startrow* can be negative.<br><br>If WM_DSTART and WM_DRSTART are specified with WM_CREATE, *startrow* must be non-negative; the new window starts *startrow* rows after the start of the old window. If *startrow* is 0, the new window takes the top portion of the old window's rows instead of the bottom. If *startrow* is positive, WM_DSIZE is ineffective: the size of the new window is dictated by the size of the old. |
| WM_VSIZE | Set the virtual window size to *vrows* long. The operations can be modified by WM_VRSIZE. In any case, the virtual display must be 1 to 28 rows long.<br><br>If the virtual display is shortened past the cursor, the cursor must be moved to within the new virtual display end. If the WM_CURSOR operation is not specified at the same time, the terminal moves the cursor to the new last line of the virtual display. |

## wmop

| | |
|---|---|
| WM_VRSIZE | Modifies WM_VSIZE so that *vrows* is an offset to the present value. *Vrows* can be negative. |
| WM_VSTART | Synchronize the window and its virtual display by making virtual display row *syncrow* (numbered from 1) the first row on the window. This operation can be modified by WM_VRSTART. The window manager will modify a WM_VSTART operation as necessary to keep the window from extending past the bottom of the virtual display. If the cursor is visible, the terminal software will modify a WM_VSTART operation as necessary to keep the cursor in the window. |
| WM_VRSTART | Modify WM_VSTART so that *syncrow* is an offset to the present value. *Syncrow* can be negative. |
| WM_SELECT | Make the window the active window. |
| WM_DESELECT | If the window is the active window, make another window the active window: if the designated window is the top window, the window below; otherwise, the window above. |
| WM_CURSOR | Position the cursor on row *crow*. |
| WM_SPLIT | Enable change of splitting permission. Used in conjunction with WM_NSPLIT. If WM_SPLIT is specified alone, the user can split the window as long as the terminal can handle another window. If WM_SPLIT and WM_NSPLIT are specified together, the SPLIT key is ineffective when the window is active. |
| WM_NSPLIT | Disable window split. Always used in conjunction with WM_SPLIT. |
| WM_NOTIFY | *Notify* is a notify procedure. Set *notify* to (int(*)())0 to disable an existing notify procedure. The calling process will be interrupted and *notify* will be called if any other process or the user changes the status of the window. Window status includes window size, location, and whether it is active; it does not include cursor location. |

Currently, all windows and displays must begin in column 0 and be 80 columns wide.

# wmop

**wmop** fails if one or more of the following are true:

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

The indicated file does not represent a terminal, or the terminal cannot support window management. [ENOTTY]

The structure pointed to by *windowreq* is invalid. [EINVAL]

The window manager is not running on the terminal. [ENOENT]

## Files

/dev/tty*
/usr/lib/libwm.a - window management library

## Diagnostics

If the operations were successful, the window ID of the affected window (the new window if one was created) is returned. Otherwise, -1 is returned and **errno** is set to indicate the error.

## Cautions

Use **wmop** conservatively and with extreme care. Indescriminate use by programs competing for window space can result in race conditions and screen image instability.

The window manager and terminal software silently enforce basic consistency. A program must not make assumptions about what the window looks like after a successful **wmop**; instead, it must examine the new **wmlayout** structure to find out what actually happened.

## See Also

**wmgetid, wmlayout, wmsetid, ferror** to get file descriptor for terminal accessed with standard I/O package.

# wmsetid

## Name

**wmsetid, wmsetids** - associate a file descriptor with a window

## Format

```
#include <oa/wm.h>

int wmsetid (fildes, windowid)
int windowid;
int fildes;

int wmsetids (fildes, windowid)
int windowid;
int fildes;
```

## Description

The **wmsetid** and **wmsetids** functions change the window with which a file descriptor is associated. *Fildes* must be a file descriptor open to a terminal on which the window manager is running. *Fildes* becomes associated with the window (on the same terminal) indicated by *windowid*, which must be a window ID obtained from a previous **wmgetid**, **wmlayout**, or **wmop** call.

