# A/UX® Programmer's Reference

## Sections 2 and 3(A-L)

030-0784

APPLE COMPUTER, INC.



Apple Computer, Inc.
20525 Mariani Ave.
Cupertino, California 95014
(408) 996-1010



Simultaneously published in the United States and Canada.

030-0784

**LIMITED WARRANTY ON MEDIA AND REPLACEMENT**

If you discover physical defects in the manual or in the media on which a software product is distributed, Apple will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to Apple or an authorized Apple dealer during the 90-day period after you purchased the software. In addition, Apple will replace damaged software media and manuals for as long as the software product is included in Apple's Media Exchange Program. While not an upgrade or update method, this program offers additional protection for up to two years or more from the date of your original purchase. See your authorized Apple dealer for program coverage and details. In some countries the replacement period may be different, check with your authorized Apple dealer.

**ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.**

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

**IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL,** even if advised of the possibility of such damages.

**THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED.** No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

# A/UX Programmer's Reference

## Contents

# Preface

## Conventions Used in This Manual

A/UX® manuals follow certain conventions regarding presentation of information. Words or terms that require special emphasis appear in specific fonts within the text of the manual. The following sections explain the conventions used in this manual.

### Significant fonts

Words that you see on the screen or that you must type exactly as shown appear in `Courier` font. For example, when you begin an A/UX work session, you see the following on the screen:

```
login:
```

The text shows `login:` in `Courier` typeface to indicate that it appears on the screen. If the next step in the manual is

Enter `start`

`start` appears in `Courier` to indicate that you must type in the word. Words that you must replace with a value appropriate to a particular set of circumstances appear in *italics*. Using the example just described, if the next step in the manual is

`login:` *username*

you type in your name—`Laura`, for example— so the screen shows:

```
login: Laura
```

### Key presses

Certain keys are identified with names on the keyboard. These modifier and character keys perform functions, often in combination with other keys. In the manuals, the names of these keys appear in the format of an Initial Capital letter followed by SMALL CAPITAL letters.

The list that follows provides the most common keynames.

| | | | |
|---|---|---|---|
| RETURN | DELETE | SHIFT | ESCAPE |
| OPTION | CAPS LOCK | CONTROL | |

For example, if you enter

```
Applee
```

instead of

```
Apple
```

you would position the cursor to the right of the word and press the DELETE key once to erase the additional *e*.

For cases in which you use two or more keys together to perform a specific function, the keynames are shown connected with hyphens. For example, if you see

Press CONTROL-C

you must press CONTROL and C simultaneously (CONTROL-C normally cancels the execution of the current command).

## Terminology

In A/UX manuals, a certain term can represent a specific set of actions. For example, the word *Enter* indicates that you type in an entry and press the RETURN key. If you were to see

Enter the following command: `whoami`

you would type `whoami` and press the RETURN key. The system would then respond by identifying your login name.

Here is a list of common terms and their corresponding actions.

| Term | Action |
|---|---|
| **Enter** | Type in the entry and press the RETURN key |
| **Press** | Press a *single* letter or key *without* pressing the RETURN key |
| **Type** | Type in the letter or letters *without* pressing the RETURN key |
| **Click** | Press and then immediately release the mouse button |

| Term | Action |
|---|---|
| **Select** | Position the pointer on an item and click the mouse button |
| **Drag** | Position the pointer on an icon, press and hold down the mouse button while moving the mouse. Release the mouse button when you reach the desired position. |
| **Choose** | Activate a command title in the menu bar. While holding down the mouse button, drag the pointer to a command name in the menu and then release the mouse button. An example is to drag the File menu down until the command name Open appears highlighted and then release the mouse button. |

## Syntax notation

A/UX commands follow a specific order of entry. A typical A/UX command has this form:

`command` [*flag-option*] [*argument*] . . .

The elements of a command have the following meanings.

| Element | Description |
|---|---|
| `command` | Is the command name. |
| *flag-option* | Is one or more optional arguments that modify the command. Most flag-options have the form [-opt...] where opt is a letter representing an option. Commands can take one or more options. |
| *argument* | Is a modification or specification of the command; usually a filename or symbols representing one or more filenames. |

| Element | Description |
|---|---|
| brackets ([ ]) | Surround an optional item—that is, an item that you do not need to include for the command to execute. |
| ellipses (...) | Follow an argument that may be repeated any number of times. |

For example, the command to list the contents of a directory (`ls`) is followed below by its possible flag options and the optional argument *names*.

```
ls [-R] [-a] [-d] [-C] [-x] [-m] [-l] [-L]
[-n] [-o] [-g] [-r] [-t] [-u] [-c] [-p] [-F]
[-b] [-q] [-i] [-s] [names]
```

You can enter

```
ls -a /users
```

to list all entries of the directory `/users`, where

| | |
|---|---|
| `ls` | Represents the command name |
| `-a` | Indicates that *all* entries of the directory be listed |
| `/users` | Names which directory is to be listed |

## Command reference notation

Reference material is organized by section numbers. The standard A/UX cross-reference notation is

*cmd(sect)*

where *cmd* is the name of the command, file, or other facility; *sect* is the section number where the entry resides.

- ☐ Commands followed by section numbers (1M), (7), or (8) are listed in *A/UX System Administrator's Reference*.
- ☐ Commands followed by section numbers (1), (1C), (1G), (1N), and (6) are listed in *A/UX Command Reference*.
- ☐ Commands followed by section numbers (2), (3), (4), and (5) are listed in *A/UX Programmer's Reference*.

For example,

```
cat(1)
```

refers to the command `cat`, which is described in Section 1 of *A/UX Command Reference*. References can also be called up on the screen. The `man` command or the `apropos` command displays pages from the reference manuals directly on the screen. For example, enter the command

```
man cat
```

In this example, the manual page for the `cat` command including its description, syntax, options, and other pertinent information appears on the screen. To exit, continue pressing the space bar until you see a command prompt, or press Q at any time to return immediately to your command prompt. The manuals often refer to information discussed in another guide in the suite. The format for this type of cross reference is "Chapter Title," *Name of Guide*. For a complete description of A/UX guides, see *Road Map to A/UX Documentation*. This guide contains descriptions of each A/UX guide, the part numbers, and the ordering information for all the guides in the A/UX documentation suite.

# Introduction

# to the A/UX Reference Manuals

## 1. How to use the reference manuals

*A/UX Command Reference*, *A/UX Programmer's Reference*, and *A/UX System Administrator's Reference* are reference manuals for all the programs, utilities, and standard file formats included with your A/UX® system.

The reference manuals constitute a compact encyclopedia of A/UX information. They are not intended to be tutorials or learning guides. If you are new to A/UX or are unfamiliar with a specific functional area (such as the shells or the text formatting programs), you should first read *A/UX Essentials* and the other A/UX user guides. After you have worked with A/UX, the reference manuals help you understand new features or refresh your memory about command features you already know.

## 2. Information contained in the reference manuals

A/UX reference manuals are divided into three volumes:

- The two-part *A/UX Command Reference* contains information for the general user. It describes commands you type at the A/UX prompt that list your files, compile programs, format text, change your shell, and so on. It also includes programs used in scripts and command language procedures. The commands in this manual generally reside in the directories `/bin`, `/usr/bin` and `/usr/ucb`.
- The two-part *A/UX Programmer's Reference* contains information for the programmer. It describes utilities for programming, such as system calls, file formats of subroutines, and miscellaneous programming facilities.
- *A/UX System Administrator's Reference* contains information for the system administrator. It describes commands you type at the A/UX prompt to control your machine, such as accounting

commands, backing up your system, and charting your system's activity. These commands generally reside in the directories `/etc`, `/usr/etc`, and `/usr/lib`.

These areas can overlap. For example, if you are the only person using your machine, then you are both the general user and the system administrator.

To help direct you to the correct manual, you may refer to *A/UX Reference Summary and Index*, which is a separate volume. This manual summarizes information contained in the other A/UX reference manuals. The three parts of this manual are a classification of commands by function, a listing of command synopses, and an index.

## 3. How the reference manuals are organized

All manual pages are grouped by section. The sections are grouped by general function and are numbered according to standard conventions as follows:

| | |
|---|---|
| 1 | User commands |
| 1M | System maintenance commands |
| 2 | System calls |
| 3 | Subroutines |
| 4 | File formats |
| 5 | Miscellaneous facilities |
| 6 | Games |
| 7 | Drivers and interfaces for devices |
| 8 | A/UX Startup shell commands |

Manual pages are collated alphabetically by the primary name associated with each. For the individual sections, a table of contents is provided to show the sequence of manual pages. A notable exception to the alphabetical sequence of manual pages is the first entry at the start of each section. As a representative example, `intro.1` appears at the start of Section 1. These `intro.`*section-number* manual pages are brought to the front of each section because they introduce the

other man pages in the same section, rather than describe a command or similar provision of A/UX.

Each of the reference manuals includes at least one complete section of man pages. For example, the *A/UX Command Reference* contains sections 1 and 6. However, since Section 1 (User Commands) is so large, this manual is divided into two volumes, the first containing Section 1 commands that begin with letters A through L, and the second containing Section 6 commands and Section 1 commands that begin with letters M through Z. The sections included in each volume are as follows.

*A/UX Command Reference* contains sections 1 and 6. Note that both of these sections describe commands and programs available to the general user.

- Section 1—User Commands
  The commands in Section 1 may also belong to a special category. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the N in `ypcat`(1N) indicates networking as described following.

  1C Communications commands, such as `cu` and `tip`.

  1G Graphics commands, such as `graph` and `tplot`.

  1N Networking commands, such as those which help support various networking subsystems, including the Network File System (NFS), Remote Process Control (RPC), and Internet subsystem.

- Section 6—User Commands
  This section contains all the games, such as `cribbage` and `worms`.

*A/UX Programmer's Reference* contains sections 2 through 5.

- Section 2—System Calls
  This section describes the services provided by the A/UX system kernel, including the C language interface. It includes two special categories. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the N in `connect`(2N) indicates networking as described following.

  2N Networking system calls

  2P POSIX system calls

- Section 3—Subroutines
  This section describes the available subroutines. The binary versions are in the system libraries in the `/lib` and `/usr/lib` directories. The section includes six special categories. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the N in `mount`(3N) indicates networking as described following.

  3C C and assembler library routines

  3F Fortran library routines

  3M Mathematical library routines

  3N Networking routines

  2P POSIX routines

  3S Standard I/O library routines

  3X Miscellaneous routines

- Section 4—File Formats
  This section describes the structure of some files, but does not include files that are used by only one command (such as the assembler's intermediate files). The C language `struct` declarations corresponding to these formats are in the `/usr/include` and `/usr/include/sys` directories. There is one special category in this section. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the N in

`protocols`(4N) indicates networking as described following.

4N Networking formats

- Section 5—Miscellaneous facilities
  This section contains various character sets, macro packages, and other miscellaneous formats. There are two special categories in this section. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the P in `tcp`(1P) indicates a protocol as described following. by the letter designation in parenthesis at the top of the page:

  5F Protocol families

  5P Protocol descriptions

*A/UX System Administrator's Reference* contains sections 1M, 7 and 8.

- Section 1M—System Maintenance Commands
  This section contains system maintenance programs such as `fsck` and `mkfs`.

- Section 7—Drivers and Interfaces for Devices
  This section discusses the drivers and interfaces through which devices are normally accessed. While access to one or more disk devices is fairly transparent when you are working with files, the provision of *device files* permits you more explicit modes with which to access particular disks or disk partitions, as well as other types of devices such as tape drives and modems. For example, a tape device may be accessed in automatic-rewind mode through one or more of the device file names in the `/dev/rmt` directory (see `tc`(7)). The **FILES** sections of these manual pages identify all the device files supplied with the system as well as those that are automatically generated by certain A/UX configuration utilities. The names of the man pages generally refer to device names or device driver names, rather than the names of the device files themselves.

- Section 8—A/UX Startup Shell Commands
  This section describes the commands that are available from within the A/UX Startup Shell, including detailed descriptions of

those that contribute to the boot process and those that help with the maintenance of file systems.

## 4. How a manual entry is organized

The name for a manual page entry normally appears twice, once in each upper corner of a page. Like dictionary guide words, these names appear at the top of every physical page. After each name is the section number and, if applicable, a category letter enclosed in parenthesis, such as (1) or (2N).

Some entries describe several routines or commands. For example, `chown` and `chgrp` share a page with the name `chown`(1) at the upper corners. If you turn to the page `chgrp`(1), you find a reference to `chown`(1). (These cross-reference pages are only included in *A/UX Command Reference* and *A/UX System Administrator's Reference*.)

All of the entries have a common format, and may include any of the following parts:

**NAME**
is the name or names and a brief description.

**SYNOPSIS**
describes the syntax for using the command or routine.

**DESCRIPTION**
discusses what the program does.

**FLAG OPTIONS**
discusses the flag options.

**EXAMPLES**
gives an example or examples of usage.

**RETURN VALUE**
describes the value returned by a function.

**ERRORS**
describes the possible error conditions.

**FILES**
lists the filenames that are used by the program.

**SEE ALSO**
provides pointers to related information.

**DIAGNOSTICS**
discusses the diagnostic messages that may be produced. Self-explanatory messages are not listed.

**WARNINGS**
points out potential pitfalls.

**BUGS**
gives known bugs and sometimes deficiencies. Occasionally, it describes the suggested fix.

# 5. Locating information in the reference manuals

The directory for the reference manuals, *A/UX Reference Summary and Index*, can help you locate information through its index and summaries. The tables of contents within each of the reference manuals can be used also.

## 5.1 Table of contents

Each reference manual contains an overall table of contents and individual section contents. The general table of contents lists the overall contents of each volume. The more detailed section contents lists the manual pages contained in each section and a brief description of their function. For the most part, entries appear in alphabetic order within each section.

## 5.2 Commands by function

This summary classifies the A/UX user and administration commands by the general, or most important function they perform. The complete descriptions of these commands are found in *A/UX Command Reference* and *A/UX System Administrator's Reference*. Each is mentioned just once in this listing.

The summary gives you a broader view of the commands that are available and the context in which they are most often used.

## 5.3 Command synopses

This section is a compact collection of syntax descriptions for all the commands in *A/UX Command Reference* and *A/UX System Administrator's Reference*. It may help you find the syntax of commands more quickly when the syntax is all you need.

## 5.4 Index

The index lists key terms associated with A/UX subroutines and commands. These key terms allow you to locate an entry when you don't know the command or subroutine name.

The key terms were constructed by examining the meaning and usage of the A/UX manual pages. It is designed to be more discriminating and easier to use than the traditional permuted index, which lists nearly all words found in the manual page NAME sections.

Most manual pages are indexed under more than one entry; for example, `lorder`(1) is included under "archive files," "sorting," and "cross-references." This way you are more likely to find the reference you are looking for on the first try.

## 5.5 Online documentation

Besides the paper documentation in the reference manuals, A/UX provides several ways to search and read the contents of each reference from your A/UX system.

To see a manual page displayed on your screen, enter the `man`(1) command followed by the name of the entry you want to see. For example,

```
man passwd
```

To see the description phrase from the NAME section of any manual page, enter the `whatis` command followed by the name of the entry you want to see. For example,

```
whatis apropos
```

To see a list of all manual pages whose descriptions contain a given keyword or string, enter the `apropos` command followed by the word or string. For example,

```
apropos remove
```

These online documentation commands are described more fully in the manual pages `man`(1), `whatis`(1), and `apropos`(1) in *A/UX Command Reference*.

# Table of Contents

## Section 2: System Calls

## NAME

intro — introduction to system calls and error numbers

## SYNOPSIS

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

## DESCRIPTION

This section describes all of the A/UX® system calls. The system calls identified with the letter "P" following the section number are part of the A/UX POSIX environment. The A/UX POSIX programming environment is described in the *A/UX Guide to POSIX* and *A/UX Programming Languages and Tools, Volume 1*. Most of these calls have one or more error returns. An error condition is indicated by a returned value that is otherwise impossible, which is almost always −1. The individual descriptions specify the details. An error number is also made available in the external variable errno, which is not cleared on successful calls. So errno, should be tested only after an error has been indicated.

There is a table of messages associated with each error and a routine for printing the message (see perror(3C)). Each system-call description attempts to list all possible error numbers.

## ERRORS

The following is a complete list of A/UX error numbers and their names as defined in <errno.h>. Also given is a description of the most likely cause of the error.

1 EPERM Not owner
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or the superuser. It is also returned when ordinary users attempt modifications reserved for the superuser.

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

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

4 EINTR Interrupted system call
An asynchronous signal, such as an interrupt or quit, which the user program elected to catch, occurred during a system call. If execution is resumed after processing the signal, it

will appear as if the interrupted system call returned this error condition.

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

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

7 E2BIG Argument list too long
An argument list longer than ARG_MAX is presented to a member of the exec family.

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

9 EBADF Bad file number
Either a file descriptor does not refer to an open file, or a read/write request is made to a file that is open only for writing/reading.

10 ECHILD No children
A wait was executed by a process that did not have existing child processes waiting for it.

11 EAGAIN No more processes
The system is out of a resource that may be available later. A fork failed because the system's process table is full or the user is not allowed to create any more processes. A system call that requires memory may also fail with this error if the system is out of memory or swap space, but the request is less than the system-imposed per process limit (see ulimit(2)).

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

gisters or if there is not enough swap space during a `fork`.

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

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

15 ENOTBLK Block device required
A nonblock file was mentioned where a block device was required, for example, in `mount`.

16 EBUSY Mount device busy
The device or resource is currently unavailable. An attempt was made to mount a device that was already mounted or to dismount a device on which there is an active file (open file, current directory, mounted-on file, or active text segment). This error also occurs if an attempt is made to enable accounting that is already enabled.

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, for example, `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, to read a write-only device.

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

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

22 EINVAL Invalid argument
An invalid argument was implemented—for example, dismounting a nonmounted device, mentioning an undefined signal in `signal` or `kill`, reading or writing a file for which `lseek` has generated a negative pointer. This error is also generated by the math functions described in the (3M) entries of this manual.

23 ENFILE File table overflow
The system file table is full and temporarily cannot accept another open.

24 EMFILE Too many open files
A process may not have more than the maximum number of file descriptors (OPEN_MAX) open at a time. When a record lock is being created with fcntl, too many files have record locks on them.

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

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

*Note:* If you are running a network file system (NFS) and you are accessing a shared binary remotely, it is possible that you will not get this errno.

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

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

29 ESPIPE Illegal seek
An lseek was issued to a pipe. This error should also be issued for other nonseekable devices.

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

31 EMLINK Too many links
An attempt was made to create more than the maximum number of links (LINK_MAX) to a file.

32 EPIPE Broken pipe
A write was attempted to a pipe on which there is no process

to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

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

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

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

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

37 ECHRNG Channel number out of range
This errno is included for compatibility with AT&T.

38 EL2NSYNC Level 2 not synchronized
This errno is included for compatibility with AT&T.

39 EL3HLT Level 3 halted
This errno is included for compatibility with AT&T.

40 EL3RST Level 3 reset
This errno is included for compatibility with AT&T.

41 ELNRNG Link number out of range
This errno is included for compatibility with AT&T.

42 EUNATCH Protocol driver not attached
This errno is included for compatibility with AT&T.

43 ENOCSI No CSI structure available
This errno is included for compatibility with AT&T.

44 EL2HLT Level 2 halted
This errno is included for compatibility with AT&T.

45 EDEADLK Deadlock
A deadlock situation was detected and avoided.

55 EWOULDBLOCK Operation would block
An operation that would cause a process to block was attempted on an object in nonblocking mode (see socket(2N)

and setcompat(2)).

56 EINPROGRESS Operation now in progress
An operation that takes a long time to complete, such as connect(2N), was started on a nonblocking object (see socket(2N)).

57 EALREADY Operation already in progress
An operation was attempted on a nonblocking object that already had an operation in progress.

58 ENOTSOCK Socket operation on nonsocket
A socket operation was attempted on an object that is not a socket.

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

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

61 EPROTOTYPE Protocol wrong type for socket
A protocol was specified that does not support the semantics of the socket type requested. For example, you cannot use the internet UDP protocol with type SOCK_STREAM.

62 ENOPROTOOPT Bad protocol option
A bad option was specified in getsockopt(2) or setsockopt(2).

63 EPROTONOSUPPORT Protocol not supported
The protocol has not been configured into the system, or there is no implementation for it.

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

65 EOPNOTSUPP Operation not supported on socket
The support for the operation on the selected socket type has not been configured, or there is no implementation for it—for example, trying to establish a connection on a datagram socket.

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

67 EAFNOSUPPORT Address not supported by protocol family
An address incompatible with the requested protocol was used. For example, PUP Internet addresses cannot necessarily be used with ARPANET protocols.

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

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

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

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

72 ENETRESET Network dropped connection on reset
The connected host crashed and rebooted.

73 ECONNABORTED Software caused connection abort
A connection abort was caused that was internal to the host machine.

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

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

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

77 ENOTCONN Socket is not connected
A request to send or receive data was disallowed because the socket had already been shut down with a previous shutdown(2).

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

80 ETIMEDOUT Connection timed out
A connect request failed because the connected party did

not properly respond after a period of time. (The timeout period is dependent on the communications protocol.)

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

82 ELOOP Too many levels of symbolic links
A pathname lookup involved more than eight symbolic links.

83 ENAMETOOLONG Filename too long
A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.

84 EHOSTDOWN Host is down
A socket operation encountered a defunct host.

85 EHOSTUNREACH No route to host
A socket operation was attempted to an unreachable host.

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

87 ENOSTR Device not a stream
A stream operation was attempted on a file descriptor that is not a streams device.

88 ENODATA No data (for no delay I/O)
Reading from a stream and the O_NEDELAY flag is set (from open(2) or fcntl(2)), but no data is ready to be read.

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

90 ENOSR Out of stream resources
During a streams open(2), either no streams queues or no streams-head data structures were available.

95 ESTALE Stale NFS file handle
A client referenced an open file after the file was deleted.

96 EREMOTE Too many levels of remote in path
An attempt was made to remotely mount a file system into a

path that already has a remotely mounted component.

97 EPROCLIM Too many processes

98 EUSERS Too many users

100 EDEADLOCK Locking deadlock error
This error is returned by locking(2) if deadlock would occur or when the lock table overflows.