If a program performs a **wmsetid** on an inherited file descriptor, all processes that have inherited and use the same file descriptor and the process they inherited it from are affected. By convention, 0 (equivalent to fileno(stdin)), 1 (equivalent to (fileno(stdout)), and 2 (equivalent to fileno(stderr)) are inherited file descriptors. The following code closes and reopens them so that a **wmsetid** on them doesn't affect other processes. It should be executed before terminal input/output begins:

```
tty=ttyname(0);
close(0);
close(1);
open(tty, O_RDWR);
close(2);
dup(0);
dup(0);
```

# wmsetid

Be sure to complete buffered terminal output before switching windows. See **fclose** if you use the standard input/output package.

**wmsetid** and **wmsetids** are different only when executed by a process group leader. If the process group leader calls **wmsetids** and the specified window is not already a controlling window for another process group, the specified window becomes the process group's controlling window. (For more details on control windows, see **termio** and **window**, both in Section 6). **wmsetid** never changes the controlling window under any circumstances.

**wmsetid** and **wmsetids** fail if one or more of the following are true:

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

The indicated file does not represent a terminal, or the terminal cannot support window management. [ENOTTY]

The structure pointed to by *windowreq* is invalid. [EINVAL]

The window manager is not running on the terminal. [ENOENT]

## Files

/dev/tty*

/usr/lib/libwm.a - window management library

## Diagnostics

A non-negative value indicates success: 0 if the file descriptor wasn't associated with a window before the call, the old window otherwise. On error, -1 is returned and **errno** is set.

## See Also

**wm** in Section 1; **wmop, wmlayout, wmgetid, ferror.**

# Permuted Index

This index includes entries for all pages of all four volumes of this guide. The entries themselves are based on the one-line descriptions or titles found in the **Name** portion of each manual entry; the significant words (keywords) of these descriptions are listed alphabetically down the center of the index.

The permuted index is a keyword-in-context index that has three columns. To use the index, read the center column to look up specific commands by name or by subject topics. Note that the entry may begin in the left column or wrap around and continue into the left column. A period (.) marks the end of the entry, and a slash (/) indicates where the entry is continued or truncated. The right column gives the manual entry under which the command or subject is described; following each manual entry name is the section number, in parentheses, in which that entry can be found.

| | | |
|---|---|---|
| sadp: disk | access profiler. | **sadp(1)** |
| common object file | access routines. ldfcn: | **ldfcn(4)** |
| file systems for optimal | access time. /copy | **dcopy(1)** |
| locking: exclusive | access to regions of a/ | **locking(2)** |
| /endutent, utmpname: | access utmp file entry. | **getut(3)** |
| access: determine | accessibility of a file. | **access(2)** |
| or disable process | accounting. /enable | **acct(2)** |
| /manipulate connect | accounting records. | **fwtmp(1)** |
| process accounting. | acct: enable or disable | **acct(2)** |
| sin, cos, tan, asin, | acos, atan, atan2:/ | **trig(3)** |
| killall: kill all | active processes. | **killall(1)** |
| sag: system | activity graph. | **sag(1)** |
| sa1, sa2, sadc: system | activity report package. | **sar(1)** |
| File Processor system | activity reporter. | **fpsar(1)** |
| sar: system | activity reporter. | **sar(1)** |
| SCCS file editing | activity. /print current | **sact(1)** |
| process data and system | activity. /report | **timex(1)** |
| | adb: absolute debugger | **adb(1)** |
| BTOS queue. quAdd: | add a new entry to a | **quadd(3)** |
| putenv: change or | add value to/ | **putenv(3)** |
| administer SCCS files. | admin: create and | **admin(1)** |
| admin: create and | administer SCCS files. | **admin(1)** |
| alarm: set a process | alarm clock. | **alarm(2)** |
| alarm clock. | alarm: set a process | **alarm(2)** |