101 ENOLCK No locks available
If either FLCKREC or FLCKFIL is reached, the lock is not allowed.

102 ENOSYS Funcnot implemented

## DEFINITIONS

### System Constants

The following are the default implementation-specific constants defined in <limits.h> for the A/UX system that is used on the Macintosh II®:

| | |
|---|---|
| ARG_MAX | Maximum length of argument to exec (5120). |
| CHAR_BIT | Number of bits in a datum of type char (8). |
| CHAR_MAX | Maximum integer value of a char (255). |
| CHILD_MAX | Maximum number of processes per user ID (25). |
| INT_MAX | Maximum decimal value of an int (2,147,483,647). |
| LINK_MAX | Maximum number of links to a single file (1000) |
| LONG_MAX | Maximum decimal value of a long (2,147,483,647). |
| MAXDOUBLE | Maximum decimal value of a double (1.79769313486231470e+308). |
| NAME_MAX | Maximum number of characters in a filename (255). On System V file systems, names are limited to 14 characters. |
| OPEN_MAX | Maximum number of files a process can have open (32). |

| | |
|---|---|
| PATH_MAX | Maximum number of characters in a pathname (1024). |
| PID_MAX | Maximum value for a process ID (30,001). |
| PIPE_MAX | Maximum number of bytes written to a pipe in a write (5120). |
| PROC_MAX | Maximum number of simultaneous system wide processes (50). |
| SHRT_MAX | Maximum decimal value of a short (65,535). |
| SYS_NMLN | Number of characters in a string returned by uname (9). |
| UID_MAX | Maximum value for a user ID or group ID (60,001). |
| USI_MAX | Maximum decimal value of an unsigned (4,294,967,295). |
| INT_MIN | Minimum decimal value for an int (–2,147,483,648). |
| LONG_MIN | Minimum decimal value for a long (–2,147,483,648). |
| SHRT_MIN | Minimum decimal value for a short (–32,768). |
| ULIMIT | Maximum number of 512-byte blocks in a file (16,777,216). |

**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 PID_MAX.

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

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

**Tty Group ID**
Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related processes upon termination of one of the processes in the group (see `exit`(2) and `signal`(3)).

**Session**
Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session membership (see `setsid`(3P)). Multiple process groups may be in the same session (see `setpgid`(3P)).

**Session Leader**
A process that has created a session (see `setsid`(3P)).

**Controlling Terminal**
A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session (see `termio`(7P)).

**Controlling Process**
The session leader that established the connection to the controlling terminal.

**Foreground Process Group**
Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. The foreground process group has priveleges, when accessing its controlling terminal, that are denied to background process groups (see `termios`(7P)).

**Background Process Group**
Any process group that is a member of a session that has established a connection with a controlling terminal that is not in the foreground process group.

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

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

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

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

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

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

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

**File Descriptor**
A file descriptor is a small integer used to do I/O on a file. The value of a file descriptor is from 0 to `OPEN_MAX`–1. A process may not have more than `OPEN_MAX` file descriptors open simultaneously. A file descriptor is returned by system calls such as `open`(2) or `pipe`(2). The file descriptor is used as an argument by calls such as `read`(2), `write`(2), `ioctl`(2), and `close`(2).

**File Pointer**
A file with associated `stdio` buffering is called a stream. A stream is a pointer to a type `FILE` defined by the `<stdio.h>` header file: The `fopen`(3S) routine creates descriptive data for a stream and returns a pointer that identifies the stream in all further transactions with other `stdio` routines.

Most `stdio` routines manipulate either a stream created by the `fopen`(3S) function or one of the three streams that are associated with three files that are expected to be open in the base system (see `termio`(7). These three streams are declared in the `<stdio.h>` header file.

| | |
|---|---|
| `stdin` | the standard input file |
| `stdout` | the standard output file |
| `stderr` | the standard error file |

Output streams, with the exception of the standard error stream `stderr`, are, by default, buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream `stderr` is, by default, unbuffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When an output stream 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 newline character is written or terminal input is requested. The `setbuf`(3S) routines may be used to change the buffering strategy of the stream.

**Filename**

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

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

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

**Pathname and Path Prefix**

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

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

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

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

where *<file>* 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 pathname begins with a slash, the path search begins at the root directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

**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.

**Root Directory and Current Working Directory**

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

**File-Access Permissions**

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

The effective user ID of the process is the superuser.

The effective user ID of the process matches the user ID of the owner of the file, 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, 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, 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.

Otherwise, the corresponding permissions are denied.

## INTERPROCESS COMMUNICATION

**Message Queue Identifier**

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

```
struct ipc_perm msg_perm; /* operation permission
```

```
                                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
                                operation */
ushort msg_lrpid;             /* pid of last msgrcv
                                operation */
time_t msg_stime;             /* last msgsnd time */
time_t msg_rtime;             /* last msgrcv time */
time_t msg_ctime;             /* last change time */
                              /* times measured in secs
                                since 00:00:00 GMT, 1/1/70 */
```

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

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

msg_qnum is the number of messages currently on the queue. msg_qbytes is the maximum number of bytes allowed on the queue. msg_lspid is the process ID of the last process that performed a msgsnd operation. msg_lrpid is the process ID of the last process that performed a msgrcv operation. msg_stime 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(2) operation that changed a member of the msg_perm structure.

**Semaphore Identifier**

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

```
struct ipc_perm sem_perm; /* operation permission
                             struct */
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 secs since
                             00:00:00 GMT, 1/1/1970 */
```

sem_perm is an ipc_perm structure that specifies the semaphore operation permission (described later in this section). This

structure includes the following members:

```
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(2) operation, and sem_ctime is the time of the last semctl(2) operation that changed a member of the structure described earlier.

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

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

semval is a non-negative integer. 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 and 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 0.

**Shared Memory Identifier**

A shared memory identifier (*shmid*) is a unique positive integer created by a shmget(2) system call. Each *shmid* has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure referred to as shmid_ds contains the following members:

```
struct  ipc_perm shm_perm; /* operation permission 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 secs
                              since 00:00:00 GMT, 1/1/70 */
```

shm_perm is an ipc_perm structure that specifies the shared-

memory-operation permission (described later in this section). This structure includes the following members:

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

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(2) 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(2) operation that changed one of the members of the structure outlined earlier.

IPC PERMISSIONS

In the msgop(2) and msgctl(2) system-call descriptions, the permission required for an operation is interpreted as follows:

00400 Read by user.
00200 Write by user.
00060 Read/write by group.
00006 Read/write by others.

**Message Operation Permissions**

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

The effective user ID of the process is the superuser.

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, 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.

The effective user ID of the process does not match msg_perm.[c]uid, 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.

**Semaphore Operation Permissions**

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 the superuser.

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.

The effective user ID of the process does not match `sem_perm.[c]uid`, 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`, 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-Operation Permissions**

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

The effective user ID of the process is superuser.

The effective user ID of the process matches `shm_perm.[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`, 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.

The effective user ID of the process does not match `shm_perm.[c]uid`, 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.

**SEE ALSO**

close(2), ioctl(2), open(2), pipe(2), read(2), write(2), intro(3), perror(3).

"Overview of the A/UX Programming Environment" in *A/UX Programming Languages and Tools, Volume 1*.

## NAME

accept — accept a connection on a socket

## SYNOPSIS

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

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

## DESCRIPTION

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

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

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

## RETURN VALUE

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

**ERRORS**

accept will fail if:

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

**SEE ALSO**

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

## NAME

access — determine accessibility of a file

## SYNOPSIS

```
#include<unistd.h>
int access(path, amode)
char *path;
int amode;
```

## DESCRIPTION

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

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

For the POSIX environment, the following values are defined for passing *amode* as the value of <unistd.h>:

| | | |
|---|---|---|
| R_OK | 04 | read |
| W_OK | 02 | write |
| X_OK | 01 | executable file or searchable directory |
| F_OK | 00 | check existence of file |

## RETURN VALUE

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

## ERRORS

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

| | |
|---|---|
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOTDIR] | A component of the path prefix is not a directory. |

| | |
|---|---|
| [ENOENT] | Read, write, or execute (search) permission is requested for a null pathname. |
| [ENOENT] | The named file does not exist. |
| [EACCES] | Search permission is denied on a component of the path prefix. |
| [EROFS] | Write access is requested for a file on a read-only file system. |
| [ETXTBSY] | Write access is requested for a pure procedure (shared text) file that is being executed.<br>*Note:* If you are running network file system (NFS) and you are accessing a shared binary remotely, it is possible that you will not get this `errno`. |
| [EACCESS] | Permission bits of the file mode do not permit the requested access. |
| [EFAULT] | The *path* points outside the allocated address space for the process. |
| [EINVAL] | Value of *amode* is invalid. |

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

The superuser is always granted execute permission even though execute permission is meaningful only for directories and regular files and even though `exec` requires that at least one execute mode bit is set for the regular file to be executable.

Notice that only access bits are checked. A directory may be announced as writable by `access`, but an attempt to open it for writing will fail because writing into the directory structure itself is not allowed, even though files may be created there. A file may look executable, but `exec` will fail unless it is in the proper format.

**SEE ALSO**

chmod(2), stat(2).

## NAME

acct — enable or disable process accounting

## SYNOPSIS

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

## DESCRIPTION

acct is used to enable or disable the system process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an exit call or a signal; see exit(2) and signal(3). The effective user ID of the calling process must be superuser to use this call.

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

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

## RETURN VALUE

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

## ERRORS

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

| | |
|---|---|
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [EPERM] | The effective user ID of the calling process is not superuser. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [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. |

| | |
|---|---|
| [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. |
| [EROFS] | The named file resides on a read-only file system. |
| [EFAULT] | *path* points to an illegal address. |

**SEE ALSO**

acct(1M), exit(2), signal(3), acct(4).

## NAME

adjtime — correct the system time

## SYNOPSIS

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

adjtime(delta, olddelta)
struct timeval *delta;
struct timeval *olddelta;
```

## DESCRIPTION

adjtime makes small adjustments to the system time, as returned by gettimeofday(2), advancing or retarding it by the time specified by the timeval *delta*. If *delta* is negative, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If *delta* is positive, a larger increment than normal is used. The skew used to perform the correction is generally a fraction of one percent. Thus, the time is always a monotonically increasing function. A time correction from an earlier call to adjtime may not be finished when adjtime is called again. If *olddelta* is nonzero, then the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

This call may be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

The call adjtime(2) is restricted to the superuser.

## RETURN VALUE

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

## ERRORS

adjtime will fail if:

| | |
|---|---|
| [EFAULT] | An argument points outside the process's allocated address space. |
| [EPERM] | The process's effective user ID is not that of the superuser. |

**SEE ALSO**

date(1).

## NAME

alarm — set a process's alarm clock

## SYNOPSIS

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

## DESCRIPTION

alarm instructs the calling process's alarm clock to send the signal SIGALRM to the calling process after the number of real time seconds specified by *sec* have elapsed; see signal(3).

alarm requests are not stacked; successive calls reset the calling process's alarm clock. If the argument is 0, any alarm request is canceled. Because the clock has a 1-second resolution, the signal may occur up to one second early; because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 4,294,967,295 (2**32-1) seconds, or 136 years.

## RETURN VALUE

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

## SEE ALSO

pause(2), setitimer(2), signal(3).

## NAME

bind — bind a name to a socket

## SYNOPSIS

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

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

## DESCRIPTION

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

## NOTES

The rules used in name binding vary between communication domains. Consult the manual entries in Section 5 (specifically inet(5F)) for detailed information.

## RETURN VALUE

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

## ERRORS

bind will fail if

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

**SEE ALSO**

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

## NAME

brk, sbrk — change data segment space allocation

## SYNOPSIS

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

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

## DESCRIPTION

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

## RETURN VALUE

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

## ERRORS

brk and sbrk will fail without making any change in the allocated space if the following is true:

| | |
|---|---|
| [ENOMEM] | Not enough space. Program asks for more space than the system is able to supply. |
| [EAGAIN] | The system has temporarily exhausted its available memory or swap space. |

Such a change would result in more space being allocated than is allowed by a system-imposed maximum (see ulimit(2)). 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(2)).

**SEE ALSO**

exec(2), shmop(2), ulimit(2).

## NAME

chdir — change working directory

## SYNOPSIS

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

## DESCRIPTION

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

## RETURN VALUE

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

## ERRORS

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

| | |
|---|---|
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [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. |

## SEE ALSO

csh(1), ksh(1), sh(1), chroot(2).

## NAME

chmod — change mode of file

## SYNOPSIS

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

## DESCRIPTION

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

Access permission bits are interpreted as follows:

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

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

If the effective user ID of the process is not the superuser, mode bit 01000 (save text image after execution) is cleared.

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

If an executable file is prepared for sharing (see the cc -n option), 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 need not be read from the file system but can simply be swapped in, saving time.

Changing the owner of a file turns off the mode bit 04000 (set user ID), unless the superuser does it. This makes the system somewhat more secure at the expense of a degree of compatibility.

RETURN VALUE

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

ERRORS

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

| | |
|---|---|
| `[ENOTDIR]` | A component of the path prefix is not a directory. |
| `[ENAMETOOLONG]` | A component of a pathname exceeded `NAME_MAX` characters, or an entire pathname exceeded `PATH_MAX`. |
| `[ELOOP]` | Too many symbolic links were encountered in translating a pathname. |
| `[ENOENT]` | The named file does not exist. |
| `[EACCES]` | Search permission is denied on a component of the path prefix. |
| `[EPERM]` | The effective user ID does not match the owner of the file, and the effective user ID is not the superuser. |
| `[EROFS]` | The named file resides on a read-only file system. |
| `[EFAULT]` | The *path* points outside the allocated address space of the process. |

SEE ALSO

`chmod`(1), `chown`(2), `open`(2), `stat`(2), `mknod`(2), `umask`(2).

## NAME

chown, fchown — change owner and group of a file

## SYNOPSIS

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

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

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

## DESCRIPTION

The file that is named by *path* or referenced by *fd* has its *owner* and *group* changed as specified. Only the superuser or the owner of the file may execute this call.

chown clears the set-user-ID and set-group-ID bits on the file to prevent accidental creation of set-user-ID and set-group-ID programs owned by the superuser.

If chown is invoked successfully by users other than the superuser, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared. This prevents ordinary users from effectively making themselves other users or members of a group to which they don't belong.

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

If the compatibility flag COMPAT_BSDCHOWN is set, chown is restricted to processes with superuser privileges. This compatibility flag also limits a process that has an effective user ID equal to the user ID of the file, but otherwise without appropriate privileges, from changing the file's group ID to the effective group ID of the process only, or to its supplementary group IDs.

For the POSIX environment, the routine posix_crt0.o does set the COMPAT_BSDCHOWN flag to support these added restrictions.

## RETURN VALUE

A value of 0 is returned if the operation was successful; A value of −1 is returned if an error occurs, and a more specific error code is placed in the global variable errno.

**ERRORS**

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

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

`fchown` will fail if one or both of the following are true:

| | |
|---|---|
| `[EBADF]` | The *fd* does not refer to a valid descriptor. |
| `[EINVAL]` | The *fd* refers to a socket, not a file. |

**SEE ALSO**

`chown`(1), `chgrp`(2), `chmod`(2).

## NAME

chroot — change root directory

## SYNOPSIS

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

## DESCRIPTION

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

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

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

## RETURN VALUE

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

## ERRORS

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

| | |
|---|---|
| [ENOTDIR] | Any component of the path name is not a directory. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOENT] | The named directory does not exist. |
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [EPERM] | The effective user ID is not the superuser. |
| [EFAULT] | *path* points outside the allocated address space of the process. |

**SEE ALSO**

chroot(1M), chdir(2).

## NAME

close — close a file descriptor

## SYNOPSIS

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

## DESCRIPTION

close closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

The argument *fildes* is a file descriptor obtained from a creat, open, dup, fcntl, pipe, or socket system call. A close system call is automatically performed on all open files are part of exit . There is a small, finite limit on the number of open files per process (OPEN_MAX), so close is necessary for programs that deal with many files.

When all file descriptors associated with a pipe or FIFO special file have been closed, any data remaining in the pipe or FIFO is discarded. When all file descriptors associated with an open file description have been closed, the file description is freed. If the link count of the file is 0 when all the file descriptors associated with the file have been closed, the space occupied by the file is freed and the file is no longer accessible.

## RETURN VALUE

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

## ERRORS

close will fail if one or both of the following are true:

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

[EINTR] close was interrupted by a signal.

## SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2), socket(2N).

## NAME

connect — initiate a connection on a socket

## SYNOPSIS

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

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

## DESCRIPTION

connect is used to initiate a connection on a socket. The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call permanently specifies the peer to which datagrams are to be sent; if it is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name* which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way.

## RETURN VALUE

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

## ERRORS

connect fails if:

| | |
|---|---|
| [EBADF] | *s* is not a valid descriptor. |
| [ENOTSOCK] | *s* is a descriptor for a file, not a socket. |
| [EADDRNOTAVAIL] | The specified address is not available on this machine. |
| [EAFNOSUPPORT] | Addresses in the specified address family cannot be used with this socket. |
| [EISCONN] | The socket is already connected. |
| [ETIMEDOUT] | Connection establishment timed out without establishing a connection. |
| [ECONNREFUSED] | The attempt to connect was forcefully rejected. |
| [ENETUNREACH] | The network isn't reachable from this host. |

| | |
|---|---|
| [EADDRINUSE] | The address is already in use. |
| [EFAULT] | The *name* parameter specifies an area outside the process address space. |
| [EWOULDBLOCK] | The socket is nonblocking and the and the connection cannot be completed immediately. It is possible to select(2N) the socket while it is connecting by selecting it for writing. |

**SEE ALSO**

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

## NAME

creat — create a new file or rewrite an existing one

## SYNOPSIS

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

## DESCRIPTION

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

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

All bits set in the file-mode-creation mask of the process are cleared. See umask(2).

Mode bit 01000 (save text image after execution bit) is cleared. See chmod(2).

For the POSIX environment the following constants for *mode* are defined in <sys/stat.h> :

| | |
|---|---|
| S_IRUSR | read permission, owner |
| S_IWUSR | write permission, owner |
| S_IXUSR | execute/search permission, owner |
| S_IRGRP | read permission, group |
| S_IWGRP | write permission, group |
| S_IXGRP | execute/search permission, group |
| S_IRUSR | read permission, others |
| S_IWUSR | write permission, others |
| S_IXUSR | execute/search permission, others |

On successful completion, the file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descrip-

tor is set to remain open across exec system calls, see fcntl(2). No process may have more than the maximum number of files, OPEN_MAX, open simultaneously.

The *mode* given is arbitrary; it need not allow writing. This feature is used by programs that deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a creat, an error is returned, and the program knows that the name is unusable for the moment.

**RETURN VALUE**

On 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.

**ERRORS**

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

| | |
|---|---|
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOENT] | A component of the path prefix does not exist. |
| [EACCES] | Search permission is denied on a component of the path prefix. |
| [ENOENT] | The pathname 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. |

*Note:* If you are running a network file system (NFS) and you are accessing a shared binary remotely, it

is possible that you will not get this `errno`.

| | |
|---|---|
| `[EACCES]` | The file exists, and write permission is denied. |
| `[EISDIR]` | The named file is an existing directory. |
| `[EMFILE]` | The maximum number of file descriptors are currently open. |
| `[EFAULT]` | The *path* points outside the allocated address space of the process. |
| `[ENFILE]` | The system file table is full. |

**BUGS**

The system-scheduling algorithm does not make this a true uninterruptible operation, and a race condition may develop if `creat` is done at precisely the same time by two different processes.

**SEE ALSO**

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

## NAME

dup — duplicate a descriptor

## SYNOPSIS

```
int dup (oldd)
int oldd;
```

## DESCRIPTION

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

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

## RETURN VALUE

The value −1 is returned if an error occurs in either call and errno is set to indicate the error.

## ERRORS

dup fails if:

| | |
|---|---|
| [EBADF] | The old descriptor is not a valid active descriptor |
| [EMFILE] | Too many descriptors are active. |

## SEE ALSO

accept(2N), open(2), close(2), getdtablesize(2N), pipe(2), socket(2N), dup2(3N).

## NAME

execl, execv, execle, execve, execlp, execvp — execute a file

## SYNOPSIS

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

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

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

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

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

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

## DESCRIPTION

exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the "new process file." There can be no return from a successful exec because the calling process is overlaid by the new process.

*path* points to a pathname 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 variable PATH (see environ(5)).

The shell is invoked if a command file is found by execlp or execvp.

*arg0*, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list that is 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 that is 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 and is directly usable in another `execv` because *argv*[*argc*] is 0.

*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`, `execv`, `execlp`, and `execvp`, the C run-time startoff routine places a pointer to the environment of the calling process in the global cell

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

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

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

By default, a new process automatically has the system default compatibility flag (see `setcompat`(2)). However, if the COMPAT_EXEC flag is set in the calling process, the new process inherits the compatibility flag of the calling process. In the A/UX® POSIX environment, the compatibility flag always includes COMPAT_EXEC (see `setposix`(3P)).

Signals set to the default action in the calling process are set to the default action in 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 the default action in the new process.

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

The shared-memory segments attached to the calling process are not attached to the new process (see `shmop`(2)).

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

Regions of physical memory mapped into the virtual address space of the calling process are detached from the address space of the new process (see phys(2)).

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

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

execl is useful when a known file with known arguments is being called. The arguments to execl are the character strings constituting the file and the associated arguments. The first argument is conventionally the same as the filename (or its last component). A 0 argument must end the argument list.

When a C program is executed, it is called by

```
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.

*envp* is a pointer to an array of strings that constitute the environment of the process. Each string consists of a name, an =, and a null-terminated value. The array of pointers is terminated by a null pointer. The shell sh(1) passes an environment entry for each global shell variable defined when the program is called. See environ(5) for some conventionally used names. The C runtime startoff routine places a copy of *envp* in the global cell

environ, which is used by execv and execl to pass the environment to any subprograms executed by the current program. The exec routines use lower-level routines, as follows, to pass an environment explicitly:

```
execve(file, argv, environ);
execle(file, arg0, arg1, ... , argn, 0, environ);
```

execlp and execvp are called with the same arguments as execl and execv; however, they duplicate the actions of the shell in searching for an executable file in a list of directories. The directory list is obtained from the environment.

**RETURN VALUE**

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

**ERRORS**

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

| | |
|---|---|
| [ENOENT] | One or more components of the pathname of the new process file do not exist. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOTDIR] | A component of the path prefix of the new process file prefix is not a directory. |
| [EACCES] | Search permission is denied for a directory listed in the path prefix of the new process file. |
| [EACCES] | The new process file is not an ordinary file. |
| [EACCES] | The new-process-file mode denies execution permission. |
| [EAGAIN] | The system has temporarily exhausted its available memory or swap space. |
| [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.<br><br>*Note:* If you are running a network file system (NFS) and you are accessing a shared binary remotely, it is possible that you will not get this errno. |
| [ENOMEM] | The new process requires more memory than is allowed by the system-imposed maximum (MAXMEM). |
| [E2BIG] | The number of bytes in the argument list of the new process is greater than the system-imposed limit of ARG_MAX. |
| [EFAULT] | The new process file is not as long as indicated by the size values in its header. |
| [EFAULT] | The pointers *path*, *argv*, or *envp* indicate an illegal address. |

**SEE ALSO**

csh(1), ksh(1), sh(1), alarm(2), exit(2), fork(2), nice(2), phys(2), ptrace(2), semop(2), setcompat(2), times(2), signal(3).

## NAME

exit, _exit — terminate process

## SYNOPSIS

```
void exit(status)
int status;

void _exit(status)
int status;
```

## DESCRIPTION

exit terminates the calling process with the following consequences:

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

If the parent process of the calling process is executing a wait(2), it is notified of the termination of the calling process and the low-order 8 bits (bits 0377) of *status* are made available to it (see wait(2)). A SIGCHLD signal is sent to the parent process of the calling process.

If the parent process of the calling process is not executing a wait, the calling process is transformed into a zombie process, and the exit status is saved for return to the parent should the parent subsequently execute a wait(2). A "zombie process" is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process-table slot that it occupies is partially overlaid with time-accounting information (see <sys/proc.h>) to be used by times.

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

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

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

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

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

If the process 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.

All open file descriptors and directory streams of the calling process are closed. If the process is a controlling process, a SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the calling process. The controlling terminal associated with the session is disassociated from the session.

If the termination of the calling process causes a process group to be orphaned and if any member of this process group is stopped, a SIGHUP, followed by a SIGCONT signal is sent to each process in the process group.

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

SEE ALSO

acct(2), fork(2), intro(2), plock(2), semop(2), wait(2), signal(3).

WARNINGS

See the warning section in signal(3).

## NAME

fcntl — file control

## SYNOPSIS

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

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

## DESCRIPTION

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

The cmd values are:

F_DUPFD — Return a new file descriptor as follows:

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

Same open file (or pipe) as the original file.

Same file pointer as the original file (that is, both file descriptors share one file pointer).

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

Same file status flags (that is, both file descriptors share the same file status flags).

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

F_GETFD — Get the file descriptor flags that are associated with the file descriptor *fildes*.

F_SETFD — Set the file descriptor flags associated with *fildes* to *arg*, which is interpreted as type int.

F_GETFL — Get file status flags and file-access modes of *fildes* file.

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

F_GETLK — Get the first lock that blocks the lock description given by the variable of type struct flock pointed to by *arg*. The information retrieved overwrites the information passed to fcntl in

the `flock` structure. If no lock is found that would prevent this lock from being created, 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`(5)). The *cmd* `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` Does 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.

`F_GETOWN` Get the process ID or process group currently receiving `SIGIO` and `SIGURG` signals. Process groups are returned as negative values.

`F_SETOWN` Set the process or process group to receive `SIGIO` and `SIGURG` signals. Process groups are specified by supplying *arg* as a negative value; otherwise, *arg* is interpreted as a process ID.

File descriptor flags, file status flags, and file-access modes are associated with one file descriptor and do not affect other file descriptors that refer to the same file.

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 only used with the *cmd* `F_GETLK` to return the value for a block that is locked. 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 extend always to the end of a file by setting `l_len` to 0. If such a lock also has `l_start` set to 0, the whole file is locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take 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`(2) system call.

**RETURN VALUE**

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

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

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

**ERRORS**

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

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

| | |
|---|---|
| [EMFILE] | *cmd* is F_DUPFD, and the maximum number of file descriptors are currently open. |
| [ENFILE] | *cmd* is F_DUPFD, and *arg* is negative or greater than the maximum number of file descriptors currently open. |
| [EINVAL] | *cmd* is F_GETLK, F_SETLK, or SETLKW, and *arg*, or the data it points to is not valid. |
| [EACCESS] | *cmd* is F_SETLK, and the type of lock (l_type) is a read (F_RDLCK) or write (F_WRLCK) lock. Also 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-locked or write-locked by another process. |
| [ENOLCK] | *cmd* is F_SETLK or F_SETLKW, and the type of lock is a read or write lock. Also, no more file-locking headers are available (too many files have segments locked), or no more record locks are available (too many files have segments locked). |
| [EDEADLK] | *cmd* is F_SETLK. When the lock is blocked by some lock from another process and sleeping (waiting) until that lock becomes free, this causes a deadlock situation. |
| [ENOTSOCK] | *cmd* is F_GETOWN or F_SETOWN, and *fildes* is not a file descriptor for a socket. |
| [EINVAL] | The value of l-whence in the flock structure is invalid. |

**SEE ALSO**

close(2), creat(2), dup(2), exec(2), ioctl(2), open(2), pipe(2), socket(2N), lockf(3C), fcntl(5).

## NAME

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

## SYNOPSIS

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

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

## DESCRIPTION

flock applies or removes an *advisory* lock on the file associated with the file descriptor *fd*. A lock is applied by specifying an *operation* parameter that is the inclusive OR of LOCK_SH or LOCK_EX and, possibly, LOCK_NB. To unlock an existing lock, the *operation* should be LOCK_UN. The values for these operations are defined as follows:

```
#define LOCK_SH 1 /* shared lock */
#define LOCK_EX 2 /* exclusive lock */
#define LOCK_NB 4 /* nonblocking lock */
#define LOCK_UN 8 /* unlock */
```

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

The locking mechanism allows two types of locks: *shared* locks and *exclusive* locks. More than one process may hold a shared lock for a file at any given time, but multiple exclusive, or both shared and exclusive, locks may not exist simultaneously on a file.

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

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

## NOTES

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

locks the file, the parent will lose its lock.

Processes blocked awaiting a lock may be awakened by signals.

## RETURN VALUE

Zero is returned on success, −1 on error, with an error code stored in `errno`.

## ERRORS

The `flock` call fails if:

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

## SEE ALSO

`close`(2), `dup`(2), `execve`(2), `fcntl`(2), `fork`(2), `open`(2), `lockf`(3).

## BUGS

Locks obtained through the `flock` mechanism are known only within the system on which they were placed. Thus, multiple clients may successfully acquire exclusive locks on the same remote file. If this behavior is not explicitly desired, the `fcntl`(2) or `lockf`(3) system calls should be used instead.

## NAME

fork — create a new process

## SYNOPSIS

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

## DESCRIPTION

fork causes creation of a new process. The new process, or child process, is an exact copy of the calling process or parent process. The child process inherits the following attributes from the parent process:

environment
close-on-exec flag (see exec(2))
signal-handling settings (such as SIG_DFL, SIG_IGN, function address)
set-user-ID mode bit
set-group-ID mode bit
process compatibility flags (see setcompat(2))
profiling on/off status
access groups (see getgroups(2))
nice value (see nice(2))
all attached shared-memory segments (see shmop(2))
process group ID
tty group ID (see exit(2) and signal(3))
trace flag (see ptrace(2) request 0)
time left until an alarm clock signal (see alarm(2))
current working directory
root directory
file-mode-creation mask (see umask(2))
file size limit (see ulimit(2))
phys regions (see phys(2)).

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

The child process has a unique process ID.

The child process has a different parent process ID (that is, the process ID of the parent process).

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

parent's open directory streams. The child and parent share directory stream positioning.

The set of signals pending for the child process is cleared.

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

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

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

## RETURN VALUE

On 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.

## ERRORS

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

| | |
|---|---|
| [EAGAIN] | The system-imposed limit on the total number of processes under execution was exceeded. |
| [EAGAIN] | The system-imposed limit on the total number of processes under execution by a single user was exceeded. |
| [EAGAIN] | The system has temporarily exhausted its available memory or swap space. |

## SEE ALSO

exec(2), nice(2), phys(2), plock(2), ptrace(2), semop(2), setcompat(2), shmop(2), times(2), wait(2), wait3(2N), signal(3).

## NAME

fsmount — mount a network file system (NFS)

## SYNOPSIS

```
#include <sys/mount.h>
int fsmount(type, dir, flags, data)
int type;
char *dir;
int flags;
caddr_t data;
```

## DESCRIPTION

fsmount attaches a file system to a directory. After a successful return, references to directory *dir* refer to the root directory on the newly mounted file system. *dir* is a pointer to a null-terminated string containing a pathname. *dir* must exist already and it must be a directory. Its old contents are inaccessible while the file system is mounted.

The argument *flags* determines if the file system can be written on and if set-user-ID execution is allowed. Physically write-protected and magnetic-tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

*type* indicates the type of the file system. It must be one of the types defined in mount.h. *data* is a pointer to a structure that contains the type-specific arguments to mount. Below is a list of the file-system types supported and the type-specific arguments to each:

```
MOUNT_UFS
struct ufs_args {
  char  *fspec;            /* block special file
                           /* to mount */
};

MOUNT_NFS
#include <nfs/nfs.h>
#include <netinet/in.h>
struct nfs_args {
  struct sockaddr_in *addr; /* file server address */
  fhandle_t *fh;            /* file handle to be
                            /* mounted */
  int flags;                /* flags */
```

```
    int wsize;              /* write size in bytes */
    int rsize;              /* read size in bytes */
    int timeo;              /* initial timeout in
                            /* .1 secs */
    int retrans;            /* times to retry send */
};
```

RETURN VALUE

`fsmount` returns 0 if the action occurred and returns –1 if *special* is inaccessible or not an appropriate file, if *name* does not exist, if *special* is already mounted, if *name* is in use, or if too many file systems are already mounted.

ERRORS

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

| | |
|---|---|
| [EPERM] | The caller is not the superuser. |
| [ENOTBLK] | *special* is not a block device. |
| [ENXIO] | The major device number of *special* is out of range, indicating no device driver exists for the associated hardware. |
| [EBUSY] | *dir* is not a directory, or another process currently holds a reference to it. |
| [EBUSY] | No space remains in the mount table. |
| [EBUSY] | The super block for the file system had a bad magic number or an out of range block size. |
| [EBUSY] | Not enough memory was available to read the cylinder group information for the file system. |
| [ENOTDIR] | A component of the path prefix in *special* or *name* is not a directory. |
| [ENAMETOOLONG] | The pathname of *special* or *name* was too long. |
| [ENOENT] | *special* or *name* does not exist. |
| [EACCES] | Search permission is denied for a component of the path prefix of *special* or *name*. |
| [EFAULT] | *special* or *name* points outside of the allocated address space of the process. |

| | |
|---|---|
| [ELOOP] | Too many symbolic links were encountered in translating the pathname of *special* or *name*. |
| [EIO] | An I/O error occurred while reading from or writing to the file system. |
| [ENOMEM] | Memory could not be allocated for cylinder group information. |
| [EINVAL] | Bad magic number or block size exceeds MAXBSIZE. |

**SEE ALSO**

unmount(2), umount(2), mount(3).

**BUGS**

Too many errors appear to the caller as one value.

## NAME

fsync — synchronize a file's in-core state with that on disk

## SYNOPSIS

```
int fsync (fd)
int fd;
```

## DESCRIPTION

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

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

## RETURN VALUE

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

## ERRORS

fsync fails if:

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

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

## SEE ALSO

sync(1), sync(2).

## BUGS

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

## NAME

getdirentries — get directory entries

## SYNOPSIS

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

int getdirentries(d, buf, nbytes, basep)
int d;
char *buf;
int nbytes;
long *basep;
```

## DESCRIPTION

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

The data in the buffer is a series of direct structures. The direct structure is defined as

```
struct direct {
        unsigned long   d_fileno;
        unsigned short  d_reclen;
        unsigned short  d_namlen;
        char            d_name[MAXNAMELEN + 1];
};
```

The d_fileno entry is a number which is unique for each distinct file in the file system. Files that are linked by hard links (see link(2)) have the same d_fileno. The d_reclen entry is the length, in bytes, of the directory record. The d_name and d_namelen entries specify the actual file name and its length.

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

turned in *basep* or zero.

RETURN VALUE

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

SEE ALSO

`link`(2), `lseek`(2), `open`(2), `stat`(2), `directory`(3).

## NAME

getdomainname, setdomainname — get/set name of current network domain

## SYNOPSIS

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

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

## DESCRIPTION

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

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

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

## RETURN VALUE

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

## ERRORS

The following errors may be returned by these calls:

| | |
|---|---|
| [EFAULT] | The *name* or *namelen* parameter gave an invalid address. |
| [EPERM] | The caller was not the superuser. |

## BUGS

Domain names are limited to 255 characters.

**NAME**

getdtablesize — get descriptor table size

**SYNOPSIS**

```
int getdtablesize()
```

**DESCRIPTION**

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

**SEE ALSO**

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

## NAME

getgroups — get group access list

## SYNOPSIS

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

int getgroups(gidsetlen, gidset)
int gidsetlen, *gidset;
```

## DESCRIPTION

getgroups gets the current group access list of the user process and stores it in the array *gidset*. The parameter *gidsetlen* indicates the number of entries that may be placed in *gidset*.

getgroups returns the actual number of groups returned in *gidset*. No more than NGROUPS, as defined in <sys/param.h>, will ever be returned.

## RETURN VALUE

A successful call returns the number of groups in the group set. A value of −1 indicates that an error occurred and the error code is stored in the global variable errno.

## ERRORS

The possible errors for getgroups are

| | |
|---|---|
| [EINVAL] | The argument *gidsetlen* is smaller than the number of groups in the group set. |
| [EFAULT] | The argument *gidset* specifies an invalid address. |

## SEE ALSO

setgroups(2), initgroups(3X).

## BUGS

The *gidset* array should be of type gid_t, but remains integer for compatibility with earlier systems.

## NAME

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

## SYNOPSIS

```
int gethostid()

int sethostid(hostid)
int hostid;
```

## DESCRIPTION

sethostid establishes a 32-bit identifier for the current processor. This identifier is intended to be unique among all systems in existence and is normally a DARPA Internet address for the local machine. This call is allowed only to the superuser and is normally performed at boot time.

## RETURN VALUE

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

sethostid returns zero upon successful completion and −1 upon error.

## SEE ALSO

hostid(1N), gethostname(2N).

## BUGS

32 bits for the identifier is too small.

## NAME

gethostname, sethostname — get/set name of current host

## SYNOPSIS

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

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

## DESCRIPTION

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

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

## RETURN VALUE

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

## ERRORS

The following errors may be returned by these calls:

| | |
|---|---|
| [EFAULT] | The *name* or *namelen* parameter gave an invalid address. |
| [EPERM] | The caller was not the superuser. |

## SEE ALSO

gethostid(2N).

## BUGS

Host names are limited to 255 characters.

## NAME

getitimer, setitimer — get/set value of interval timer

## SYNOPSIS

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

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

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

## DESCRIPTION

The system provides each process with three interval timers, defined in <sys/time.h>. The getitimer call returns the current value for the timer specified in *which* in the structure at *value*. The setitimer call sets a timer to the specified *value* (returning the previous value of the timer if *ovalue* is nonzero).

A timer value is defined by the itimerval structure:

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

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

Time values smaller than the resolution of the system clock are rounded up to this resolution (16 milliseconds on this system, 10 milliseconds on the VAX).

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

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

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

## NOTES

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

## RETURN VALUE

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

## ERRORS

The possible errors are:

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

## SEE ALSO

sigvec(2), gettimeofday(2).

## NAME

getpeername — get name of connected peer

## SYNOPSIS

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

## DESCRIPTION

getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

## RETURN VALUES

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

## ERRORS

getpeername fails if:

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

## SEE ALSO

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

## NAME

getpid, getpgrp, getppid — get process, process group, or parent process IDs

## SYNOPSIS

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

pid_t getpid()

pid_t getpgrp()

pid_t getppid()
```

## DESCRIPTION

getpid returns the process ID of the calling process. Each active process in the system is uniquely identified by a positive integer. The range of this integer is from 1 to the system-imposed limit, or PID_MAX.

getpgrp returns the process group ID of the calling process. Each active process is a member of a process group that is identified by a positive integer. This grouping permits the signaling of related processes.

getppid returns the parent process ID of the calling process. The parent process ID is the process ID of its creator.

## RETURN VALUE

| | |
|---|---|
| getpid | Returns the process ID of the calling process. |
| getpgrp | Returns the process group ID of the calling process. |
| getppid | Returns the parent process ID of the calling process. |

These system calls are useful for generating uniquely named temporary files.

## SEE ALSO

exec(2), fork(2), gethostid(2N), intro(2), setpgrp(2), signal(3).

## NAME

getsockname — get socket name

## SYNOPSIS

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

## DESCRIPTION

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

## RETURN VALUES

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

## ERRORS

getsockname fails if:

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

## SEE ALSO

bind(2N), getpeername(2N), getsockopt(2N), socket(2N).

## NAME

getsockopt, setsockopt — get and set options on sockets

## SYNOPSIS

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

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

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

## DESCRIPTION

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

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

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

*optname* and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file <sys/socket.h> contains definitions for "socket" level options; see socket(2N). Options at other protocol levels vary in format and name; consult the appropriate entries in Section 5 of this manual (appropriate entries are marked (5P)).

**RETURN VALUE**

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

**ERRORS**

The callls fail if:

| | |
|---|---|
| [EBADF] | The argument *s* is not a valid descriptor. |
| [ENOTSOCK] | The argument *s* is a file, not a socket. |
| [ENOPROTOOPT] | The option is unknown. |
| [EFAULT] | The options are not in a valid part of the process address space. |

**SEE ALSO**

getsockname(2N), socket(2N), getprotoent(3N).

## NAME

gettimeofday, settimeofday — get/set date and time

## SYNOPSIS

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

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

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

## DESCRIPTION

The system's notion of the current Greenwich time and the current time zone is obtained with the gettimeofday call and set with the settimeofday call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware dependent, and the time may be updated continuously or in "ticks." If *tzp* is zero, the time zone information will not be returned or set.

The structures referenced by *tp* and *tzp* are defined in <sys/time.h> as:

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

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

The timezone structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving Time applies locally only when Daylight Savings Time is in effect.

Only the superuser may set the time of day or time zone. Changes to the time zone structure are effective for the current process only.

**RETURN VALUE**

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

**ERRORS**

The calls fail if:

| | |
|---|---|
| `[EFAULT]` | An argument address referenced invalid memory. |
| `[EPERM]` | A user other than the superuser attempted to set the time. |

**SEE ALSO**

`date`(1), `adjtime`(2), `time`(2), `stime`(2), `ctime`(3).

## NAME

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

## SYNOPSIS

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

uid_t getuid()

uid_t geteuid()

uid_t getgid()

uid_t getegid()
```

## DESCRIPTION

getuid returns the real user ID of the calling process The real user ID is a positive integer by which each user allowed on the system is identified.

geteuid returns the effective user ID of the calling process. Each active process has an effective user ID that is equal to the process's real user ID unless the process of one of its ancestors evolved from a fail that had the set-user-ID bit set (see exec(2)).

getgid returns the real group ID of the calling process. A real group ID is a positive integer that identifies each user as a member of a group.

getegid returns the effective group ID of the calling process. Each active process has an effective group ID that is equal to the process's real group ID unless the process of one of its ancestors evolved from a fail that had the set-group-ID bit set (see exec(2)).

## RETURN VALUE

| | |
|---|---|
| getuid | Returns the real user ID of the calling process. |
| geteuid | Returns the effective user ID of the calling process. |
| getgid | Returns the real group ID of the calling process. |
| getegid | Returns the effective group ID of the calling process. |

**SEE ALSO**

intro(2), setreuid(2), setuid(2).

## NAME

ioctl — control device

## SYNOPSIS

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

## DESCRIPTION

ioctl performs a variety of functions on character special files (devices). Section 7 of the *A/UX System Administrator's Reference* describes the ioctl requests that apply to the given device.

## RETURN VALUE

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

## ERRORS

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

| | |
|---|---|
| [EBADF] | *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 7 of the *A/UX System Administrator's Reference*. |
| [EINTR] | A signal was caught during the ioctl system call. |

## SEE ALSO

intro(2), fcntl(2), intro(7). termio(7).

## NAME

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

## SYNOPSIS

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

## DESCRIPTION

kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in signal(3), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

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

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

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

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

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

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

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

## RETURN VALUE

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

**ERRORS**

`kill` will fail and no signal will be sent if one or more of the following is 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 sending process is not sending to itself, its effective user ID is not the superuser, and its real or effective user ID does not match the real or effective user ID of the receiving process. |

**SEE ALSO**

`kill`(1), `getpid`(2), `setpgrp`(2), `sigvec`(2), `signal`(3).

## NAME

link — link to a file

## SYNOPSIS

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

## DESCRIPTION

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

## RETURN VALUE

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

## ERRORS

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

| | |
|---|---|
| [ENOTDIR] | A component of either path prefix is not a directory. |
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOENT] | 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 the superuser. |
| [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. |

**SEE ALSO**

symlink(2), unlink(2).

NAME

listen — listen for connections on a socket

SYNOPSIS

```
listen(s, backlog)
int s, backlog;
```

DESCRIPTION

To accept connections, a socket is first created with socket(2N), a backlog for incoming connections is specified with listen(2N) and then the connections are accepted with accept(2N). The listen call applies only to sockets of type SOCK_STREAM or SOCK_PKTSTREAM.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to.

RETURN VALUE

A 0 return value indicates success; −1 indicates an error.

ERRORS

listen will fail if:

| | |
|---|---|
| [EBADF] | The argument *s* is not a valid descriptor. |
| [ENOTSOCK] | The argument *s* is not a socket. |
| [EOPNOTSUPP] | The operation is not supported on a socket. |

If a connection request arrives with the queue full the client will receive an error with an indication of ECONNREFUSED. The socket is not of a type that supports the operation listen.

SEE ALSO

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

BUGS

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

## NAME

`locking` — provide exclusive file regions for reading or writing

## SYNOPSIS

```
int locking(fildes, mode, size)
int fildes;
int mode;
int size;
```

## DESCRIPTION

`locking` will allow a specified number of bytes to be accessed only by the locking process (mandatory locking). Other processes which attempt to lock, read, or write the locked area will sleep until the area becomes unlocked. (Advisory locking is available via `lockf`(3C)).

*fildes* is the word returned from a successful `open`, `creat`, `dup`, or `pipe` system call.

*mode* is zero to unlock the area. *mode* is one or two for making the area locked. If the *mode* is one and the area has some other lock on it, then the process will sleep until the entire area is available. If the *mode* is two and the area is locked, an error will be returned.

*size* is the number of contiguous bytes to be locked or unlocked. The area to be locked starts at the current offset in the file. If *size* is zero, the area to the end of file is locked.

The potential for a deadlock occurs when a process controlling a locked area is put to sleep by accessing another process's locked area. Thus calls to `locking`, `read`, or `write` scan for a deadlock prior to sleeping on a locked area. An error return is made if sleeping on the locked area would cause a deadlock.

Lock requests may, in whole or part, contain or be contained by a previously locked area for the same process. When this or adjacent areas occur, the areas are combined into a single area. If the request requires a new lock element with the lock table full, an error is returned, and the area is not locked.

Unlock requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining areas are still locked by the process. Release of the center section of a locked area requires an additional lock element to hold the cut off section. If the lock table is full, an error is returned, and the requested area is not released.

While locks may be applied to special files or pipes, read/write operations will not be blocked. Locks may not be applied to a directory.

Note that close(2) automatically removes any locks that were associated with the closed file descriptor.

## RETURN VALUE

The value −1 is returned if the file does not exist, or if a deadlock using file locks would occur.

## ERRORS

locking will fail if the following are true:

| | |
|---|---|
| [EACCES] | The area is already locked by another process. |
| [EDEADLOCK] | Returned by read, write, or locking if a deadlock would occur. |
| [EDEADLOCK] | Locktable overflow. |
| [EREMOTE] | *fildes* is a file descriptor that refers to file on a remotely mounted file system. |

## SEE ALSO

close(2), creat(2), dup(2), open(2), read(2), write(2), lockf(3C).

## NAME

lseek — move read/write file pointer

## SYNOPSIS

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

#include <unistd.h>

off_t lseek(fildes, offset, whence)
int fildes;
off_t offset;
int whence;
```

## DESCRIPTION

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

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

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

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

In the POSIX environment, the following values are defined in <unistd.h> for passing as the value of *whence*:

| | |
|---|---|
| SEEK_SET | 0 |
| SEEK_CUR | 1 |
| SEEK_END | 2 |

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

## RETURN VALUE

On 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.