| | | |
|---|---|---|
| for sendmail. | aliases: aliases file | **aliases(5)** |
| sendmail. aliases: | aliases file for | **aliases(5)** |
| /ofDelete: | allocate BTOS files. | **ofcreate(3)** |
| data segment space | allocation. /change | **brk(2)** |
| calloc: main memory | allocator. /realloc, | **malloc(3)** |
| fast main memory | allocator. /mallinfo: | **malloc(3) (fast version)** |
| accept, reject: | allow/prevent LP/ | **accept(1)** |
| brc, bcheckrc, rc, | allrc, conrc: system/ | **brc(1)** |
| running process/ renice: | alter priority of | **renice(1)** |
| sort: sort | and/or merge files. | **sort(1)** |
| and link editor output. | a.out: common assembler | **a.out(4)** |
| Processor number. | apnum: print Application | **apnum(1)** |
| number. apnum: print | Application Processor | **apnum(1)** |
| console: control | Application Processor/ | **console(1)** |
| /a process on a specific | Application Processor. | **spawn(1)** |
| /a process on a specific | Application Processor. | **spawn(3)** |
| /to commands and | application programs. | **intro(1)** |
| code. exServeRq: | appropriate a request | **exserverq(2)** |
| maintainer for portable/ | ar: archive and library | **ar(1)** |
| format. | ar: common archive file | **ar(4)** |
| arithmetic/ bc: | arbitrary-precision | **bc(1)** |
| maintainer for/ ar: | archive and library | **ar(1)** |
| cpio: format of cpio | archive | **cpio(4)** |
| ar: common | archive file format. | **ar(4)** |

| | | |
|---|---|---|
| header of a member of an | archive file. /archive | **ldahread(3)** |
| /convert object and | archive files to common/ | **convert(1)** |
| ldahread: read the | archive header of a/ | **ldahread(3)** |
| tar: tape file | archiver. | **tar(1)** |
| maintainer for portable | archives. /and library | **ar(1)** |
| cpio: copy file | archives in and out. | **cpio(1)** |
| varargs: handle variable | argument list. | **varargs(5)** |
| /output of a varargs | argument list. | **vprintf(3)** |
| xargs: construct | argument list(s) and/ | **xargs(1)** |
| /get option letter from | argument vector. | **getopt(3)** |
| expr: evaluate | arguments as an/ | **expr(1)** |
| echo: echo | arguments. | **echo(1)** |
| bc: arbitrary-precision | arithmetic language. | **bc(1)** |
| expr: evaluate arguments | as an expression. | **expr(1)** |
| | as: assembler. | **as(1)** |
| ascii: map of | ASCII character set. | **ascii(5)** |
| hd: hexadecimal and | ascii file dump. | **hd(1)** |
| character set. | ascii: map of ASCII | **ascii(5)** |
| long integer and base-64 | ASCII string. /between | **a64l(3)** |
| atof: convert | ASCII string to/ | **atof(3)** |
| date/ /localtime, gmtime, | asctime, tzset: convert | **ctime(3)** |
| sin, cos, tan, | asin, acos, atan, atan2:/ | **trig(3)** |
| help: | ask for help. | **help(1)** |

| | | |
|---|---|---|
| compiler/interp reter/ | bs: a | bs(1) |
| sorted table | bsearch: binary search a | bsearch(3) |
| /ofDlDir, ofReadDirSector: | BTOS directory functions. | ofdir(3) |
| ofWrite: Input/output on a | BTOS file. ofRead, | ofread(3) |
| ofRename: rename a | BTOS file. | ofrename(3) |
| ofSetFileStatus: | BTOS File Status. | ofstatus(3) |
| ofcopy: copy to or from the | BTOS file system. | ofcopy(1) |
| directories. ofls: list | BTOS files and | ofls(1) |
| /ofDelete: Allocate | BTOS files. | ofcreate(3) |
| ofed, ofvi: edit | BTOS files. | ofeditors(1) |
| ofCloseAllFiles: Access | BTOS files. /ofCloseFile. | ofopenfile(3) |
| interpreter for interactive | BTOS JCL. ofcli: command line | ofcli(1) |
| CENTIX kernel and copy it to | BTOS. mkboot: reformat | mkboot(1) |
| quAdd: add a new entry to a | BTOS queue. | quadd(3) |
| quReadKeyed: examine | BTOS queue. quReadNext. | quread(3) |
| quRemove: take back a | BTOS queue request. | quremove(3) |
| stdio: standard | buffered input/output package. | stdio(3) |
| setbuf, setvbuf: assign | buffering to a stream | setbuf(3) |
| mknod: | build special file. | mknod(1) |
| swapshort, swaplong: translate | byte orders to Motorola/Intel. | swapshort(3) |
| swab: swap | bytes. | swab(3) |

| | | |
|---|---|---|
| of a file. | chown: change owner and group | **chown(2)** |
| group. | chown, chgrp: change owner or | **chown(1)** |
| directory. | chroot: change root | **chroot(2)** |
| for a command. | chroot: change root directory | **chroot(1)** |
| isgraph, iscntrl, isascii: | classify characters. /isprint, | **ctype(3)** |
| uuclean: uucp spool directory | clean-up. | **uuclean(1)** |
| screen. | clear: clear terminal | **clear(1)** |
| clri: | clear i-node. | **clri(1)** |
| clear: | clear terminal screen. | **clear(1)** |
| status./ ferror, feof, | clearerr, fileno: stream | **ferror(3)** |
| exRespond: send a message to a | client. | **exrespond(2)** |
| set a process alarm | clock. alarm: | **alarm(2)** |
| cron: | clock daemon. | **cron(1)** |
| used. | clock: report CPU time used. | **clock(3)** |
| ldclose, ldaclose: | close a common object file. | **ldclose(3)** |
| close: | close a file descriptor. | **close(2)** |
| descriptor. | close: close a file | **close(2)** |
| fclose, fflush: | close or flush a stream. | **fclose(3)** |
| | clri: clear i-node. | **clri(1)** |
| | cmp: compare two files. | **cmp(1)** |
| appropriate a request | code. exServeRq: | **exserverq(2)** |
| line-feeds. | col: filter reverse | **col(1)** |
| deltas. | comb: combine SCCS | **comb(1)** |
| comb: | combine SCCS deltas. | **comb(1)** |

| | | |
|---|---|---|
| common to two sorted files. | comm: select or reject lines | **comm(1)** |
| nice: run a | command at low priority. | **nice(1)** |
| change root directory for a | command. chroot: | **chroot(1)** |
| env: set environment for | command execution. | **env(1)** |
| uux: remote system | command execution. | **uux(1)** |
| quits. nohup: run a | command immune to hangups and | **nohup(1)** |
| interactive BTOS JCL. ofcli: | command line interpreter for | **ofcli(1)** |
| getyopt: parse | command options. | **getopt(1)** |
| locate executable file for | command. path: | **path(1)** |
| shell, the standard/ restricted | command programming language. | **sh(1)** |
| data and/ timex: time a | command; report process | **timex(1)** |
| system: issue a shell | command. | **system(3)** |
| test: condition evaluation | command. | **test(1)** |
| time: time a | command. | **time(1)** |
| argument list(s) and execute | command. xargs: construct | **xargs(1)** |
| intro: introduction to | commands and applicaton/ | **intro(1)** |
| at, batch: execute | commands at a later/ | **at(1)** |
| install: install | commands. | **install(1)** |
| cdc: change the delta | commentary of an SCCS delta. | **cdc(1)** |
| ar: | common archive file format. | **ar(4)** |

| | | |
|---|---|---|
| ioctl: | control device. | **ioctl(2)** |
| fcntl: file | control. | **fcntl(2)** |
| init, icode, telinit: process | control initialization. | **init(1)** |
| msgctl: message | control operations. | **msgctl(2)** |
| semctl: semaphore | control operations | **semctl(2)** |
| shmctl: shared memory | control operations. | **shmctl(2)** |
| fcntl: file | control options. | **fcntl(5)** |
| uucp status inquiry and job | control. uustat: | **uustat(1)** |
| vc: version | control. | **vc(1)** |
| interface. tty: | controlling terminal | **tty(6)** |
| RS-232 channels. tp: | controlling terminal's local | **tp(6)** |
| terminals. term: | conventional names for | **term(5)** |
| units: | conversion program. | **units(1)** |
| dd: | convert and copy a file. | **dd(1)** |
| floating-point number. atof: | convert ASCII string to | **atof(3)** |
| integers and/ l3tol, ltol3: | convert between 3-byte | **l3tol(3)** |
| and base-64 ASCII/ a64l, l64a: | convert between long integer | **a64l(3)** |
| and archive files to/ | convert: convert object | **convert(1)** |
| /gmtime, asctime, tzset: | convert date and time to/ | **ctime(3)** |
| to string. ecvt, fcvt, gcvt | convert floating-point nubmner | **ecvt(3)** |
| scanf, fscanf, sscanf: | convert formatted input. | **scanf(3)** |
| archive files/ convert: | convert object and | **convert(1)** |
| strtod, atof: | convert string to/ | **strtod(3)** |