## ERRORS

`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] | *whence* is not 0, 1, or 2. |
| [EINVAL] | The resulting file pointer would be negative. |

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

**SEE ALSO**

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

## NAME

mkdir — make a directory file

## SYNOPSIS

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

## DESCRIPTION

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

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

The newly-created directory will contain entries for . and . . .

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

## RETURN VALUE

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

## ERRORS

mkdir will fail and no directory will be created if:

| | |
|---|---|
| [EEXIST] | The named file exists. |
| [EFAULT] | *path* points outside the process's allocated address space. |
| [EIO] | An I/O error occurred while writing to the file system. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ENOENT] | A component of the path prefix does not exist. |
| [ENOTDIR] | A component of the path prefix is not a directory. |

| | |
|---|---|
| [EPERM] | The *path* argument contains a byte with the high-order bit set. |
| [EROFS] | The named file resides on a read-only file system. |

**SEE ALSO**

mkdir(1), chmod(2), rmdir(2), stat(2), umask(2).

## NAME

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

## SYNOPSIS

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

## DESCRIPTION

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

0170000 file type mask; one of the following:

0010000 FIFO special
0020000 character special
0040000 directory
0060000 block special
0100000 or 0000000 ordinary file
0120000 symbolic link
0140000 socket
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 of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process.

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

mknod may be invoked only by the superuser for file types other than FIFO special.

RETURN VALUE

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

ERRORS

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

| | |
|---|---|
| [EPERM] | The effective user ID of the process is not superuser. |
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [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. |

SEE ALSO

mkdir(1), mknod(1), chmod(2), exec(2), stat(2), umask(2), fs(4), stat(5).

## NAME

msgctl — message control operations

## SYNOPSIS

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

int msgctl(id, cmd, buf)
int id, cmd;
struct msqid_ds *buf;
```

## DESCRIPTION

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

IPC_STAT — Place the current value of each member of the data structure associated with *id* into the structure referenced by *buf*. The contents of this structure are defined in intro(2).

IPC_SET — Set the value of the following members of the data structure associated with *id* to the corresponding value found in the structure referenced by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode (only low 9 bits)
msg_qbytes
```

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

IPC_RMID — Remove the message queue identifier specified by *id* 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 *id*. The identifier and its associated data structure are not actually removed until

there are no more referencing processes. See ipcrm(1), and ipcs(1).

**RETURN VALUE**

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

**ERRORS**

msgctl will fail if one or more of the following is true.

| | |
|---|---|
| [EINVAL] | *id* is not a valid message queue identifier. |
| [EINVAL] | *cmd* is not a valid command. |
| [EACCES] | *cmd* is equal to IPC_STAT and operation permission is denied to the calling process (see intro(2)). |
| [EPERM] | *cmd* is equal to IPC_RMID or IPC_SET. The effective user ID of the calling process is not equal to that of superuser and it is not equal to the value of msg_perm.uid in the data structure associated with *id*. |
| [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 superuser. |
| [EFAULT] | *buf* points to an illegal address. |

**SEE ALSO**

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

## NAME

msgget — get message queue

## SYNOPSIS

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

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

## DESCRIPTION

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

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

*key* is equal to IPC_PRIVATE.

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

The key IPC_PRIVATE will create an identifier and associated data structure that is unique to the calling process and its children.

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 equal to 0.

msg_ctime is set equal to the current time.

msg_qbytes is set equal to the system limit.

## RETURN VALUE

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.

**ERRORS**

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

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

`[ENOENT]` A message queue identifier does not exist for *key* and (*msgflg* `&` `IPC_CREAT`) is "false".

`[ENOSPC]` A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.

`[EEXIST]` A message queue identifier exists for *key* but `(`(*msgflg* `&` `IPC_CREAT`) `&&` (*msgflg* `&` `IPC_EXCL`)`)` is "true".

**SEE ALSO**

`intro(2)`, `msgctl(2)`, `msgop(2)`.

## NAME

msgop, msgsnd, msgrcv — message operations

## SYNOPSIS

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

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

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

## DESCRIPTION

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

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

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

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

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

The total number of messages on all queues systemwide 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(2)). When this occurs, errno is set equal to EIDRM, and a value of −1 is returned.

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

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

msg_qnum is incremented by 1.

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

msg_stime is set equal to the current time.

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

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

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

*msgtyp* specifies the type of message requested as follows:

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

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

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

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

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

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 sigvec(2)).

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

msg_qnum is decremented by 1.

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

msg_rtime is set equal to the current time.

## RETURN VALUES

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.

**ERRORS**

`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`(2)). |
| `[EINVAL]` | *mtype* is less than 1. |
| `[EAGAIN]` | The message cannot be sent for one of the reasons cited above and (*msgflg* & `IPC_NOWAIT`) is "true". |
| `[EINVAL]` | *msgsz* is less than zero or greater than the system-imposed limit. |
| `[EFAULT]` | *msgp* points to an illegal address. |

`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 message of the desired type and (*msgtyp* & `IPC_NOWAIT`) is "true". |
| `[EFAULT]` | *msgp* points to an illegal address. |

**SEE ALSO**

`intro`(2), `msgctl`(2), `msgget`(2), `sigvec`(2), `signal`(3).

## NAME

nfssvc, async_daemon — NFS daemons

## SYNOPSIS

```
int nfssvc(sock)
int sock;

async_daemon()
```

## DESCRIPTION

nfssvc starts an NFS daemon listening on socket *sock*. The socket must be AF_INET, and SOCK_DGRAM (protocol UDP/IP). The system call will return only if the process is killed.

async_daemon implements the NFS daemon that handles asynchronous I/O for an NFS client. The system call never returns.

## BUGS

These two system calls allow kernel processes to have user context.

## SEE ALSO

mountd(1M), nfsd(1M).

## NAME

nfs_getfh — get a file handle

## SYNOPSIS

```
#include <rpc/types.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <nfs/nfs.h>

int nfs_getfh (fildes, fhp)
int fildes;
fhandle_t *fhp;
```

## DESCRIPTION

nfs_getfh returns the file handle associated with the file descriptor *fd*. This call is restricted to the superuser.

## RETURN VALUE

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

## ERRORS

The following errors may be returned by these calls:

| | |
|---|---|
| [EPERM] | The caller was not the superuser. |
| [EBADF] | *fd* is not a valid open file descriptor. |
| [EFAULT] | The *fhp* parameter gave an invalid address. |

## NAME

nice — change priority of a process

## SYNOPSIS

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

## DESCRIPTION

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

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

## RETURN VALUE

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. If a value of −1 is a valid return value on successful completion (i.e., if your new nice value is 19), errno is not changed.

## ERRORS

nice will fail if:

[EPERM] nice will fail and not change the nice value if *incr* is negative or greater than 40 and the effective user ID of the calling process is not superuser.

## SEE ALSO

nice(1), exec(2).

## NAME

open — open for reading or writing

## SYNOPSIS

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

## DESCRIPTION

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

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY or O_NONBLOCK

Either flag may affect subsequent reads and writes. See read(2) and write(2).

When opening a FIFO with O_RDONLY or O_WRONLY set and

if O_NDELAY or O_NONBLOCK is set, an open for reading-only returns without delay. An open for writing-only returns an error if no process currently has the file open for reading.

if O_NDELAY and O_NONBLOCK are clear, an open for reading-only blocks until a process opens the file for writing. An open for writing-only blocks until a process opens the file for reading. When opening a file associated with a communication line and

if O_NDELAY or O_NONBLOCK is set, the open returns without waiting for a carrier.

if O_NDELAY and O_NONBLOCK are clear, an open blocks until a carrier is present.

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

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

All bits set in the file-mode-creation mask of the process are cleared. See umask(2).

Mode bit 01000 (save text image after execution bit) is cleared. See chmod(2).

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

O_EXCL — If O_EXCL and O_CREAT are set, open fails if the file exists.

O_NOCTTY — If set, and *path* identifies a terminal device, the open does not cause the terminal device to become the controlling terminal for the process. O_NOCTTY has been added for compliance with POSIX.

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

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

## RETURN VALUE

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

## ERRORS

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.

| | |
|---|---|
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOENT] | O_CREAT is not set, and the named file does not exist. |
| [EACCES] | A component of the path prefix denies search permission. |
| [EACCES] | An *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. |
| [EMFILE] | The per-process open file limit would be exceeded. |
| [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.<br>*Note:* If you are running an network file system (NFS) and you are accessing a shared binary remotely, it is possible that you will not get this errno. |
| [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. |
| [ENOSPC] | The directory or file system that would contain the new file cannot be extended. |

**SEE ALSO**

chmod(2), close(2), creat(2), dup(2), fcntl(2), lseek(2), read(2), umask(2), write(2), fopen(3), ferror(3).

## NAME

pause — suspend process until signal

## SYNOPSIS

```
pause()
```

## DESCRIPTION

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

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

When a signal is caught by the calling process, the behavior of pause will vary according to flags set by setcompat(2) or set42sig(3). If the COMPAT_SYSCALLS flag is set when control is returned from the signal catching function, then the process will once again pause; otherwise, the calling process will resume as described below.

## ERRORS

If the signal is caught by the calling process and control is returned from the signal-catching function (see signal(3)), the calling process resumes execution from the point of suspension; the return value of pause will be set to −1 and errno will be set to EINTR.

## SEE ALSO

alarm(2), kill(2), wait(2), signal(3).

## NAME

phys — allow a process to access physical addresses

## SYNOPSIS

```
int phys (physnum, virtaddr, size, physaddr)
int physnum;
char *virtaddr;
unsigned int size;
char *physaddr;
```

## DESCRIPTION

The phys system call allows the superuser to map a region of physical memory into a process's virtual address space.

The calling process chooses *physnum* to specify the phys region this call references. The maximum number of regions per process is defined by the v_phys field in the var structure returned by uvar(2). *physnum* must be between zero and v_phys−1, and is only used to identify a particular phys region to the kernel during a phys system call.

*virtaddr* is the base virtual address for the region in the process's virtual address space, and *size* is the length in bytes of the desired region. The virtual address range of the region must not overlap any of the existing address space of the process, including text, data, stack, shared memory regions (see shmget(2)), and any other active phys regions. All addresses in this range must be valid user virtual addresses (see the example below). Care should also be taken to avoid placing a phys region at a virtual address that the data or stack segments might grow to encompass.

If *size* is zero, any previous phys mapping is cleared for the region specified by *physnum*.

A phys region's *virtaddr* and *size* are dependent on the implementation decisions for the memory management unit. In particular, the base *virtaddr* must be on a kernel segment boundary and the *size* will be rounded up to an integral multiple of the page size. These values may be computed from the v_segshift and v_pageshift fields returned by uvar(2); that is, the segment size is

```
1 << v_segshift
```

and the page size is

```
1 << v_pageshift
```

The *physaddr* argument is the base physical address for the region. *physaddr* is rounded down to the previous page boundary. Also, *physaddr* to *physaddr+size* should be inside the range of physical addresses supported by the hardware. `phys` regions are inherited across `fork`(2) system calls and disowned across `exec`s.

`phys` may only be executed by a process with an effective user ID of root.

As an example, suppose a process wishes to map a piece of memory-mapped hardware into its address space. This hardware has 0x8800 bytes of memory and control registers located at physical address 0xFA000000. By calling `uvar`(2), the process finds that `v_pageshift` is 12 and `v_segshift` is 20; thus, the page size is 0x1000 and the segment size is 0x100000. Also, `v_phys` is found to be 32, so any number from zero to 31 may be used for *physnum*.

The `var` structure also contains `v_ustart` and `v_uend`, the starting and ending virtual addresses for user processes. For this example, assume `v_ustart` is zero and `v_uend` is 0x20000000. The first few segments are used for the running program's text and data and the last are used for the user stack. The process might decide it is unlikely that its data and text segment will exceed 0x4000000, which is an integral multiple of 0x100000 (the segment size).

The call:

```
phys(0, 0x4000000, 0x8800, 0xFA000000);
```

will allow the process access to physical locations from 0xFA00000 to 0xFA009000 by referencing virtual addresses 0x4000000 to 0x4009000. The range has been adjusted to 0x9000 bytes because that is the next page boundary.

In this example, referencing 0x4008804 (an address in the `phys` region, but outside of the known hardware memory) will result in unpredictable failures. A useless value may be read off the hardware lines, a write may appear to succeed without affecting anything, the program may get a `SIGSEGV` (see `signal`(3)), the hardware may react randomly, or the entire system may crash. There may be other possibilities depending on system configuration.

If the process wished to add another `phys` region without deleting the first region, the next available *virtaddr* would be 0x4100000 (the next segment boundary) and *physnum* could be any number from one to 31.

## RETURN VALUES

The value zero is returned if the call was successful; otherwise −1 is returned. `phys` will fail if the effective user ID of the calling process is not root, if *virtaddr* or *physaddr* is not in the proper range, or if the range of virtual addresses overlaps a portion of the user's virtual address space that is already in use.

## NOTES

`phys` is hardware and implementation dependent and must be used with extreme caution. The intention is to give the superuser complete access to the physical hardware. To insure maximum portability, *virtaddr* and *size* should be calculated as described in the example.

Different hardware may respond differently to mistakes in addressing. Sometimes all the bits of a physical address are not decoded, making (for example) 0xFD100000 the same as 0xFD000000. If *physaddr* or *size* is wrong it is possible to crash the system.

Most versions of UNIX do not support this system call.

## SEE ALSO

`uvar`(2), `shmget`(2), `signal`(3).

## NAME

pipe — create an interprocess channel

## SYNOPSIS

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

## DESCRIPTION

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

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

## RETURN VALUE

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

## ERRORS

pipe will fail if one or more of the following is true.

[EMFILE] pipe will fail if the per-process open file limit would be exceeded.

[ENFILE] The system file table is full.

## SEE ALSO

read(2), write(2).

## NAME

plock — lock process, text, or data in memory

## SYNOPSIS

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

int plock(op)
int op;
```

## DESCRIPTION

plock allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. plock also allows these segments to be unlocked. The effective user ID of the calling process must be superuser 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 |

## RETURN VALUE

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.

## ERRORS

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

| | |
|---|---|
| [EPERM] | The effective user ID of the calling process is not superuser. |
| [EAGAIN] | The system has temporarily exhausted its available memory or swap space. |
| [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.

**SEE ALSO**

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

## NAME

profil — execution time profile

## SYNOPSIS

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

## DESCRIPTION

profil is used to report performance analysis of an application. *buff* points to an area of core whose length (in bytes) is given by *bufsiz*. After the call, the user's program counter (pc) is examined for each clock tick; *offset* is subtracted from it, and the result 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 16 bits of fraction: 0x10000 gives a 1–1 mapping of pc's to words in *buff*; 0x8000 maps each pair of instruction words together; 2 maps all instructions onto the beginning of buff (producing a noninterrupting 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.

## RETURN VALUE

Not defined.

## SEE ALSO

prof(1), monitor(3C).

## NAME

ptrace — process trace

## SYNOPSIS

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

## DESCRIPTION

ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging. The child process behaves normally until it encounters a signal (see sigvec(2) for the list), at which time it enters a stopped state and its parent is notified via wait(2). When the child is in the stopped state, its parent can examine and modify its "core image" using ptrace. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

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

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

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

1, 2
With these requests, the word at location *addr* in the address space of the child is returned to the parent process. 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. Ad-

dresses are system dependent. 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*. 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:

the general registers
the condition codes
certain bits of the Processor Status Word

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

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

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

stepping of the child.

*Note:* The trace bit remains set after an interrupt.

10 Read user register; *pid* = child process ID; *addr* = register number; *data* is ignored; returns value of child's register.

11 Write user register; *pid* = child process ID; *addr* = register number; *data* = integer value to be written into named register.

*Note:* For both requests 10 and 11, the register numbers are as shown below for the 68000-family (these numbers are system dependent).

| Register | Register # | Register | Register # |
|---|---|---|---|
| d0 | 0 | a1 | 9 |
| d1 | 1 | a2 | 10 |
| d2 | 2 | a3 | 11 |
| d3 | 3 | a4 | 12 |
| d4 | 4 | a5 | 13 |
| d5 | 5 | a6 | 14 |
| d6 | 6 | SP | 15 |
| d7 | 7 | PC | 16 |
| a0 | 8 | PS | 17 |

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

ERRORS

`ptrace` will in general 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`.

NOTES

Request 11 largely supercedes request 6, and request 10 largely supercedes request 3 (request 3 can read any part of the child's user area while request 10 can only read register values of the child).

**SEE ALSO**

exec(2), sigvec(2), wait(2), signal(3).

## NAME

read, readv — read from file

## SYNOPSIS

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

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

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

## DESCRIPTION

read attempts to read *nbytes* bytes from the file associated with *fildes* into the buffer pointed to by *buf*. readv performs the same action but scatters the input data into the *iovcnt* buffers specified by the members of the *iovec* array:

iov[0], iov[1], ..., iov[*iovcnt*-1]

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

For readv, the iovec structure is defined as

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

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

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

On devices incapable of seeking, reading always starts from the current position. The value of a file pointer associated with such a file is undefined.

On successful completion, read and readv return the number of bytes actually read and placed in the buffer, this number may be less than *nbytes* if the file is associated with a communication line (see ioctl(2), socket(2N), and termio(7)) or if the number of bytes left in the file is less than *nbytes*. A value of 0 is returned when an end-of-file has been reached. If *nbyte* is 0, read returns 0 and has no other result.

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

if O_NDELAY is set, the read returns 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.

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

if O_NDELAY is set, the read returns 0.

if O_NDELAY is clear, the read blocks until data becomes available.

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

if O_NONBLOCK is set, the read returns −1 and sets errno to EAGAIN.

if O_NONBLOCK is clear, the read blocks until some data is written or the pipe is closed by all processes that had the pipe open for writing.

if no process has the pipe open for writing, the read returns 0 to indicate end-of-file.

When attempting to read a file associated with a terminal that has no data currently available and

if O_NONBLOCK is set, the read returns −1 and set errno to EAGAIN.

if O_NONBLOCK is clear, the read blocks until data becomes available.

RETURN VALUE

On successful completion, a non-negative integer is returned indicating the number of bytes actually read. If the process compatibility flag COMPAT_SYSCALLS is set (see setcompat(2)), as in the POSIX environment, and read is interrupted by a signal after successfully reading some data, it returns the number of bytes read. Otherwise, a −1 is returned and errno is set to

EINTR to indicate the error.

**ERRORS**

When attempting to read from a stream that has no data currently available and if O_NDELAY is set, read returns −1 and errno is set to ENODATA. If O_NDELAY is clear, read blocks until data becomes available.

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

| | |
|---|---|
| [EIO] | A physical I/O error has occurred or the process is in a background process group and is attempting to read from its controlling terminal and the process is ignoring or blocking SIGTTIN or the process group of the process is orphaned. |
| [ENXIO] | The device associated with the file descriptor is a block-special or character-special file and the value of the file pointer is out of range. |
| [EWOULDBLOCK] | The file was marked for nonblocking I/O, and no data was ready to be read. |
| [EBADF] | The file descriptor *fildes* is not valid and open for reading. |
| [EFAULT] | *buf* points outside the allocated address space. |
| [EINTR] | A signal was caught during the read system call. |
| [ENODATA] | A read from a stream was attempted when no data was available and O_NDELAY was set. |

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

| | |
|---|---|
| [EINVAL] | *iovcnt* was less than or equal to 0, or greater than 16. |
| [EINVAL] | One of the iov_len values in the iov array was negative. |
| [EINVAL] | The sum of the iov_len values in the iov array overflowed a 32-bit integer. |

**SEE ALSO**

creat(2), fcntl(2), ioctl(2), open(2), pipe(2), socket(2N), setcompat(2), termio(7).

## NAME

readlink — read value of a symbolic link

## SYNOPSIS

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

## DESCRIPTION

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

## RETURN VALUE

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

## ERRORS

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

| | |
|---|---|
| [EPERM] | The *path* argument contained a byte with the high-order bit set. |
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOENT] | The pathname was too long. |
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [ENOENT] | The named file does not exist. |
| [ENXIO] | The named file is not a symbolic link. |
| [EACCES] | Search permission is denied on a component of the path prefix. |
| [EPERM] | The effective user ID does not match the owner of the file and the effective user ID is not the superuser. |

| | |
|---|---|
| [EINVAL] | The named file is not a symbolic link. |
| [EFAULT] | *buf* extends outside the process's allocated address space. |
| [ELOOP] | Too many symbolic links were encountered in translating the pathname. |

**SEE ALSO**

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

## NAME

reboot — reboot system or halt processor

## SYNOPSIS

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

int reboot(howto)
int howto;
```

## DESCRIPTION

reboot halts and restarts the processor and reloads the operating system from the disk file "unix" on the root file system of the default boot device. By default, in-core data for mounted file systems is flushed, as in sync(2). Only the superuser may reboot a machine.

howto is a mask of options passed to the operating system. One of the following bits must be set in howto:

RB_AUTOBOOT
: Restart the processor and reload the operating system.

RB_HALT
: Halt the processor. No reboot takes place.

RB_BUSYLOOP
: Hang the processor an infinite loop.