| | | |
|---|---|---|
| sleep: suspend executin for | interval. | sleep(3) |
| commands and application/ | intro: introduction to | intro(1) |
| formats. | intro: introduction to file | intro(4) |
| miscellany. | intro: introduction to | intro(5) |
| files. | intro: introduction to special | intro(6) |
| subroutines and libraries. | intro: introduction to | intro(3) |
| calls and error numbers. | intro: introduction to system | intro(2) |
| applicaton programs. intro: | introduction to commands and | intro(1) |
| intro: | introduction to file formats. | intro(4) |
| intro: | introduction to miscellany. | intro(5) |
| intro: | introduction to special files. | intro(6) |
| and libraries. intro: | introduction to subroutines | intro(3) |
| and error numbers. intro: | introduction to system calls | intro(2) |
| ncheck: generate names from | i-numbers. | ncheck(1) |
| in large files and direct | I/O. /performance improvement | pilf(5) |
| | ioctl: control device. | ioctl(2) |
| abort: generate an | IOT fault. | abort(3) |
| semaphore set or shared/ | ipcrm: remove a message queue, | ipcrm(1) |
| communication facilities/ | ipcs: report inter-process | ipcs(1) |
| /islower, isdigit, isxdigit, | isalnum, isspace, ispunct,/ | ctype(3) |
| isdigit, isxdigit, isalnum,/ | isalpha, isupper, islower, | ctype(3) |

| | | |
|---|---|---|
| set and get file creation | mask. umask: | **umask(2)** |
| table. master: | master device information | **master(4)** |
| information table. | master: master device | **master(4)** |
| regular expression commpile and | match routines. regexp: | **regexp(5)** |
| math: | math functions and constants. | **math(5)** |
| constants. | math: math functions and | **math(5)** |
| function. | matherr: error-handling | **matherr(3)** |
| processor type. | mc68k, pdp11, u3b, vax: | **machid(1)** |
| | mem, kmem: core memory. | **mem(6)** |
| memcpy, memset: memory/ | memccpy, memchr, memcmp, | **memory(3)** |
| memset: memory/ memccpy, | memchr, memcmp, memcpy, | **memory(3)** |
| operations. memccpy, memchr, | memcmp, memcpy, memset: memory | **memory(3)** |
| memccpy, memchr, memcmp, | memcpy, memset: memory/ | **memory(3)** |
| free, realloc, calloc: main | memory allocator. malloc, | **malloc(3)** |
| mallopt, mallinfo: fast main | memory allocator. /calloc, | **malloc(3) (fast version)** |
| shmctl: shared | memory control operations. | **shmctl(2)** |
| queue, semaphore set or shared | memory id. /remove a message | **ipcrm(1)** |
| mem, kmem: core | memory. | **mem(6)** |
| memcmp, memcpy, memset: | memory operations. /memchr, | **memory(3)** |
| shmop: shared | memory operations. | **shmop(2)** |
| text, or data in | memory. /lock process, | **plock(2)** |
| shmget: get shared | memory segment. | **shmget(2)** |

| | | |
|---|---|---|
| setjmp, longjmp: | non-local goto. | **setjmp(3)** |
| drand48, erand48, lrand48, | nrand48, mrand48, jrand48,/ | **drand48(3)** |
| null: the | null file. | **null(6)** |
| | null: the null file. | **null(6)** |
| nl: line | numbering filter. | **nl(1)** |
| to/ convert: convert | object and archive files | **convert(1)** |
| ldfcn: common | object file access routines. | **ldfcn(4)** |
| dump selected parts of an | object file. dump: | **dump(1)** |
| ldopen, ldaopen: open a common | object file for reading. | **ldopen(3)** |
| number entries of a common | object file function. /line | **ldlread(3)** |
| ldaclose: close a common | object file. ldclose, | **ldclose(3)** |
| the file header of a common | object file. ldfhread: read | **ldfhread(3)** |
| of a section of a common | object file. /number entries | **ldlseek(3)** |
| file header of a common | object file. /to the optional | **ldohseek(3)** |
| of a section of a common | object file. /entries | **ldrseek(3)** |
| header of a common | object file. /section | **ldshread(3)** |
| section header of a common | object file. /indexed/named | **ldsseek(3)** |
| symbol table entry of a common | object file. /the index of a | **ldtbindex(3)** |
| symbol table entry of a common | object file. /read an indexed | **ldtbread(3)** |
| the symbol table of a common | object file. /seek to | **ldtbseek(3)** |
| number entries in a common | object file. linenum: line | **linenum(4)** |

| | | |
|---|---|---|
| ofReadDirSector: BTOS/ | ofCrDir, ofDlDir, | **ofdir(3)** |
| ofDelete: Allocate BTOS/ | ofCreate, ofChangeFileLength, | **ofcreate(3)** |
| ofCreate, ofChangeFileLength, | ofDelete: Allccate BTOS files. | **ofcreate(3)** |
| directory functions. ofCrDir, | ofDlDir, ofReadDirSector: BTOS | **ofdir(3)** |
| | ofed, ofvi: edit BTOS files. | **ofeditors(1)** |
| ofSetFileStatus: BTOS File/ | ofGetFileStatus, | **ofstatus(3)** |
| directories. | ofls: list BTOS files and | **ofls(1)** |
| ofCloseAllFiles: Access BTOS/ | ofOpenFile, ofCloseFile, | **ofopenfile(3)** |
| on a BTOS file. | ofRead, ofWrite: Input/output | **ofread(3)** |
| directory/ ofCrDir, ofDlDir, | ofReadDirSector: BTOS | **ofdir(3)** |
| | ofRename: rename a BTOS file. | **ofrename(3)** |
| Status. ofGetFileStatus, | ofSetFileStatus: BTOS File | **ofstatus(3)** |
| ofed, | ofvi: edit BTOS files. | **ofeditors(1)** |
| BTOS file. ofRead, | ofWrite: Input/output on a | **ofread(3)** |
| reading. ldopen, ldaopen: | open a common object file for | **ldopen(3)** |
| fopen, freopen, fdopen: | open a stream. | **fopen(3)** |
| dup: duplicate an | open file descriptor. | **dup(2)** |
| open: | open for reading or writing. | **open(2)** |
| writing. | open: open for reading or | **open(2)** |
| profiler. prf: | operating system | **prf(6)** |
| prfdc, prfsnap, prfpr: | operating system/ | **profiler(1)** |
| memcmp, memcpy, memset: memory | operations. memccpy, memchr, | **memory(3)** |

| | | |
|---|---|---|
| | setuname: set name of system. | **setuname(1)** |
| /getutid, getutline, pututline, | setutent, endutent, utmpname:/ | **getut(3)** |
| stream. setbuf, | setvbuf: asign buffering to a | **setbuf(3)** |
| data in a/ sputl, | sgetl: access long integer | **sputl(3)** |
| standard/restricted command/ | sh, rsh: shell, the | **sh(1)** |
| operations. shmctl: | shared memory control | **shmctl(2)** |
| queue, semaphore set or | shared memory id. /a message | **ipcrm(1)** |
| shmop: | shared memory operations. | **shmop(2)** |
| shmget: get | shared memory segment. | **shmget(2)** |
| system: issue a | shell command. | **system(3)** |
| cprofile: setting up a C | shell environment at/ | **cprofile(4)** |
| conrc: system initialization | shell scripts. /rc, allrc, | **brc(1)** |
| command programming/ sh, rsh: | shell, the standard/restricted | **sh(1)** |
| operations. | shmctl: shared memory control | **shmctl(2)** |
| segment. | shmget: get shared memory | **shmget(2)** |
| operations. | shmop: shared memory | **shmop(2)** |
| processing. | shutdown, halt: terminate all | **shutdown(1)** |
| program. sdiff: | side-by-side difference | **sdiff(1)** |
| login: | sign on. | **login(1)** |
| pause: suspend process until | signal. | **pause(2)** |
| whaqt to do upon receipt of a | signal. signal: specify | **signal(2)** |
| upon receipt of a signal. | signal: specify what to do | **signal(2)** |