The following optional behavior may be requested by setting these bits in howto:

RB_NOSYNC
: Do not flush the system buffers to disk.

RB_KILLALL
: Terminate all running processes before halting the system.

RB_UNMOUNT
: Unmount all mounted file systems before halting the system.

## RETURN VALUES

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

## ERRORS

The following error condition could occur:

[EPERM] The caller is not the superuser.

**SEE ALSO**

reboot(1M), shutdown(1M), sync(2).

## NAME

recv, recvfrom, recvmsg — receive a message from a socket

## SYNOPSIS

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

int recv(s, buf, len, flags)
int s;
char *buf;
int len, flags;

int recvfrom(s, buf, len, flags, from, fromlen)
int s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

int recvmsg(s, msg, flags)
int s;
struct msghdr msg[];
int flags;
```

## DESCRIPTION

recv, recvfrom, and recvmsg are used to receive messages from a socket.

The recv call may be used only on a connected socket (that is, when connect(2N) has been used), while recvfrom and recvmsg may be used to receive data on a socket whether it is in a connected state or not.

If *from* is nonzero, the source address of the message is filled in. *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from; see socket(2N).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see ioctl(2)) in which case a −1 is returned with the external variable errno set to EWOULDBLOCK.

The select(2N) call may be used to determine when more data arrives.

The *flags* argument to a send call is formed by ORing one or more of the values,

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

The recvmsg call uses a msghdr structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in <sys/socket.h>:

```
struct msghdr {
  caddr_t  msg_name;       /* optional address */
  int      msg_namelen;    /* size of address */
  struct   iov *msg_iov;   /* scatter/gather array */
  int      msg_iovlen;     /* # elements in msg_iov */
  caddr_t  msg_accrights; /* access rights sent/received */
  int      msg_accrightslen;
};
```

Here msg_name and msg_namelen specify the destination address if the socket is unconnected; msg_name may be given as a null pointer if no names are desired or required. The msg_iov and msg_iovlen describe the scatter gather locations. Access rights to be sent along with the message are specified in msg_accrights, which has length msg_accrightslen.

## RETURN VALUE

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

## ERRORS

The calls fail if:

| | |
|---|---|
| [EBADF] | The argument *s* is an invalid descriptor. |
| [ENOTSOCK] | The argument *s* is not a socket. |
| [EWOULDBLOCK] | The socket is marked nonblocking and the receive operation would block. |
| [EINTR] | The receive was interrupted by delivery of a signal before any data was available for the receive. |
| [EFAULT] | The data was specified to be received into a nonexistent or protected part of the process address space. |

**SEE ALSO**

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

## NAME

rename — change the name of a file

## SYNOPSIS

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

## DESCRIPTION

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

rename guarantees that an instance of the file will always exist, even if the system should crash in the middle of the operation.

## CAVEAT

The system can deadlock if a loop in the file system graph is present. This loop takes the form of an entry in directory "a" say a/foo, being a hard link to directory "b", and an entry in directory "b", say b/bar, being a hard link to directory "a". When such a loop exists and two separate processes attempt to perform

```
rename a/foo b/bar
```

and

```
rename b/bar a/foo
```

respectively, the system may deadlock attempting to lock both directories for modification. Hard links to directories should be replaced by symbolic links by the system administrator.

## RETURN VALUE

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

## ERRORS

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

| | |
|---|---|
| [ENOTDIR] | A component of either path prefix is not a directory. |
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire path- |

| | |
|---|---|
| | name exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOENT] | A component of either path prefix does not exist. |
| [EACCES] | A component of either path prefix denies search permission. |
| [ENOENT] | The file named by *from* does not exist. |
| [EPERM] | The file named by *from* is a directory and the effective user ID is not superuser. |
| [EXDEV] | The link named by *to* and the file named by *from* are on different logical devices (file systems). |
| [EACCES] | The requested link requires writing in a directory with a mode that denies write permission. |
| [EROFS] | The requested link requires writing in a directory on a read-only file system. |
| [EFAULT] | *path* points outside the process's allocated address space. |
| [EINVAL] | *from* is a parent directory of *to*. |

**SEE ALSO**

mv(1), open(2).

## NAME

rmdir — remove a directory file

## SYNOPSIS

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

## DESCRIPTION

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

## RETURN VALUE

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

## ERRORS

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

| | |
|---|---|
| [ENOTESMPTY] | The named directory contains files other than . and . . in it. |
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [ENOENT] | The named file does not exist. |
| [EACCES] | A component of the path prefix denies search permission. |
| [EACCES] | Write permission is denied on the directory containing the link to be removed. |
| [EBUSY] | The directory to be removed is the mount point for a mounted file system. |
| [EROFS] | The directory entry to be removed resides on a read-only file system. |
| [EFAULT] | *path* points outside the process's allocated address space. |

**SEE ALSO**

rmdir(1), mkdir(2), unlink(2).

## NAME

select — synchronous I/O multiplexing

## SYNOPSIS

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

int select(nfds, readfds, writefds, execptfds, timeout)
int nfds, *readfds, *writefds, *execptfds;
struct timeval *timeout;
```

## DESCRIPTION

`select` examines the I/O descriptors specified by the bit masks *readfds*, *writefds*, and *execptfds* to see if they are ready for reading, writing, or have an exceptional condition pending, respectively. File descriptor *f* is represented by the bit 1<<*f* in the mask. *nfds* descriptors are checked, that is, the bits from 0 through *nfds*–1 in the masks are examined. `select` returns, in place, a mask of those descriptors which are ready. The total number of ready descriptors is returned.

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

Any of *readfds*, *writefds*, and *execptfds* may be given as 0 if no descriptors are of interest.

## RETURN VALUE

`select` returns the number of descriptors which are contained in the bit masks, or –1 if an error occurred. If the time limit expires then `select` returns 0.

## ERRORS

An error return from `select` indicates:

| | |
|---|---|
| `[EBADF]` | One of the bit masks specified an invalid descriptor. |
| `[EINTR]` | A signal was delivered before any of the selected for events occurred or the time limit expired. |

## SEE ALSO

`accept`(2N), `connect`(2N), `recv`(2N), `readv`(2), `send`(2N), `writev`(2).

## BUGS

The descriptor masks are always modified on return, even if the call returns as the result of the timeout.

## NAME

semctl — semaphore control operations

## SYNOPSIS

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

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

## DESCRIPTION

semctl 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* (see intro(2) for required permissions and structure declarations):

GETVAL Return the value of semval (see intro(2)).

SETVAL Set the value of semval to *arg.val*. When this command is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of sempid.

GETNCNT Return the value of semncnt.

GETZCNT Return the value of semzcnt.

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

GETALL Place semvals into array pointed to by *arg.array*.

SETALL Set semvals according to the array pointed to by *arg.array*. When this command is successfully executed, the semadj values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

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

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 command can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of sem_perm.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 command can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of sem_perm.uid in the data structure associated with *semid*. The identifier and its associated data structure are not actually removed until there are no more referencing processes. See ipcrm(1), and ipcs(1).

## RETURN VALUE

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.

**ERRORS**

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. |
| [EACCES] | Operation permission is denied to the calling process (see intro(2)). |
| [ERANGE] | *cmd* is SETVAL or SETALL and the value to which semval is to be set is greater than the system imposed maximum. |
| [EPERM] | *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of superuser 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. |

**SEE ALSO**

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

## NAME

semget — get set of semaphores

## SYNOPSIS

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

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

## DESCRIPTION

semget returns the semaphore identifier associated with *key*.

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

The key IPC_PRIVATE will create an identifier and associated data structure that is unique to the calling process and its children.

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.

## RETURN VALUE

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.

**ERRORS**

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(2)) as specified by the low-order 9 bits of *semflg* would not be granted.

[EINVAL] A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to zero.

[ENOENT] A semaphore identifier does not exist for *key* and (*semflg* & IPC_CREAT) is "false".

[ENOSPC] A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded.

[EEXIST] A semaphore identifier exists for *key* but ((*semflg* & IPC_CREAT) && (*semflg* & IPC_EXCL)) is "true".

**SEE ALSO**

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

## NAME

semop — semaphore operations

## SYNOPSIS

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

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

## DESCRIPTION

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

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

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

sem_op specifies one of three semaphore operations as follows (see intro(2) for permissions and structure declarations:

If sem_op is a negative integer, one of the following will occur:

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

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

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

The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in signal(3).

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

If sem_op is zero, one of the following will occur:

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(3).

RETURN VALUE

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 semval at the time of the call for the last operation in the array pointed to by *sops* is returned. Otherwise, a value of −1 is returned and errno is set to indicate the error.

ERRORS

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

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

[EFBIG] sem_num is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.

[E2BIG] *nsops* is greater than the system-imposed maximum.

[EACCES] Operation permission is denied to the calling process (see intro(2)).

[EAGAIN] The operation would result in suspension of the calling process but (sem_flg & IPC_NOWAIT) is "true".

[ENOSPC] The limit on the number of individual processes requesting an SEM_UNDO would be exceeded.

[EINVAL] The number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the limit.

[ERANGE] An operation would cause a semval to overflow the system-imposed limit.

[ERANGE] An operation would cause a semadj value to overflow the system-imposed limit.

[EFAULT] *sops* points to an illegal address.

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

**SEE ALSO**

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

## NAME

send, sendto, sendmsg — send a message from a socket

## SYNOPSIS

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

int send(s, msg, len, flags)
int s;
char *msg;
int len, flags;

int sendto(s, msg, len, flags, to, tolen)
int s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;

int sendmsg(s, msg, flags)
int s;
struct msghdr msg[];
int flags;
```

## DESCRIPTION

send, sendto, and sendmsg are used to transmit a message to another socket. send may be used only when the socket is in a connected state (i.e., when connect(2N) has been used), while sendto and sendmsg may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

If no message space is available at the socket to hold the message to be transmitted, then send normally blocks, unless the socket has been placed in nonblocking I/O mode. The select(2N) call may be used to determine when it is possible to send more data.

The *flags* parameter may be set to MSG_OOB to send "out-of-band" data on sockets which support this notion (e.g. SOCK_STREAM).

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

**RETURN VALUE**

The call returns the number of characters sent, or −1 if an error occurred.

No indication of failure to deliver is implicit in a send. Return values of −1 indicate some locally detected errors.

**ERRORS**

| | |
|---|---|
| [EBADF] | An invalid descriptor was specified. |
| [ENOTSOCK] | The argument *s* is not a socket. |
| [EFAULT] | An invalid user space address was specified for a parameter. |
| [EMSGSIZE] | The socket requires that message be sent atomically, and the size of the message to be sent made this impossible. |
| [EWOULDBLOCK] | The socket is marked nonblocking and the requested operation would block. |

**SEE ALSO**

connect(2N), recv(2N), socket(2N).

## NAME

setcompat, getcompat — set or get process compatibility mode

## SYNOPSIS

```
#include <compat.h>

int setcompat(flags)
int flags;

int getcompat();
```

## DESCRIPTION

setcompat sets a process's compatibility mode according to the *flags* argument. The argument governs the type of compatibility enforced. *flags* may be COMPAT_SVID for strictest adherence to the System V interface definition, COMPAT_POSIX to enable all POSIX functionality, or the bitwise OR of one or more of the following symbolic constants. If set, other flags always take precedence over COMPAT_SVID.

| | |
|---|---|
| COMPAT_BSDNBIO | Changes the error handling in 4.2 BSD nonblocking I/O code. read and write system calls on slow devices, that is, terminals, which are marked for non-blocking may return −1 with errno set to EWOULDBLOCK instead of returning 0. (Operations which may block, that is, connect, accept, and recv on sockets which are marked for nonblocking always return an error and set errno to EWOULDBLOCK.) |
| COMPAT_BSDGROUPS | Enables the use of the 4.2 BSD groups code which permits users to be members of more than one group simultaneously and creates files whose group is determined by the group of the directory in which the file is created. |
| COMPAT_BSDSETUGID | When selected, changes the behavior of the setuid and setgid calls to be BSD-compatible; that is, no handling of the saved set-user (group) |

ID from exec. When cleared, the `setreuid` and `setregid` calls behave as `setuid` and `setgid`, respectively.

`COMPAT_BSDSIGNALS` Allows a process to use 4.2 BSD-compatible signals. The state of this flag may not be changed unless no signals are pending, caught, or held. This option enables reliable signal delivery. Caught signals will be held while a signal handler is invoked and reset upon exit from the signal hander.

`COMPAT_BSDTTY` Enables 4.2 BSD job control. When first set, this process and its descendants will be identified as 4.2 processes via a bit in the flag word of the kernel `proc` data structure. Membership in a 4.2 process group persists across `exec` system calls. Jobs that are 4.2 process group members are effected by job control signals. When `COMPAT_BSDTTY` is set the `setpgrp` system call may be used to manipulate the process group of other processes. This flag may only be used in conjunction with the `COMPAT_BSDSIGNALS` flag. Normally, `COMPAT_BSDTTY` is set by a login shell.

`COMPAT_CLRPGROUP` Disables 4.2 BSD job control. Resets the 4.2 process group bit in the flag word of the kernel `proc` data structure. It may be used by a V.2 process which wants to sever any job control associations with an invoking shell (for itself and its descendants). This bit provides a "one shot" clear. When read by `getcompat`, this bit is always zero.

| | |
|---|---|
| COMPAT_EXEC | If this flag is set, compatibility flags are inherited across exec system calls. To provide child process with a System V interface environment, both the COMPAT_SVID and the COMPAT_EXEC flags must be set by ORing the flags. |
| COMPAT_SYSCALLS | If selected, read, write, ioctl, or wait calls which are interrupted by a signal handler will not return an EINTR error, but will instead resume at the point they were interrupted. This flag may only be used in conjunction with the COMPAT_BSDSIGNALS flag. |
| COMPAT_BSDCHOWN | If selected, chown(2) is restricted to processes with superuser privileges. However, a process with effective user ID equal to the user ID of the file, but without superuser privileges, can change the group ID of the file to the effective group ID of the process or to one of the process's supplementary group IDs. |
| COMPAT_BSDNOTRUNC | If selected, pathname components longer than NAME_MAX generate an error. |

getcompat returns the current process compatibility flags. By default, compatibility flags are preserved across forks and are reset by execs (see COMPAT_EXEC above).

The default process compatibility flags are COMPAT_BSDPROT and COMPAT_BSDNBIO.

**RETURN VALUE**

Upon successful completion, setcompat returns the previous compatibility mask and getcompat returns the current compatibility mask. Otherwise, a value of −1 is returned and errno is set to indicate the error.

**ERRORS**

`setcompat` will return the following error code.

`[EINVAL]` *flag* results in a change in the state of the `COMPAT_BSDSIGNALS` bit and a signal is currently pending, caught, or held.

`[EINVAL]` *flag* is either `COMPAT_BSDTTY` or `COMPAT_SYSCALLS` and the `COMPAT_BSDSIGNALS` are also not set.

**SEE ALSO**

`exec`(2), `fork`(2), `sigvec`(2), `set42sig`(3), `signal`(3), `setuid`(3), `termio`(7).

## NAME

setgroups — set group access list

## SYNOPSIS

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

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

## DESCRIPTION

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

Only the superuser may set new groups.

## RETURN VALUE

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

## ERRORS

The setgroups call will fail if

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

## SEE ALSO

getgroups(2), initgroups(3X).

## NAME

setpgid — set process group ID for job control

## SYNOPSIS

```
int setpgid(pid, pgid)
pid_t pid, pgid;
```

## DESCRIPTION

setpgid is used to join an existing process group or to create a new process group within the session of the calling process. The process group ID of a session group leader cannot be changed. The process group ID of the process specified by *pid* is set to *pgid*. If *pid* or *pgid* is 0, the process ID of the calling process is used.

## RETURN VALUE

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

## ERRORS

If any of the following conditions occur, setpgid returns –1 and sets errno to the corresponding value:

| | |
|---|---|
| [EACCESS] | *pid* matches the process ID of a child process of the calling process and the child has successfully executed one of the exec functions. |
| [EINVAL] | The value of *pgid* is less than 0 or exceeds {PID_MAX}. |
| [EPERM] | The process indicated by *pid* is a session group leader.<br><br>The value of *pid* is valid but matches the process ID of a child of the calling process and the child process is not in the same session as the calling process. The value of *pgid* does not match the process ID of the process indicated by *pid* and there is no process with a process group ID that matches the value of *pgid* in the same session as the calling process. |
| [ESRCH] | *pid* does not match the process ID of the calling process or of a child process of the calling process. |

**SEE ALSO**

exec(2), getpgrp(2), setsid(2P), tcsetpgrp(3P).

## NAME

setpgrp — set process group ID

## SYNOPSIS

```
int setpgrp()

int setpgrp(pid,pgrp)
int pid, pgrp;
```

## DESCRIPTION

The first form of 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.

The second form of setpgrp is available when the process has requested 4.2 BSD compatibility. setpgrp will then set the process group of the specified process *pid* to the specified *pgrp*. If *pid* is zero, then the call applies to the current process.

If the user is not superuser, then the affected process must have the same effective user ID as the invoking user or be a descendant of the invoking process.

## RETURN VALUE

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

## ERRORS

The setpgrp call fails if:

[ESRCH] the process is not found.

[EPERM] The caller is not superuser.

## SEE ALSO

exec(2), fork(2), getpid(2), intro(2), kill(2), setcompat(2), signal(3).

## NAME

setregid — set real and effective group ID

## SYNOPSIS

```
int setregid(rgid, egid)
int rgid, egid;
```

## DESCRIPTION

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

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

## RETURN VALUE

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

## ERRORS

[EPERM] The current process is not the superuser and a change other than changing the effective group ID to the real group ID was specified.

## NOTES

This call only works in COMPAT_BSDPROT compatibility mode.

## SEE ALSO

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

## NAME

setreuid — set real and effective user ID

## SYNOPSIS

```
int setreuid(ruid, euid)
int ruid, euid;
```

## DESCRIPTION

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

## RETURN VALUE

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

## ERRORS

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

## NOTES

This call only works in COMPAT_BSDPROT compatibility mode.

## SEE ALSO

getuid(2), setcompat(2), setregid(2), setuid(2).

## NAME

setsid — create session and set process group ID

## SYNOPSIS

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

pid_t setsid()
```

## DESCRIPTION

setsid creates a new session if the calling process is not a process group leader. The calling process becomes the session leader of this new session, the process group leader of a new process group, and does not have a controlling terminal. The process group ID of the calling process is set to the process ID of the calling process.

## RETURN VALUE

On successful completion, setsid returns the value of the process group ID of the calling process.

## ERRORS

If any of the following conditions occur, setsid returns −1 and sets *errno* to the corresponding value:

| | |
|---|---|
| [EPERM] | The calling process is already a process group leader or the process group ID of a process other than the calling process matches the process ID of the calling process. |

## SEE ALSO

exec(2), _exit(2), fork(2), getpid(2), kill(2), setpgid(2P), sigaction(3P).

## NAME

setuid, setgid — set user and group ID

## SYNOPSIS

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

uid_t setuid(uid)
int uid;

int setgid(gid)
gid_t gid;
```

## DESCRIPTION

setuid sets the real user ID, effective user ID, and saved set-user-ID of the calling process. If the effective user ID of the calling process is superuser, the real user ID, effective user ID, and saved set-user-ID are set to *uid*. If the effective user ID of the calling process is not the superuser, 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 the superuser, but the saved set-user-ID from exec(2) is equal to *uid*, the effective user ID is set to *uid*.

setgid sets the real group ID, effective group ID, and saved set-group-ID of the calling process.

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

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

## RETURN VALUE

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

## ERRORS

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

[EPERM] The real user or group ID of the calling process is not equal to *uid* or *gid* and its effective user

ID is not the superuser.

[EINVAL] *uid* is out of range.

**SEE ALSO**

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

**NAME**

shmctl — shared memory control operations

**SYNOPSIS**

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

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

**DESCRIPTION**

shmctl provides a variety of shared memory control operations as specified by *cmd*. (Structure definitions and permissions are described in intro(2).) The following *cmd*s are available:

IPC_STAT — Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*.

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 superuser or to the value of shm_perm.uid in the data structure associated with *shmid*.

IPC_RMID — Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of shm_perm.uid in the data structure associated with *shmid*. The identifier and its associated data structure are not actually removed until there are no more referencing processes. See ipcrm(1), and ipcs(1).

**RETURN VALUE**

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

**ERRORS**

`shmctl` will fail if one or more of the following is true.

| | |
|---|---|
| `[EINVAL]` | *shmid* is not a valid shared memory identifier. |
| `[EINVAL]` | *cmd* is not a valid command. |
| `[EACCES]` | *cmd* is equal to `IPC_STAT` and READ operation permission is denied to the calling process (see `intro`(2)). |
| `[EAGAIN]` | The system has temporarily exhausted its available memory or swap space. |
| `[EPERM]` | *cmd* is equal to `IPC_RMID` or `IPC_SET` and the effective user ID of the calling process is not equal to that of superuser 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. |

**SEE ALSO**

`intro`(2), `shmget`(2), `shmop`(2).

## NAME

shmget — get shared memory segment

## SYNOPSIS

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

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

## DESCRIPTION

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

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

*key* is equal to IPC_PRIVATE.

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

*Note:* A shared memory segment of *size* is always rounded up to the nearest whole page.

The key IPC_PRIVATE will create an identifier and associated data structure that is unique to the calling process and its children.

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 equal to 0.

shm_ctime is set equal to the current time.

## RETURN VALUE

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.

**ERRORS**

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(2)) as specified by the low-order 9 bits of *shmflg* would not be granted.

[EAGAIN] The system has temporarily exhausted its available memory or swap space.

[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) && (*shmflg* & IPC_EXCL)) is "true".

**SEE ALSO**

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

## NAME

shmop, shmat, shmdt — shared memory operations

## SYNOPSIS

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

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

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

## DESCRIPTION

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

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

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

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

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

## RETURN VALUES

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, a value of −1 is returned and errno is set to indicate the error.

**ERRORS**

`shmat` will fail and not attach the shared memory segment if one or more of the following is true.

`[EINVAL]` *shmid* is not a valid shared memory identifier.

`[EACCES]` Operation permission is denied to the calling process (see `intro`(2)).

`[EAGAIN]` The system has temporarily exhausted its available memory or swap space.

`[ENOMEM]` The available data space is not large enough to accommodate the shared memory segment.

`[EINVAL]` *shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus `SHMLBA`)) is an illegal address.

`[EINVAL]` *shmaddr* is not equal to zero, (*shmflg* & `SHM_RND`) is "false", and the value of *shmaddr* is an illegal address.

`[EMFILE]` The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

`[EINVAL]` `shmdt` 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.

**SEE ALSO**

`exec`(2), `exit`(2), `fork`(2), `intro`(2), `shmctl`(2), `shmget`(2).

## NAME

shutdown — shut down part of a full-duplex connection

## SYNOPSIS

```
int shutdown(s, how)
int s, how;
```

## DESCRIPTION

The shutdown call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

## RETURN VALUE

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

## ERRORS

The call succeeds unless:

| | |
|---|---|
| [EBADF] | *s* is not a valid descriptor. |
| [ENOTSOCK] | *s* is a file, not a socket. |
| [ENOTCONN] | The specified socket is not connected. |

## SEE ALSO

connect(2N), socket(2N).

## NAME

sigblock, sigmask — block signals

## SYNOPSIS

```
#include <signal.h>

int sigblock(mask);
int mask;

sigmask(signum)
int signum;
```

## DESCRIPTION

sigblock causes the signals specified in *mask* to be added to the set of signals currently being blocked from delivery. Signals are blocked if the corresponding bit in *mask* is a 1; the macro sigmask is provided to construct the mask for a given *signum*.

It is not possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

## RETURN VALUE

The previous set of masked signals is returned.

## SEE ALSO

kill(2), sigvec(2), sigsetmask(2), signal(3).

## NAME

sigpause — release blocked signals and wait for interrupt

## SYNOPSIS

```
int sigpause (mask)
int mask;
```

## DESCRIPTION

sigpause assigns *mask* to the set of blocked signals and then waits for a signal to arrive; on return the set of masked signals is restored. *mask* is usually 0 to indicate that no signals are now to be blocked.

In normal usage, a signal is blocked using sigblock(2). To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses, awaiting work, by using sigpause with the mask returned by sigblock.

## RETURN VALUE

sigpause always terminates by being interrupted, returning −1.

## ERRORS

sigpause always terminates by being interrupted with errno set to EINTR.

## SEE ALSO

sigblock(2), sigvec(2), signal(3).

NAME

sigpending — examine pending signals

SYNOPSIS

```
#include <signal.h>

int sigpending(set)
sigset_t *set;
```

DESCRIPTION

sigpending stores the set of signals that are blocked from delivery and are pending for the calling process at the location referenced by *set*.

RETURN VALUE

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

ERRORS

If *set* points to an invalid address, sigpending will return −1 and set errno to EFAULT.

SEE ALSO

sigsetops(3P), sigprocmask(3P).

## NAME

sigsetmask — set current signal mask

## SYNOPSIS

```
#include <signal.h>

int sigsetmask(mask);
int mask;

sigmask(signum)
int signum;
```

## DESCRIPTION

sigsetmask sets the current signal mask (those signals that are blocked from delivery). Signals are blocked if the corresponding bit in *mask* is a 1; the macro sigmask is provided to construct the mask for a given *signum*.

The system quietly disallows SIGKILL, SIGSTOP, or SIGCONT to be blocked.

## RETURN VALUE

The previous set of masked signals is returned.

## SEE ALSO

kill(2), sigvec(2), sigblock(2), sigpause(2), signal(3).

## NAME

sigstack — set or get signal stack context

## SYNOPSIS

```
#include <signal.h>

struct sigstack {
    caddr_t ss_sp;
    int ss_onstack;
};

int sigstack(ss, oss);
struct sigstack *ss, *oss;
```

## DESCRIPTION

sigstack allows users to define an alternate stack on which signals are to be processed. If *ss* is nonzero, it specifies a signal stack on which to deliver signals and tells the system if the process is currently executing on that stack. When a signal's action indicates its handler should execute on the signal stack (specified with a sigvec(2) call), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution. If *oss* is nonzero, the current signal stack state is returned.

## NOTES

Signal stacks are not "grown" automatically, as is done for the normal stack. If the stack overflows, unpredictable results may occur.

## RETURN VALUE

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

## ERRORS

sigstack will fail and the signal stack context will remain unchanged if one of the following occurs.

[EFAULT] Either *ss* or *oss* points to memory that is not a valid part of the process address space.

## SEE ALSO

sigvec(2), setjmp(3), signal(3).

## NAME

sigvec — optional BSD-compatible software signal facilities

## SYNOPSIS

```
#include <signal.h>

struct sigvec {
    int (*sv_handler)();
    int sv_mask;
    int sv_flags;
};

int sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;
```

## DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new process context is built. A process may specify a *handler* to which a signal is delivered or specify that a signal is to be *blocked* or *ignored.* A process may also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process. This can be changed on a per-handler basis, so that signals are taken on a special "signal stack."

All signals have the same priority. Signal routines execute with the signal that caused their invocation to be *blocked*, but other signals may yet occur. A global "signal mask" defines the set of signals currently blocked from being delivered to a process. The signal mask for a process is initialized from the signal mask of its parent (normally 0). It may be changed with a sigblock(2) or sigsetmask(2) call, or by a signal being delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process, then it is delivered. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described later), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally, the process will resume execution in the same context as before the signal's delivery. If the process wishes to resume in a different context,

then it must arrange to restore the previous context.

When a signal is delivered to a process, a new signal mask is installed for the same duration as the process's signal handler (or until a `sigblock` or `sigsetmask` call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and ORing in the signal mask associated with the handler to be invoked.

`sigvec` assigns a handler for a specific signal. If *vec* is nonzero, it specifies a handler routine and mask to be used when delivering the specified signal. If the `SV_ONSTACK` bit is set in `sv_flags`, the system will deliver the signal to the process on a signal stack, specified by `sigstack`(2). If the `SV_INTERRUPT` bit is set in `sv_flags`, system calls interrupted by a signal will not be restarted. If the `SV_NOCLDSTOP` bit is set in `sv_flags`, `SIGCHLD` will not be generated if a child process stops. If *ovec* is nonzero, the previous handling information for the signal is returned to the user.

The following is a list of the A/UX signals with the corresponding names of the include file <`signal.h`>.

| | | |
|---|---|---|
| `SIGHUP` | 1 | hangup |
| `SIGINT` | 2 | interrupt |
| `SIGQUIT` | 3* | quit |
| `SIGILL` | 4* | illegal instruction |
| `SIGTRAP` | 5* | trace trap |
| `SIGIOT` | 6* | IOT instruction |
| `SIGEMT` | 7* | EMT instruction |
| `SIGFPE` | 8* | floating point exception |
| `SIGKILL` | 9 | kill (cannot be caught, blocked, or ignored) |
| `SIGBUS` | 10* | bus error |
| `SIGSEGV` | 11* | segmentation violation |
| `SIGSYS` | 12* | bad argument to system call |
| `SIGPIPE` | 13 | write on a pipe with no one to read it |
| `SIGALRM` | 14 | alarm clock |
| `SIGTERM` | 15 | software termination signal |
| `SIGUSR1` | 16 | user defined signal 1 |
| `SIGUSR2` | 17 | user defined signal 2 |
| `SIGCLD` | 18• | child status has changed |
| `SIGPWR` | 19 | power-fail restart |
| `SIGTSTP` | 20† | stop signal generated from keyboard |
| `SIGTTIN` | 21† | background read attempted from control terminal |
| `SIGTTOU` | 22† | background write attempted to control terminal |
| `SIGSTOP` | 23† | stop (cannot be caught, blocked, or ignored) |
| `SIGXCPU` | 24 | cpu time limit exceeded |

| | | |
|---|---|---|
| SIGXFSZ | 25 | file size limit exceeded |
| SIGVTALRM | 26 | virtual time alarm (see setitimer(2)) |
| SIGPROF | 27 | profiling timer alarm (see setitimer(2)) |
| SIGWINCH | 28• | window size change |
| SIGCONT | 29• | continue after stop (cannot be blocked) |
| SIGURG | 30• | urgent condition present on socket |
| SIGIO | 31• | I/O is possible on a descriptor (see fcntl(2)) |

The signals marked with an asterisk (*) in the list above cause a core image to be dumped if not caught or ignored.

Once a signal handler is installed, it remains installed until another sigvec call is made or an execve(2) is performed. The default action for a signal may be reinstated by setting sv_handler to SIG_DFL; this default is termination (with a core image for starred signals) except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop if the process is part of a 4.2 job control group. They are ignored when using 5.2 signals. If sv_handler is SIG_IGN, the signal is subsequently ignored, and pending instances of the signal are discarded.

If a caught signal occurs during certain system calls, the call is normally restarted. The affected system calls are read(2) or write(2) on a slow device (such as a terminal, but not a file) and during a wait(2). This behavior may be modified by options supplied to the setcompat(2) system call.

After a fork(2), the child inherits all signals, the signal mask, and the signal stack.

execve(2) resets all caught signals to the default action and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; the signal handler reverts to the 5.2 signal mechanism.

## NOTES

The mask specified in *vec* is not allowed to block SIGKILL, SIGSTOP, or SIGCONT. This is done silently by the system.

## RETURN VALUE

A 0 value indicates that the call succeeded. A −1 return value indicates that an error occurred and errno is set to indicate the reason.

## ERRORS

`sigvec` will fail and no new signal handler will be installed if one of the following occurs.

| | |
|---|---|
| [EFAULT] | Either *vec* or *ovec* points to memory that is not a valid part of the process address space. |
| [EINVAL] | *sig* is not a valid signal number. |
| [EINVAL] | An attempt is made to ignore or to supply a handler for `SIGKILL` or `SIGSTOP`. |
| [EINVAL] | An attempt is made to ignore `SIGCONT` (by default `SIGCONT` is ignored). |

## SEE ALSO

`kill`(1), `ptrace`(2), `kill`(2), `sigblock`(2), `setcompat`(2), `sigsetmask`(2), `sigpause`(2), `sigstack`(2), `set42sig`(3), `signal`(3), `termio`(7).

## NAME

socket — create an endpoint for communication

## SYNOPSIS

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

int socket(af, type, protocol)
int af, type, protocol;
```

## DESCRIPTION

socket creates an endpoint for communication and returns a descriptor.

The *af* parameter specifies an address format with which addresses specified in later operations using the socket should be interpreted. These formats are defined in the include file <sys/socket.h>. The currently understood formats are:

| | |
|---|---|
| AF_UNIX | (UNIX path names) |
| AF_INET | (ARPA Internet addresses) |
| AF_PUP | (Xerox PUP-I Internet addresses) |
| AF_IMPLINK | (IMP "host at IMP" addresses) |

*Note:* The only address format currently supported on this implementation is AF_INET.

The socket has the indicated *type* which specifies the semantics of communication. Currently defined types are:

```
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM
```

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams with an out-of-band data transmission mechanism. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network interfaces. The types SOCK_RAW, which is available only to the superuser, and SOCK_SEQPACKET and SOCK_RDM, which are planned, but not yet implemented, are not described here.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type using a given address format. However, it is possible that many protocols may exist in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see services(4N) and protocols(4N).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect(2N) call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(2N) and recv(2N) calls. When a session has been completed a close(2) may be performed. Out-of-band data may also be transmitted as described in send(2N) and received as described in recv(2N).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with −1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send(2N) calls. It is also possible to receive datagrams at such a socket with recv(2N).

An fcntl(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives.

The operation of sockets is controlled by socket level *options*. These options are defined in the file <sys/socket.h> and explained below. setsockopt and getsockopt(2N) are used to set and get options, respectively.

| | |
|---|---|
| SO_DEBUG | turn on recording of debugging information |
| SO_REUSEADDR | allow local address reuse |
| SO_KEEPALIVE | keep connections alive |
| SO_DONTROUTE | do no apply routing on outgoing messages |
| SO_LINGER | linger on close if data present |
| SO_DONTLINGER | do not linger on close |

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a bind(2N) call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address. SO_LINGER and SO_DONTLINGER control the actions taken when unsent messages are queued on socket and a close(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the setsockopt call when SO_LINGER is requested). If SO_DONTLINGER is specified and a close is issued, the system will process the close in a manner which allows the process to continue as quickly as possible.

**RETURN VALUE**

A −1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

**ERRORS**

The socket call fails if:

| | |
|---|---|
| [EAFNOSUPPORT] | The specified address family is not supported in this version of the system. |

| | |
|---|---|
| [ESOCKTNOSUPPORT] | The specified socket type is not supported in this address family. |
| [EPROTONOSUPPORT] | The specified protocol is not supported. |
| [EMFILE] | The per-process descriptor table is full. |
| [ENOBUFS] | No buffer space is available. The socket cannot be created. |

SEE ALSO

accept(2N), bind(2N), connect(2N), getsockname(2N), getsockopt(2N), ioctl(2), listen(2N), recv(2N), select(2N), send(2N), shutdown(2N).

BUGS

The use of keepalives is a questionable feature for this layer.

## NAME

stat, fstat, lstat — get file status

## SYNOPSIS

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

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

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

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

## DESCRIPTION

stat obtains information about the named file. *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.

lstat is like stat except in the case where the named file is a symbolic link, in which case lstat returns information about the link, while stat returns information about the file the link references.

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 is placed concerning the file.

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

| | |
|---|---|
| ushort st_mode; | File mode; see stat(5) |
| ino_t st_ino; | Inode number |
| dev_t st_dev; | ID of device containing a directory entry for this file |
| dev_t st_rdev; | ID of device. This entry is defined only for character special or block special files |

| | |
|---|---|
| `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 when file data was last accessed (times measured in seconds since 00:00:00 GMT, Jan. 1, 1970). Changed by the following system calls: `creat`(2), `mknod`(2), `pipe`(2), `utime`(2), and `read`(2). |
| `time_t st_mtime;` | Time when data was last modified (times measured in seconds since 00:00:00 GMT, Jan. 1, 1970). Changed by the following system calls: `creat`(2), `mknod`(2), `pipe`(2), `utime`(2), and `write`(2). |
| `time_t st_ctime;` | Time wehn file status last changed (times measured in seconds since 00:00:00 GMT, Jan. 1, 1970) Changed by the following system calls: `chmod`(2), `chown`(2), `creat`(2), `link`(2), `mknod`(2), `pipe`(2), `unlink`(2), `utime`(2), and `write`(2). |
| `long st_blksize;` | optimal blocksize for I/O ops |
| `long st_blocks;` | actual number of blocks allocated |

RETURN VALUE

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

ERRORS

`stat` and `lstat` will fail if one or more of the following are true:

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

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

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

| | |
|---|---|
| [EBADF] | *fildes* is not a valid open file descriptor. |
| [EFAULT] | *buf* points to an invalid address. |

**SEE ALSO**

chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), readlink(2), statfs(2), time(2), unlink(2), ustat(2), utime(2), write(2), stat(5).

## NAME

statfs — get file-system statistics

## SYNOPSIS

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

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

int fstatfs(fildes, buf)
int fildes;
struct statfs *buf;
```

## DESCRIPTION

statfs returns information about a mounted file system. Replace *path* with the pathname of any file within the mounted file system and replace *buf* with a pointer to a statfs structure defined as follows:

```
typedef long fsid_t[2];

struct statfs {
   long  f_type;       /* type of info, zero
                          for now */
   long  f_bsize;      /* fundamental file system
                          block size */
   long  f_blocks;     /* total blocks in file
                          system */
   long  f_bfree;      /* free blocks */
   long  f_bavail;     /* free blocks available to
                          nonsuperuser */
   long  f_files;      /* total file nodes in
                          file system */
   long  f_ffree;      /* free file nodes in fs */
   fsid_t  f_fsid;     /* file system ID */
   long  f_spare[7];   /* spare for later */
};
```

Fields that are undefined for a particular file system are set to –1. fstatfs returns the same information about an open file referenced by the descriptor *fildes*.

**RETURN VALUE**

On successful completion, a value of 0 is returned. Otherwise, −1 is returned and the global variable `errno` is set to indicate the error.

**SEE ALSO**

`stat`(2), `ustat`(2).

## NAME

stime — set time

## SYNOPSIS

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

## DESCRIPTION

stime sets the the time and date. *tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

## RETURN VALUE

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

## ERRORS

stime will fail if:

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

## SEE ALSO

date(1), gettimeofday(2), settimeofday(2), time(2).

## NAME

symlink — make symbolic link to a file

## SYNOPSIS

```
int symlink(name1, name2)
char *name1, *name2;
```

## DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name may be an arbitrary path name; the files need not be on the same file system.

## RETURN VALUE

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in errno and a −1 value is returned.

## ERRORS

The symbolic link is made unless on or more of the following are true:

| | |
|---|---|
| [EPERM] | Either *name1* or *name2* contains a character with the high-order bit set. |
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOENT] | One of the pathnames specified was too long. |
| [ENOTDIR] | A component of the *name2* prefix is not a directory. |
| [EEXIST] | *name2* already exists. |
| [EACCES] | A component of the *name2* path prefix denies search permission. |
| [EROFS] | The file *name2* would reside on a read-only file system. |

| | |
|---|---|
| [EFAULT] | *name1* or *name2* points outside the process's allocated address space. |

**SEE ALSO**

ln(1), link(2), readlink(2), unlink(2).

## NAME

sync — update superblock

## SYNOPSIS

```
void sync()
```

## DESCRIPTION

The sync system call causes all information in memory that should be on disk to be written out. This includes modified superblocks, modified inodes, and delayed block I/O.

It should be used by programs which examine a file system, for example fsck, df, etc. It is mandatory before a reboot or a system shutdown.

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

## SEE ALSO

sync(1), fsync(2).

## NAME

time — get time

## SYNOPSIS

```
#include <time.h>

time_t time((long*)0)

time_t time(tloc)
time_t *tloc;
```

## DESCRIPTION

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

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

## RETURN VALUE

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

## ERRORS

time will fail if the following is true

[EFAULT] *tloc* points to an illegal address.

## SEE ALSO

date(1), gettimeofday(2), stime(2), ctime(3).

## NAME

times — get process and child process times

## SYNOPSIS

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

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

## DESCRIPTION

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

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

This information comes from the calling process and each of its terminated child processes for which it has executed a wait. All times are in 60ths of a second.

| | |
|---|---|
| tms_utime | CPU time used while executing instructions in the user space of the calling process. |
| tms_stime | CPU time used by the system on behalf of the calling process. |
| tms_cutime | sum of the tms_utimes and tms_cutimes of the child processes. |
| tms_cstime | sum of the tms_stimes and tms_cstimes of the child processes. |

## RETURN VALUE

Upon successful completion, times returns the elapsed real time, in 60ths of a second, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of times to another. If times fails, a −1 is returned and errno is set to indicate the error.

## ERRORS

times will fail if

[EFAULT] *buffer* points to an illegal address.

**SEE ALSO**

exec(2), fork(2), time(2), wait(2).

## NAME

truncate, ftruncate — truncate a file to a specified length

## SYNOPSIS

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

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

## DESCRIPTION

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

## RETURN VALUES

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

## ERRORS

truncate will fail if:

| | |
|---|---|
| [EPERM] | The pathname contains a character with the high-order bit set. |
| [ENOENT] | The pathname was too long. |
| [ENOTDIR] | A component of the path prefix of *path* is not a directory. |
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOENT] | The named file does not exist. |
| [EACCES] | A component of the *path* prefix denies search permission. |
| [EISDIR] | The named file is a directory. |
| [EROFS] | The named file resides on a read-only file system. |

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

*Note:* If you are running an NFS system and you are accessing a shared binary remotely, it is possible that you will not get this errno.

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

ftruncate will fail if:

[EBADF] The *fd* is not a valid descriptor.

[EINVAL] The *fd* references a socket, not a file.

## SEE ALSO

open(2).

## BUGS

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

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

## NAME

ulimit — get and set user limits

## SYNOPSIS

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

## DESCRIPTION

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

1 Get the 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 superuser may increase the limit.

3 Get the maximum possible break value. See brk(2).

## RETURN VALUE

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

## ERRORS

ulimit will fail and the limit will be unchanged if the following is true:

[EPERM] a process with an effective user ID other than superuser attempts to increase its file size limit.

## SEE ALSO

brk(2), write(2).

## NAME

umask — set and get file creation mask

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(cmask)
mode_t cmask;
```

## DESCRIPTION

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

The file-mode-creation mask is used whenever a file is created by creat(2), mknod(2) or open(2). The actual mode (see chmod(2)) of the newly created file is the difference between the given mode and *cmask*. In other words, *cmask* shows the bits to be turned off when a new file is created.

For the POSIX environment, the following constants for *cmask* are defined in <sys/stat.h>:

| | |
|---|---|
| S_IRUSR | read permission, owner |
| S_IWUSR | write permission, owner |
| S_IXUSR | execute/search permission, owner |
| S_IRGRP | read permission, group |
| S_IWGRP | write permission, group |
| S_IXGRP | execute/search permission, group |
| S_IRUSR | read permission, others |
| S_IWUSR | write permission, others |
| S_IXUSR | execute/search permission, others |

The previous value of *cmask* is returned by the call. The value is initially 022, which is an octal "mask" number representing the complement of the desired mode. The value 022 here means that no permissions are withheld from the owner, but write permission is forbidden to the group and to others. Its complement, the mode of the file, would be 0755. The file-mode-creation mask is inherited by child processes.

**RETURN VALUE**

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

**SEE ALSO**

csh(1), ksh(1), chmod(1), mkdir(1), sh(1), chmodl(2), creat(2), mknod(2), open(2).

## NAME

umount — unmount a file system

## SYNOPSIS

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

## DESCRIPTION

umount is used to unmount System V file systems only. unmount is used to unmount all others (see unmount(2)).

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

umount may be invoked only by the superuser.

## RETURN VALUE

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

## ERRORS

umount will fail if one or more of the following is true.

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

## SEE ALSO

unmount(2), mount(3).

## NAME

uname — get name of current system

## SYNOPSIS

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

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

## DESCRIPTION

uname stores information identifying the current system in the structure referenced by *name*.

uname uses the structure defined in <sys/utsname.h>:

```
struct      utsname {
     char   sysname[9];
     char   nodename[9];
     char   release[9];
     char   version[9];
     char   machine[9];
};
extern struct utsname utsname;
```

uname returns a null-terminated character string naming the current system in the character array sysname. Similarly, nodename contains the name by which the system is known on a communications network. release and version further identify the operating system. machine contains a standard name that identifies the hardware that the system is running on.

## RETURN VALUE

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

## ERRORS

uname will fail if the following is true:

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

## SEE ALSO

uname(1).

## NAME

unlink — remove directory entry

## SYNOPSIS

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

## DESCRIPTION

unlink removes the directory entry named by the path name referenced by *path.*

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

## RETURN VALUE

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

## ERRORS

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. |
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |
| [ENOENT] | The named file does not exist. |
| [EACCES] | Search permission is denied for a component of the path prefix. |
| [EACCES] | Write permission is denied on the directory containing the link to be removed. |
| [EISDIR] | The named file is a directory. |
| [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.

*Note:* If you are running an NFS system and you are accessing a shared binary remotely, it is possible that you will not get this errno.