| | | |
|---|---|---|
| | tee: pipe fitting. | **tee(1)** |
| initialization. init, icode, | telinit: process control | **init(1)** |
| temporary file. tmpnam, | tempnam: create a name for a | **tmpnam(3)** |
| tmpfile: create a | temporary file. | **tmpfile(3)** |
| tempnam: create a name for a | temporary file. tmpnam, | **tmpnam(3)** |
| terminals. | term: conventional names for | **term(5)** |
| term: format of compiled | term file. . | **term(4)** |
| file. | term: format of compiled term | **term(4)** |
| data base. | termcap: terminal capability | **termcap(4)** |
| termcap: | terminal capability data base. . | **termcap(4)** |
| terminfo: | terminal capability data base. | **terminfo(4)** |
| console: console | terminal. | **console(6)** |
| ct: spawn getty to a remote | terminal. | **ct(1)** |
| generate file name for | terminal. ctermid: | **ctermid(3)** |
| tdl: rs232 | terminal download. | **tdl(1)** |
| /terminal interface, and | terminal environment. | **tset(1)** |
| /tgetstr, tgoto, tputs: | terminal independent/ | **termcap(3)** |
| terminal/ tset: set terminal, | terminal interface, and | **tset(1)** |
| termio: general | terminal interface. | **termio(6)** |
| tty: controlling | terminal interface. | **tty(6)** |
| dial: establish an out-going | terminal line connection. | **dial(3)** |

| | | |
|---|---|---|
| ex: | text editor. | **ex(1)** |
| ex for casual/ edit: | text editor (variant of | **edit(1)** |
| change the format of a | text file. newform: | **newform(1)** |
| fspec: format specification in | text files. | **fspec(4)** |
| plock: lock process | text, or data in memory. | **plock(2)** |
| more, page: | text persual. | **more(1)** |
| strings: extract the ASCII | text strings in a file. | **strings(1)** |
| binary search types. tsearch, | tfind, tdelete, twalk: manage | **tsearch(3)** |
| tgetstr, tgoto, tputs:/ | tgetent, tgetnum, tgetflag, | **termcap(3)** |
| tputs:/ tgetent, tgetnum, | tgetflag, tgetstr, tgoto, | **termcap(3)** |
| tgoto, tputs:/ tgetent, | tgetnum, tgetflag, tgetstr, | **termcap(3)** |
| tgetent, tgetnum, tgetflag, | tgetstr, tgoto, tputs:/ | **termcap(3)** |
| /tgetnum, tgetflag, tgetstr, | tgoto, tputs: terminal/ | **termcap(3)** |
| | tic: terminfo compiler. | **tic(1)** |
| data and system/ timex: | time a command; report process | **timex(1)** |
| time: | time a command. | **time(1)** |
| commands at a later | time. /batch: execute | **at(1)** |
| environment at login | time. /up a C shell | **cprofile(4)** |
| systems for optimal access | time. dcopy: copy file | **dcopy(1)** |
| | time: get time. | **time(2)** |
| profil: execution | time profile.. | **profil(2)** |
| up an environment at login | time. profile: setting | **profile(4)** |

Title: ______________________________

Form Number: ______________________ Date: ____________

Burroughs Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information.

Please check type of suggestion: ☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments: ______________________________

______________________________

______________________________

______________________________

Name ______________________________

Title ______________________________

Company ______________________________

Address ______________________________
Street City State Zip

Telephone Number (     ) ______________________________
Area Code

Title: ______________________________

Form Number: ______________________ Date: ____________

Burroughs Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information.

Please check type of suggestion: ☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments: ______________________________

______________________________

______________________________

______________________________

Name ______________________________

Title ______________________________

Company ______________________________

Address ______________________________
Street City State Zip

Telephone Number (     ) ______________________________
Area Code

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY CARD**

FIRST CLASS PERMIT NO. 817 DETROIT, MI 48232

POSTAGE WILL BE PAID BY ADDRESSEE

Burroughs Corporation
Production Services – East
209 W. Lancaster Avenue
Paoli, Pa 19301 USA

**ATTN: Corporate Product Information**

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY CARD**

FIRST CLASS PERMIT NO. 817 DETROIT, MI 48232

POSTAGE WILL BE PAID BY ADDRESSEE

Burroughs Corporation
Production Services – East
209 W. Lancaster Avenue
Paoli, Pa 19301 USA

**ATTN: Corporate Product Information**