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

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

**SEE ALSO**

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

## NAME

unmount — remove a file system

## SYNOPSIS

```
unmount (name)
char *name;
```

## DESCRIPTION

unmount is used to unmount all non-System V file systems. umount is used to unmount System V file systems only (see umount(2)).

unmount announces to the system that the directory *name* is no longer to refer to the root of a mounted file system. The directory *name* reverts to its ordinary interpretation.

## RETURN VALUE

unmount returns 0 if the action occurred; −1 if if the directory is inaccessible or does not have a mounted file system, or if there are active files in the mounted file system.

## ERRORS

unmount may fail with one of the following errors:

| | |
|---|---|
| [EINVAL] | The caller is not the superuser. |
| [EINVAL] | *name* is not the root of a mounted file system. |
| [EBUSY] | A process is holding a reference to a file located on the file system. |

## SEE ALSO

fsmount(2), mount(3), umount(2).

## BUGS

The error codes are in a state of disarray; too many errors appear to the caller as one value.

## NAME

ustat — get file system statistics

## SYNOPSIS

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

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

## DESCRIPTION

ustat returns information about a mounted file system. *dev* is a device number identifying a device containing a mounted file system. *buf* is a pointer to a ustat structure that includes to following elements:

```
daddr_t f_tfree;        /* Total free blocks */
ino_t   f_tinode;       /* Number of free inodes */
char    f_fname[6];     /* Filsys name */
char    f_fpack[6];     /* Filsys pack name */
```

## RETURN VALUE

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

## ERRORS

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

[EINVAL] *dev* is not the device number of a device containing a mounted file system.

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

## SEE ALSO

stat(2), statfs(2), fs(4).

## NAME

utime — set file access and modification times

## SYNOPSIS

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

## DESCRIPTION

utime sets the access and modification times of the named file. The pointer *path* points to a pathname for naming a file.

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

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

The times in the following structure, defined in <utime.h> are measured in seconds since 00:00:00 GMT, January 1, 1970.

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

## RETURN VALUE

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

## ERRORS

utime will fail if one or more of the following is true.

| | |
|---|---|
| [ENOENT] | The named file does not exist. |
| [EPERM] | A pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX. |
| [ELOOP] | Too many symbolic links were encountered in translating a pathname. |

| | |
|---|---|
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [EACCES] | Search permission is denied by a component of the path prefix. |
| [EPERM] | The effective user ID is not superuser and not the owner of the file and `times` is not NULL. |
| [EACCES] | The effective user ID is not superuser and not the owner of the file, `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. |

**SEE ALSO**

`stat`(2).

## NAME

uvar — return system-specific configuration information

## SYNOPSIS

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

int uvar(v)
struct var *v;
```

## DESCRIPTION

uvar returns system-specific configuration information contained in the kernel. The information returned contains table sizes, mask words, and other system-specific information for programs such as ps(1).

Presently, a maximum of 512 bytes of information is returned. The structure variable *v* points to the var structure.

```
struct var {
        int     v_buf;          /* Number of system buffers */
        int     v_call;         /* Maximum number of
                                   simultaneous callouts */
        int     v_inode;        /* Maximum number of incore
                                   inodes */
        char*   ve_inode;       /* Pointer to last incore
                                   inode table  */
        int     v_file;         /* Maximum number of open
                                   files */
        char*   ve_file;        /* Pointer to last open
                                   file table */
        int     v_mount;        /* Maximum number of file
                                   systems mountable */
        char*   ve_mount;       /* Pointer to last mounted
                                   file system table */
        int     v_proc;         /* Maximum number of
                                   processes */
        char*   ve_proc;        /* Pointer to last process
                                   table  */
        int     v_text;         /* Maximum number of shared
                                   text segments */
        char*   ve_text;        /* Pointer to last shared
                                   text segment table  */
        int     v_clist;        /* Maximum number of clists */
        int     v_sabuf;        /* Maximum number of system
                                   activity buffers */
        int     v_maxup;        /* Maximum number of user
                                   processes */
        int     v_cmap;         /* Size of core memory
                                   allocation map */
        int     v_smap;         /* Size of swap memory
                                   allocation map */
```

```
int   v_hbuf;        /* Maximum number of buffer
                        headers */
int   v_hmask;       /* Maximum number of buffer
                        headers - 1 */
int   v_flock;       /* Maximum number of file locks */
int   v_phys;        /* Maximum number of simultaneous
                        phys calls */
int   v_clsize;      /* Click size */
int   v_txtrnd;      /* Number of clicks per segment */
int   v_bsize;       /* Block size */
int   v_cxmap;       /* Context map size */
int   v_clktick      /* Clock tick */
int   v_hz;          /* Hz */
int   v_usize;       /* Size of user structure */
int   v_pageshift;   /* Page shift */
int   v_pagemask;    /* Page mask */
int   v_segshift;    /* Segment shift */
int   v_segmask;     /* Segment mask */
int   v_ustart;      /* Starting virtual address for
                        user program */
int   v_uend;        /* Ending virtual address for
                        user program */
char* ve_call;       /* Pointer to last callout table */
int   v_stkgap;      /* Obsolete */
int   v_cputype;     /* CPU type (1=68000) */
int   v_cpuver;      /* CPU version ID
                        (1=68000, 2=68010, 3=68020) */
int   v_mmutype;     /* MMU type
                        (1=none, 2=SUN, 3=68451) */
int   v_doffset;     /* Data offset */
int   v_kvoffset;    /* Kernel virtual offset */
int   v_svtext;      /* Maximum number of text
                        loitering segments */
char* ve_svtext;     /* Pointer to last text
                        loitering segment
                        in table  */
int   v_pbuf;        /* Maximum number of buffers
                        for physio */
int   v_nscatload;   /* Maximum number of entries
                        in scatter map */
int   v_udot;        /* Address of user structure */
int   v_region;      /* Number of memory regions */
int   v_sptmap;      /* Size of system virtual space */
int   v_vhndfrac;    /* Fraction of MAXMEM to set a
                        limit for running vehand  */
int   v_maxpmem;     /* Maximum physical memory to use */
int   v_nmbufs;      /* Buffers for networking */
int   v_npty;        /* Number of pseudo tty's */
int   v_maxcore;     /* Space used by kernel's heap
                        (.../GEN/sys/heap_kmem.c) */
int   v_maxheader;   /* Headers used by kernel's heap
                        (.../GEN/sys/heap_kmem.c) */
```

```
        int   v_nstream;     /* Number of stream heads */
        int   v_nqueue;      /* Number of stream queues */
        int   v_nblk4096;    /* Number of of 4K stream blocks */
        int   v_nblk2048;    /* Number of of 2K stream blocks */
        int   v_nblk1024;    /* Number of 1K stream blocks */
        int   v_nblk512;     /* Number of 512K stream blocks */
        int   v_nblk256;     /* Number of 256K stream blocks */
        int   v_nblk64;      /* Number of 64K stream blocks */
        int   v_nblk16;      /* Number of 16 byte stream blocks */
        int   v_nblk4;       /* Number of 4 byte stream blocks */
        char  *ve_proctab    /* &proc[0] */
        int   v_slice        /* a process's time slice */
        int   v_sbufsz       /* system buffer's sizes */
        int   v_fill[128-67]/* sized to make var 512 bytes */
};
```

## RETURN VALUE

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

## ERRORS

`uvar` will fail if

`[EFAULT]` *v* points to an illegal address.

## SEE ALSO

`ps`(1).

## NAME

wait — wait for child process to stop or terminate

## SYNOPSIS

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

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

## DESCRIPTION

wait 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 wait system call will return prematurely if a signal is received and if a child process stopped or terminated prior to the call on wait, return is immediate.

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

If the child process stopped, the high order 8 bits of *status* will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to 0177.

If the child process terminated due to an exit call, the low order 8 bits of *status* will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to exit; see exit(2).

If the child process terminated due to a signal, the high order 8 bits of *status* will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit 200) is set, a "core image" will have been produced; see signal(3).

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

## RETURN VALUE

If `wait` returns due to the receipt of a signal, a value of −1 is returned to the calling process and `errno` is set to `EINTR`. If `wait` returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of −1 is returned and `errno` is set to indicate the error.

## ERRORS

`wait` will fail and return immediately if one or more of the following are true:

`[ECHILD]` The calling process has no existing unwaited-for child processes.

## SEE ALSO

`exec(2)`, `exit(2)`, `fork(2)`, `intro(2)`, `pause(2)`, `ptrace(2)`, `wait3(2N)`, `signal(3)`.

## WARNINGS

See WARNINGS in `signal(3)`.

## NAME

wait3 — wait for child process to stop or terminate

## SYNOPSIS

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

int wait3(status, options, 0)
union wait *status;
int options;
```

## DESCRIPTION

wait3 provides an interface for programs which must not block when collecting the status of child processes. The *status* parameter is defined as above. The *options* parameter is used to indicate the call should not block if there are no processes which wish to report status (WNOHANG), and/or that children of the current process that are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal should also have their status reported (WUNTRACED).

When the WNOHANG option is specified and no processes wish to report status, wait3 returns a *pid* of 0. The WNOHANG and WUNTRACED options may be combined by ORing the two values.

The declaration of "union wait" is found in <sys/wait.h>. The third argument, 0, is a placeholder. The "normal case" is the same as wait(2).

## RETURN VALUE

wait3 returns −1 if there are no children not previously waited for; 0 is returned if WNOHANG is specified and there are no stopped or exited children.

## SEE ALSO

exit(2), wait(2).

## NAME

write, writev — write on a file

## SYNOPSIS

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

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

int writev(fildes, iov, ioveclen)
int fildes;
struct iovec *iov;
int ioveclen;
```

## DESCRIPTION

write attempts to write *nbytes* bytes from the buffer pointed to by *buf* to the file associated with *fildes*. writev performs the same action, but gathers the output data from the *iovlen* buffers specified by the members of the iovec array: iov[0], iov[1], and so on.

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

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

On devices incapable of seeking, writing always starts 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.

When writing to a pipe (or FIFO), write requests of PIPE_BUF bytes or less are not interleaved with data from other processes writing to the same pipe. Writes of greater than PIPE_BUF bytes may have data interleaved, on arbitrary boundaries, with writes by other processes.

RETURN VALUE

On successful completion, the number of bytes actually written is returned. If the process compatibility flag COMPAT_SYSCALL is set (see setcompat(2)), as in the POSIX environment, and write-interrupted by a signal after successfully writing some data, it returns the number of bytes written. If *nybytes* is 0, write returns 0 and have no other result. Otherwise, −1 is returned and errno is set to indicate the error.

ERRORS

When attempting to write to a stream when buffer space is not currently available and if O_NDELAY or O_NONBLOCK is set, the write returns the number of bytes written before there were no buffers available. If O_NDELAY and O_NONBLOCK are clear, the write blocks until buffers become available.

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

[EIO] A physical I/O error has occurred or the process is in a background process group and is attempting to write to its controlling terminal (see termio(7P)). TOSTOP is set, the process is not blocking or ignoring SIGTTOU. The process group of the processes is orphaned or the write was to an open device after a modem disconnect or hangup occurred.

[ENXIO] The device associated with the file descriptor is a block device file or character device file and the value of the file pointer is out of range.

[EBADF] The file descriptor, *fildes*, is not valid and 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(2).

[EFAULT]
Part of *iov* or data to be written to the file points outside the allocated address space of the process.

[EFAULT]
*buf* points outside the allocated address space of the

process.

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

[ENOSPC]
Not enough space was left on the device containing the file.

If the number of bytes specified in a write request exceeds the available space limit, that is, the per-process file size (see ulimit(2)), or exceeds the size of the physical media, only as many bytes for which there is room will be written. For example, suppose there is space in a file for an additional 20 bytes before reaching a limit. A write of 512 bytes returns 20. The next write of a nonzero number of bytes gives a failure return with the following exceptions:

If the file being written is a pipe (or FIFO) and

if the O_NDELAY flag of the file-flag word is set, then writing to a full pipe (or FIFO) returns a count of 0.

if O_NDELAY is clear, writes to a full pipe (or FIFO) blocks until space becomes available.

If the file being written is a pipe (or FIFO) and

if O_NONBLOCK is set, write requests for PIPE_BUF or fewer bytes either succeed completely or return −1 and set errno to EAGAIN.

if O_NONBLOCK is clear, a write request may block until space is available.

When writing to a file descriptor (other than a pipe or FIFO) that supports nonblocking writes and cannot accept data immediately and if O_NONBLOCK is set, write writes what data it can, returning −1 and setting errno to EAGAIN. If O_NONBLOCK is clear, write blocks until the data is accepted.

SEE ALSO

creat(2), dup(2), fcntl(2), lseek(2), open(2), pipe(2), select(2N), socket(2N), ulimit(2).

# Table of Contents

## Section 3: Subroutines (A-L)

## NAME

intro — introduction to subroutines and libraries

## SYNOPSIS

```
#include <stdio.h>

FILE *stdin, *stdout, *stderr

#include <math.h>
```

## DESCRIPTION

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

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library `libc`, which is automatically loaded by the C compiler `cc`(1). The link editor `ld`(1) searches this library under the `-lc` flag option. Some functions require declarations that can be included in the program being compiled by adding the line

`#include <`*header-filename*`>`

The appropriate header file is indicated in the synopsis of a function description.

(3F) These functions constitute the Fortran intrinsic function library `libF77` and are automatically available to the Fortran programmer. They require no special invocation of the compiler. These functions are flagged with the (3F) suffix on the associated manual page entries and appear in their own alphabetically organized subsection at the end of this section.

(3M) These functions constitute the Math Library `libm`. They are automatically loaded as needed by the Fortran compiler `f77`(1). They are not automatically loaded by the C compiler `cc`(1); however, the link editor searches this library under the `-lm` flag option. Declarations for these functions may be obtained from the header file `<math.h>`.

(3N) These functions are networking routines and, unless otherwise noted, are found in the Standard C Library `libc.a`.

(3P) These functions provide POSIX functionality and are found in `libposix.a`. The POSIX environment is described in the *A/UX Guide to POSIX* and *A/UX Programming Languages and Tools, Volume 1*.

(3X) These functions pertain to various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

(3S) These functions constitute the standard I/0 package. An introduction to this package follows under the heading "STANDARD I/O." The functions are in the Standard C Library `libc`. Declarations should be obtained from the include file `<stdio.h>`.

DEFINITIONS

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

Many groups of Fortran intrinsic functions have "generic" function names that do not require explicit or implicit type declarations. The type of the function is determined by the type of its argument(s). For example, the generic function `max` returns an integer value if given integer arguments (`max0`), a real value if given real arguments (`amax1`), or a double-precision value if given double-precision arguments (`dmax1`).

STANDARD I/O

The functions described in the entries of subclass (3S) in this manual provide an efficient, user-level I/O buffering scheme. The functions are in the library `libc` and declarations should be obtained from the header file `<stdio.h>`.

The I/O function may be grouped into the following categories: file access, file status, input, output, and miscellaneous. For lists of the functions in each category, refer to the library sections of *A/UX Programming Languages and Tools, Volume 1*. The inline macros getc(3S) and putc(3S) 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 getc and putc; these macros and routinges can be freely intermixed.

A file with associated buffering is called a stream and is declared to be a pointer to a defined type FILE. fopen(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

| | |
|---|---|
| stdin | standard input file |
| stdout | standard output file |
| stderr | standard error file |

*Note:* Invalid stream pointers cause serious errors, including possible program termination. Individual function descriptions describe the possible error conditions.

A constant NULL (0) designates a nonexistent pointer.

An integer constant EOF (–1) is returned upon an 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.

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

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

The functions and constants mentioned in the (3S) entries are declared in the header file <stdio.h> and need no further declaration. The constants and the following functions are implemented as macros: getc, getchar, putc, putchar, feof, ferror, clearerr, and fileno. Redeclaration of these names is perilous.

For descriptions and examples of header files, refer to "The Standard C Library (libc)," "The C Math Library," and "The C Object Library," in *A/UX Programming Languages and Tools, Volume 1*.

RPC SERVICE LIBRARY

These functions are part of the Remote Process Control (RPC) service library librpcsvc. To have the link editor load this library, use the -lrpcsvc option of cc. Declarations for these functions may be obtained from various include files within <rpcsvc/*.h>.

RETURN VALUE

Functions in the C and Math Libraries (3C and 3M) may return the conventional values 0 or ±HUGE (the largest-magnitude, double-precision floating-point numbers; HUGE is defined in the <math.h> header file) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable errno (see intro(2)) is set to the value EDOM or ERANGE. Because many of the Fortran intrinsic functions use the routines found in the Math Library, the same conventions apply.

FILES

/lib/libc.a
/usr/lib/libF77.a
/lib/libm.a

SEE ALSO

ar(1), cc(1), f77(1), ld(1), lint(1), nm(1), open(2), close(2), lseek(2), pipe(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S), math(5).

*A/UX Programming Languages and Tools, Volume 1*.

WARNINGS

Many of the functions in the libraries call or refer to other functions and external variables described in this section and in Section 2 (System Calls). If a program inadvertently defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded. The lint(1) program checker reports name conflicts of this kind

as "multiple declarations" of the names in question. Definitions for sections 2, 3C, and 3S are checked automatically. Other definitions can be included by using the `-l` option. (For example, `-lm` includes definitions for `libm`, the Math Library, section 3M). The use of `lint` is highly recommended.

## NAME

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

## SYNOPSIS

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

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

## DESCRIPTION

These functions are used to maintain numbers stored in base-64 ASCII characters. This is a notation by which long integers can be represented by up to 6 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 6 characters, uses the first 6.

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

## BUGS

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

## NAME

abort — generate an IOT fault

## SYNOPSIS

```
int abort()
```

## DESCRIPTION

abort 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 control if SIGIOT is caught or ignored, in which case the value returned is that of the kill(2) 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

sdb(1), exit(2), kill(2), signal(3).

**NAME**

abort — terminate Fortran program

**SYNOPSIS**

```
call abort()
```

**DESCRIPTION**

abort terminates the program which calls it, closing all open files truncated to the current position of the file pointer.

**DIAGNOSTICS**

When invoked, abort prints

```
Fortran abort routine called
```

on the standard error output.

**SEE ALSO**

abort(3C).

**NAME**

abs — return integer absolute value

**SYNOPSIS**

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

**DESCRIPTION**

abs returns the absolute value of its integer operand.

**BUGS**

In two's-complement representation, the absolute value of the negative integer with largest magnitude is returned.

Some implementations trap this error, but others simply ignore it.

**SEE ALSO**

floor(3M).

## NAME

abs, iabs, dabs, cabs, zabs — Fortran absolute value

## SYNOPSIS

```
integer i1, i2
real r1, r2
double precision dp1, dp2
complex cx1, cx2
double complex dx1, dx2

r2=abs(r1)

i2=iabs(i1)
i2=abs(i1)

dp2=dabs(dp1)
dp2=abs(dp1)

cx2=cabs(cx1)
cx2=abs(cx1)

dx2=zabs(dx1)
dx2=abs(dx1)
```

## DESCRIPTION

abs is the family of absolute value functions. iabs returns the integer absolute value of its integer argument. dabs returns the double-precision absolute value of its double-precision argument. cabs returns the complex absolute value of its complex argument. zabs returns the double-complex absolute value of its double-complex argument. The generic form abs returns the type of its argument.

## SEE ALSO

floor(3M).

## NAME

acos, dacos — Fortran arccosine intrinsic function

## SYNOPSIS

```
real r1, r2
double precision dp1, dp2

r2=acos(r1)

dp2=dacos(dp1)
dp2=acos(dp1)
```

## DESCRIPTION

acos returns the real arccosine of its real argument. dacos returns the double-precision arccosine of its double-precision argument. The generic form acos may be used with impunity because its argument determines the type of the returned value.

## SEE ALSO

trig(3M).

## NAME

aimag, dimag — Fortran imaginary part of complex argument

## SYNOPSIS

```
real r
complex cxr
double precision dp
double complex cxd

r=aimag(cxr)

dp=dimag(cxd)
```

## DESCRIPTION

aimag returns the imaginary part of its single-precision complex argument. dimag returns the double-precision imaginary part of its double-complex argument.

## NAME

aint, dint — Fortran integer part intrinsic function

## SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2=aint(r1)
dp2=dint(dp1)
dp2=aint(dp1)
```

## DESCRIPTION

aint returns the truncated value of its real argument in a real. dint returns the truncated value of its double-precision argument as a double-precision value. aint may be used as a generic function name, returning either a real or double-precision value depending on the type of its argument.

## NAME

asin, dasin — Fortran arcsine intrinsic function

## SYNOPSIS

```
real r1, r2
double precision dp1, dp2

r2=asin(r1)

dp2=dasin(dp1)
dp2=asin(dp1)
```

## DESCRIPTION

asin returns the real arcsine of its real argument. dasin returns the double-precision arcsine of its double-precision argument. The generic form asin may be used with impunity as it derives its type from that of its argument.

## SEE ALSO

trig(3M).

## NAME

assert — verify program assertion

## SYNOPSIS

```
#include <assert.h>

assert (expression)
int expression;
```

## DESCRIPTION

This macro is useful for putting diagnostics into programs. If *expression* is false (zero) when `assert` is executed, `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(1)) or with the preprocessor control statement `#define NDEBUG` ahead of the `#include <assert.h>` statement, stops assertions from being compiled into the program.

## NOTES

`assert` cannot be used in an expression since it turns into an `if` statement.

## SEE ALSO

cpp(1), abort(3C).

## NAME

atan, datan — Fortran arctangent intrinsic function

## SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2=atan(r1)
dp2=datan(dp1)
dp2=atan(dp1)
```

## DESCRIPTION

atan returns the real arctangent of its real argument. datan returns the double-precision arctangent of its double-precision argument. The generic form atan may be used with a double-precision argument returning a double-precision value.

## SEE ALSO

trig(3M).

## NAME

atan2, datan2 — Fortran arctangent intrinsic function

## SYNOPSIS

```
real r1, r2, r3
double precision dp1, dp2, dp3

r3=atan2(r1, r2)

dp3=datan2(dp1, dp2)
dp3=atan2(dp1, dp2)
```

## DESCRIPTION

atan2 returns the arctangent of *arg1*/*arg2* as a real value. datan2 returns the double-precision arctangent of its double-precision arguments. The generic form atan2 may be used with impunity with double-precision arguments.

## SEE ALSO

trig(3M).

## NAME

atof — convert ASCII string to floating-point number

## SYNOPSIS

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

## DESCRIPTION

atof converts a character string pointed to by *nptr* to a double-precision floating point number. The first unrecognized character ends the conversion. atof recognizes an optional string of white space characters (blanks or tabs), 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.

```
atof(str)
```

is equivalent to

```
strtod(str, (char **)NULL)
```

## ERRORS

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

## SEE ALSO

scanf(3S), strtod(3C), strtol(3C).

**NAME**

atp_open, atp_close, atp_sendreq, atp_getreq, atp_sendrsp, atp_getresp, atp_asynch_kind — AppleTalk Transaction Protocol (ATP) interface

**SYNOPSIS**

```
#include <at/appletalk.h>
#include <at/atp.h>
cc [flags] files -lat [libraries]

int atp_open(socket)
at_socket *socket;

int atp_close(fd)
int fd;

int atp_sendreq(fd, dest, buf, len, userdata, xo,
xo_relt, tid, resp, retry, asynch);
int fd;
at_inet_t *dest;
char *buf;
int len, userdata, xo, xo_relt;
u_short *tid;
at_resp_t *resp;
at_retry_t *retry;
int asynch;

int atp_getreq(fd, src, buf, len, userdata, xo, tid,
bitmap, asynch);
int fd;
at_inet_t *src;
char *buf;
int *len, *userdata, *xo;
u_short *tid;
u_char bitmap;
int asynch;

int atp_sendrsp(fd, dest, xo, tid, resp);
int fd;
at_inet_t *dest;
int xo;
u_short tid;
at_resp_t *resp;
```

```
int atp_getresp(fd, tid, resp);
int fd;
u_short tid;
at_resp_t *resp;

int atp_asynch_kind(fd, tid);
int fd;
u_short *tid;
```

DESCRIPTION

The ATP interface provides applications with access to the services of the AppleTalk Transaction Protocol, a transport layer protocol.

These routines use the following structures, defined in <at/appletalk.h>.

The at_inet_t structure specifies the AppleTalk internet address of a Datagram Delivery Protocol (DDP) AppleTalk socket endpoint:

```
typedef struct  at_inet {
        u_short      net;
        at_node      node;
        at_socket    socket;
} at_inet_t;
```

The at_retry_t structure specifies the retry interval and maximum count for a transaction:

```
typedef struct  at_retry {
        short      interval;
        short      retries;
        u_char     backoff;
} at_retry_t;
```

The members of this structure are

*interval* The interval in seconds before ATP retries a request.

*retries* The maximum number of retries for this ATP request. If *retries* is AT_INF_RETRY, the request will be repeated infinitely.

*backoff* The value by which the interval is multiplied on each retry, to a maximum interval of 16 seconds.

The at_resp_t, defined in <at/atp.h>, specifies buffers to be used for response data:

```
typedef struct  at_resp {
```

```
        u_char          bitmap;
        struct iovec    resp[ATP_TRESP_MAX];
        int             userdata[ATP_TRESP_MAX];
} at_resp_t;
```

The members of this structure are

*bitmap* The bitmap of responses expected and for which buffers are allocated.

*resp* An `iovec` structure describing the response buffers and their lengths.

*userdata* An array of 32-bit words holding the user bytes for each ATP response.

The `atp_open` routine opens an ATP AppleTalk socket and returns a file descriptor for use with the remaining ATP calls.

*socket* A pointer to the DDP socket number to open. If the AppleTalk socket number is 0, or if *socket* is NULL, a DDP socket is dynamically assigned. If non-NULL, the socket number is returned in *socket*.

The `atp_close` routine closes the ATP AppleTalk socket identified by the file descriptor *fd*.

The `atp_sendreq` routine sends an ATP request to another socket. This call blocks until a response is received. The parameters are

*fd* The ATP file descriptor to use in sending the request.

*dest* The AppleTalk internet address of the AppleTalk socket to which the request should be sent.

*buf* The request data buffer.

*len* The size of request data buffer size.

*userdata* The user bytes for the ATP request header.

*xo* Is true (nonzero) if the request is to be an exactly-once (XO) transaction.

*xo_relt* Is ignored if *xo* is not set to true. Otherwise, it may be used to set the release timer value on the remote end. The default value is ATP_XO_DEF_REL_TIME. The other permissible values are: ATP_XO_30SEC, ATP_XO_1MIN, ATP_XO_2MIN, ATP_XO_4MIN, and ATP_XO_8MIN. These are declared in the file

<at/atp.h>. All other values are illegal.

*tid* On return, the transaction identifier for this transaction. Can be NULL if the caller is not interested in the transaction identifier.

The atp_sendreq routine requires a pointer to an at_resp_t structure containing two arrays for the response data: *resp*, an eight-entry iovec array, and *userdata*, an eight-entry array. The field *iov_base* in each iovec entry points to a buffer that will contain response data. The field *iov_len* specifies the length of the buffer. The field *bitmap* indicates the responses expected; on return it indicates the responses received.

On return each *iov_len* entry indicates the length of the actual response data. If the number of responses is lower than expected, either an end-of-message was received or the retry count was exceeded. In the latter case, an error is returned. Each *userdata* entry in *resp* contains the user data for the respective ATP response packet. The *retry* pointer specifies the ATP request retry timeout in seconds and the maximum retry count. If *retry* is NULL, the default timeout, ATP_DEF_INTERVAL; the default retries, ATP_DEF_RETRIES; and a *backoff* value of 1 are used. The *retries* parameter of *retry* can be set to AT_INF_RETRY, in which case the transaction will be repeated infinitely.

The atp_getreq routine receives an incoming ATP request. It is completed when a request is received. The parameters are

*fd* The ATP file descriptor to use in receiving the request.

*src* The AppleTalk internet address of the AppleTalk socket from which the request was sent.

*buf* The data buffer in which to store the incoming request.

*len* The data buffer size.

*userdata* On return the user bytes from the ATP request header. Can be NULL if the caller is not interested in the *userdata*.

*xo* Is true (nonzero) if the request is an exactly once (XO) transaction.

*tid* The transaction identifier for this transaction.

*bitmap* The responses expected by the requester.

Because the transaction may require a response, the *xo*, *tid*, and *bitmap* parameters are always returned.

The `atp_sendrsp` routine sends an ATP response to another AppleTalk socket. All response data is passed in one `atp_sendrsp` call. In the case of an XO transaction, the call does not return until a release is received from the requester or the release timer expires. In the latter case an error is returned. The parameters are

*fd* The ATP file descriptor to use in sending the response.

*dest* The AppleTalk internet address of the AppleTalk socket to which the response should be sent.

*tid* The transaction identifier for this transaction.

*xo* Is true (nonzero) if the response is an exactly once (XO) transaction.

The `atp_sendrsp` routine requires a pointer to an `at_resp_t` structure containing two arrays for the response data: *resp*, an eight-entry `iovec` array, and *userdata*, an eight-entry array. The field *iov_base* in each `iovec` entry points to a buffer containing response data. The field *iov_len* specifies the length of the response data. Each *userdata* entry in *resp* contains the user data to be sent with the respective ATP response packet. The *bitmap* field indicates the responses to be sent.

**ERRORS**

All routines return −1 on error with a detailed error code in `errno`. For additional errors returned by the underlying DDP module, see `ddp`(3N). The error messages are

`[EAGAIN]` The request failed due to a temporary resource limitation; try again. When this error occurs, no XO transaction is initiated.

`[EINVAL]` The *dest*, *len*, *resp*, or *retry* parameter was invalid.

`[ENOENT]` The request was an attempt to send a response to a nonexistent transaction.

`[ETIMEDOUT]` The request exceeded the maximum retry count.

[EMSGSIZE] The response was larger than the buffer, or more responses were received than expected; truncated to available buffer space (atp_sendreq).
The request buffer is too small for request data; truncated (atp_getreq).
The response is too large; maximum is ATP_DATA_SIZE bytes (atp_sendrsp).

**WARNINGS**

The parameter *asynch* and the routines atp_getresp and atp_asynch_kind allow asynchronous sending and receiving of ATP requests. Asynchronous requests are not currently supported, so set *asynch* to 0 to indicate synchronous operation.

The length of each response buffer, specified in *iov_len*, is overwritten by the actual response length when atp_sendreq returns.

**SEE ALSO**

ddp(3N), nbp(3N), pap(3N), rtmp(3N); *Inside AppleTalk;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming*.

## NAME

j0, j1, jn, y0, y1, yn — Bessel functions

## SYNOPSIS

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

double jn(n, x)
int n;
double x;

double y0(x)
double x;

double y1(x)
double x;

double yn(n, x)
int n;
double x;
```

## DESCRIPTION

j0 and j1 return Bessel functions of *x* of the first kind of orders 0 and 1 respectively. jn returns the Bessel function of *x* of the first kind of order *n*.

y0 and y1 return the Bessel functions of *x* of the second kind of orders 0 and 1 respectively. yn returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

## ERRORS

Nonpositive 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 set errno to ERANGE. In addition, a message indicating TLOSS error is printed on the standard error output.

## NOTES

These error-handling procedures may be changed with the function matherr(3M).

**SEE ALSO**

matherr(3M).

## NAME

blt, blt512 — block transfer data

## SYNOPSIS

```
int blt(to, from, count)
char *to;
char *from;
int count;

int blt512(to, from, count)
char *to;
char *from;
int count;
```

## DESCRIPTION

blt does a fast copy of *count* bytes of data starting at address *from* to address *to*.

blt512 does a fast copy of *count* number of consecutive 512 byte units starting at address *from* to address *to*.

## SEE ALSO

memory(3).

## NAME

and, or, xor, not, lshift, rshift — Fortran bitwise boolean functions

## SYNOPSIS

```
integer i, j, k
real a, b, c
double precision dp1, dp2, dp3

k=and(i, j)
c=or(a, b)
j=xor(i, a)
j=not(i)
k=lshift(i, j)
k=rshift(i, j)
```

## DESCRIPTION

The generic intrinsic boolean functions `and`, `or`, and `xor` return the value of the binary operations on their arguments. `not` is a unary operator returning the one's complement of its argument. `lshift` and `rshift` return the value of the first argument shifted left or right, respectively, the number of times specified by the second (integer) argument.

The boolean functions are generic, i.e., defined for all data types as arguments and return values. Where required, the compiler generates appropriate type conversions.

## NOTES

Although defined for all data types, use of boolean functions on non-integer data is not productive.

## BUGS

The implementation of the shift functions may cause large shift values to deliver unexpected results.

## NAME

bsearch — binary search a sorted table

## SYNOPSIS

```
#include <search.h>

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

## DESCRIPTION

bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer to a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. The pointer *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; *width* is the width of an element in bytes (the `sizeof (*key)` should be used for *width*); *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 0, depending on whether or not the first argument is to be considered less than, equal to, or greater than the second.

## EXAMPLES

The example following searches a table that contains pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

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

#define   TABSIZE   1000

struct node {  /* these are stored in the table */
   char *string;
   int length;
};
struct node table[TABSIZE];  /* table to be searched */
   .
   .
```

```
    .
{
    struct node /*node_ptr, node;
    int node_compare( );  /* routine to compare 2 nodes */
    char str_space[20];   /* space to read string into */
    .
    .
    .
    node.string = str_space;
    while (scanf("%s", node.string) != EOF) {
        node_ptr = (struct node *)bsearch((char *)(&node),
              (char *)table, TABSIZE,
              sizeof(struct node), node_compare);
        if (node_ptr != NULL) {
            (void)printf("string = %20s, length = %d\n",
                node_ptr->string, node_ptr->length);
        } else {
            (void)printf("not found: %s\n", node.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);
}
```

NOTES

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

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

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

RETURN VALUE

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

**SEE ALSO**

hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C).
Donald Knuth, *The Art of Computer Programming: Volume 3, Sorting and Searching.*

## NAME

bcopy, bcmp, bzero — bit and byte string operations

## SYNOPSIS

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

int bcopy(b1, b2, length)
char *b1, *b2;
int length;

int bcmp(b1, b2, length)
char *b1, *b2;
int length;

int bzero(b, length)
char *b;
int length;
```

## DESCRIPTION

The macro bcopy, and the functions bcmp and bzero operate on variable length strings of bytes. They do not check for null bytes as the routines in string(3C) do.

bcopy copies *length* bytes from string *b1* to the string *b2*.

bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, nonzero otherwise. Both strings are assumed to be *length* bytes long.

bzero places *length* 0 bytes in the string *b1*.

## FILES

/usr/include/sys/param.h

## BUGS

The bcmp and bcopy routines take parameters backwards from strcmp and strcpy.

## SEE ALSO

memory(3C), string(3).

## NAME

htonl, htons, ntohl, ntohs — convert values between host and network byte order

## SYNOPSIS

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

u_long htonl(hostlong);
u_long hostlong;

u_short htons(hostshort);
u_short hostshort;

u_long ntohl(netlong);
u_long netlong;

u_short ntohs(netshort);
u_short netshort;
```

## DESCRIPTION

These macros convert 16 and 32 bit quantities between network byte order and host byte order. On machines in the Motorola 68000-family such as the Macintosh II, these routines are defined as null macros in the include file <netinet/in.h>.

These routines are most often used in conjunction with Internet addresses and ports as returned by getservent(3N).

## SEE ALSO

getservent(3N).

## NAME

cfgetospeed, cfgetispeed, cfsetospeed, cfsetispeed — get or set the value of the output and input baud rate

## SYNOPSIS

```
#include <termios.h>

speed_t cfgetospeed(termio_p)
struct termios *termio_p;

speed_t cfgetispeed(termio_p)
struct termios *termio_p;

cfsetospeed(termio_p, speed)
struct termios *termio_p;
speed_t speed;

cfsetispeed(termio_p, speed)
struct termios *termio_p;
speed_t speed;
```

## DESCRIPTION

These routines are used for getting and setting the input and output baud rate. A/UX® does not support different values for the input and output baud rate; cfsetispeed and cfsetospeed change both the input and output baud rate.

cfgetospeed returns the output baud rate stored in the *c_cflag* that is referenced by *termio_p*.

cfgetispeed returns the input baud rate stored in the *c_cflag* that is referenced by *termio_p*.

The following baud- rate values are supported for the value of *speed*:

| | |
|---|---|
| B0 | Hang up |
| B50 | 50 baud |
| B75 | 75 baud |
| B110 | 110 baud |
| B134 | 134 baud |
| B150 | 150 baud |
| B200 | 200 baud |
| B300 | 300 baud |
| B600 | 600 baud |
| B1200 | 1200 baud |
| B1800 | 1800 baud |
| B2400 | 2400 baud |

| | |
|---|---|
| B4800 | 4800 baud |
| B9600 | 9600 baud |
| B19200 | 19200 baud |
| B38400 | 38400 baud |

cfsetospeed sets the baud rate stored in *c_cflag* that is referenced by *termio_p* to *speed*. B0 is used to terminate the connection. If B0 is specified, the modem control lines are no longer asserted.

cfsetispeed sets the baud rate stored in *c_cflag* that is referenced by *termio_p* to *speed*. If *speed* is 0, the baud-rate is not changed.

For any particular hardware, unsupported baud rate changes are ignored.

cfsetispeed and cfsetospeed only modify the termios structure. For the baud rate changes to take place, tcsetattr(3P) must be called with the modified structure as an argument.

## RETURN VALUE

cfgetispeed and cfgetospeed return the appropriate baud rate. On successful completion, cfsetispeed and cfsetospeed return 0. If an error is detected, cfsetospeed and cfsetispeed return −1 and set errno to indicate the error.

## ERRORS

If the following condition occurs, cfsetispeed and cfsetospeed return −1 and set errno to the corresponding value:

[EINVAL] *speed* specifies an invalid baud rate.

## SEE ALSO

termios(7P), tcgetattr(3P).

## NAME

charcvt — convert the character code to another encoding scheme

## SYNOPSIS

```
#include <intl.h>

charcvt(c, conv)
int c;
int conv;
```

## DESCRIPTION

charcvt returns a character code that has been converted from the given character code represented in *c*. The value of *conv* contains a value defining how *c* is to be converted from one character-set encoding scheme to another. The current values of *conv* are:

| | |
|---|---|
| ITOM | Convert from ISO encoding to Macintosh® encoding. |
| MTOI | Convert from Macintosh encoding to ISO encoding. |

## RETURN VALUE

charcvt returns -1 if there is no equivalent character in the other character set.

## SEE ALSO

mactoiso(1), maccodes(5), isocodes(5).

**NAME**

clock — report CPU time used

**SYNOPSIS**

```
long clock()
```

**DESCRIPTION**

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

**SEE ALSO**

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

**BUGS**

The value returned by clock is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned wraps around after accumulating only 2,147 seconds of CPU time (about 36 minutes).

## NAME

conjg, dconjg — Fortran complex conjugate intrinsic function

## SYNOPSIS

```
complex cx1, cx2
double complex dx1, dx2

cx2=conjg(cx1)

dx2=dconjg(dx1)
```

## DESCRIPTION

conjg returns the complex conjugate of its complex argument. dconjg returns the double-complex conjugate of its double-complex argument.

## NAME

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

## SYNOPSIS

```
#include <ctype.h>

int toupper(c)
int c;

int tolower(c)
int c;

int _toupper(c)
int c;

int _tolower(c)
int c;

int toascii(c)
int c;
```

## DESCRIPTION

toupper and tolower have as domain the range of getc(3S): the integers from −1 through 255. If the argument of toupper represents a lowercase letter, the result is the corresponding uppercase letter. If the argument of tolower represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

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

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

## SEE ALSO

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

## NAME

cos, dcos, ccos — Fortran cosine intrinsic function

## SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, fx2

r2=cos(r1)

dp2=dcos(dp1)
dp2=cos(dp1)

cx2=ccos(cx1)
cx2=cos(cx1)
```

## DESCRIPTION

cos returns the real cosine of its real argument. dcos returns the double-precision cosine of its double-precision argument. ccos returns the complex cosine of its complex argument. The generic form cos may be used with impunity because its returned type is determined by that of its argument.

## SEE ALSO

trig(3M).

## NAME

cosh, dcosh — Fortran hyperbolic cosine intrinsic function

## SYNOPSIS

```
real r1, r2
double precision dp1, dp2

r2=cosh(r1)

dp2 =dcosh(dp1)
dp2=cosh(dp1)
```

## DESCRIPTION

cosh returns the real hyperbolic cosine of its real argument. dcosh returns the double-precision hyperbolic cosine of its double-precision argument. The generic form cosh may be used to return the hyperbolic cosine in the type of its argument.

## SEE ALSO

sinh(3M).

## NAME

crypt, encrypt — generate DES encryption

## SYNOPSIS

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

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

## DESCRIPTION

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

*key* is a user's typed password. *salt* is a 2-character string chosen from the set [a-zA-Z0-9./]; this string is used to perturb the DES algorithm in one of 4,096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first 2 characters are the *salt* itself.

The encrypt entry provides (rather primitive) access to the actual DES algorithm.

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 changed in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm. If *edflag* is zero, the argument is encrypted; if nonzero, it is decrypted.

## SEE ALSO

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

## BUGS

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

## NAME

ctermid — generate filename for terminal

## SYNOPSIS

```
#include <stdio.h>

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

## DESCRIPTION

ctermid generates the pathname of the controlling terminal for the current process, and stores it in a string.

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

## NOTES

The difference between ctermid and ttyname(3C) is that ttyname must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while ctermid returns a string (/dev/tty) that refers to the terminal if used as a filename. For this reason, ttyname is useful only if the process already has at least one file open to a terminal.

## SEE ALSO

ttyname(3C).

## NAME

asctime, ctime, difftime, gmtime, localtime, mktime, tzset, tzsetwall — convert date and time to ASCII

## SYNOPSIS

```
extern char *tzname[2];

void tzset()

void tzsetwall()

#include <sys/types.h>

char *ctime(clock)
time_t *clock;

double difftime(time1, time0)
time_t time1;
time_t time0;

#include <time.h>

char *asctime(tm)
struct tm *tm;

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

time_t mktime(tm)
struct tm *tm;

extern long timezone;
extern int daylight;
```

## DESCRIPTION

`tzset` uses the value of the environment variable `TZ` to set time conversion information used by `localtime`. If `TZ` does not appear in the environment, the best available approximation to local wall-clock time, as specified by the file `localtime` in the system time-conversion information directory, is used by the `localtime` function. If `TZ` appears in the environment but its value is a null string, Coordinated Universal Time (UTC) is used (without leap-second correction). If `TZ` appears in the environment and its value is not a null string, `TZ` is used in one of the following ways:

If the value begins with a colon (:), it is used as a pathname of a file from which to read the time-conversion information.

If the value does not begin with a colon, it is first used as the pathname of a file from which to read the time conversion information, and if that file cannot be read, it is used directly as a specification of the time-conversion information.

If it begins with a slash (/), it is used as an absolute pathname when `TZ` is used as a pathname; otherwise, it is used as a pathname relative to a system time-conversion information directory. The file must be in the format specified in `tzfile`(5).

When `TZ` is used directly as a specification of the time-conversion information, it must have the following syntax (spaces are inserted for clarity):

*std offset*[*dst*[*offset*][,*rule*]]

The placeholders mean the following:

*std* and *dst* — Specify three or more bytes that are the designation for the standard (*std*) or summer (*dst*) time zone. Only *std* is required. If *dst* is missing, then summer time does not apply in this locale. Uppercase and lowercase letters are explicitly allowed. Any characters except a leading colon (:), digits, a comma (,), a minus sign (–), a plus sign (+), and ASCII NUL are allowed.

*offset* — Add the value one of offset to the local time to arrive at UTC. The format of *offset* is:

*hh*[:*mm*[:*ss*]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The value of *offset* following *std* is required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between 0 and 24, and the minutes and seconds, if present, between 0 and 59.

If preceded by a - (minus sign), the time zone is east of the prime meridian; otherwise, it is west, which may be indicated by an optional preceding + (plus sign).

*rule* Specify when to change to and back from summer time. The format of *rule* is:

*date*/*time*,*date*/*time*

where the first *date* describes when the change from standard to summer time occurs and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* is one of the following:

`J`*n* The Julian day *n* ($1 \le n \le 365$). Leap days are not counted so that in all years, including leap years, February 28 is day 59, and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

*n* The zero-based Julian day ($0 \le n \le 365$). Leap days are counted, and it is possible to refer to February 29.

`M`*m*`.`*n*`.`*d* The *d*'th day ($0 \le d \le 6$) of week *n* of month *m* of the year ($1 \le n \le 5$, $1 \le m \le 12$, where week 5 means "the last *d* day in month *m*" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day 0 is Sunday.

The *time* has the same format as *offset* except that no leading - or + (minus or plus sign) is allowed. The default, if *time* is not given, is `02:00:00`.

If no *rule* is present in TZ, the rules specified by the tzfile(5)-format file posixrules in the system time-conversion information directory are used, with the standard and summer time offsets from UTC replaced by those specified by the values of *offset* in TZ.

For compatibility with System V Release 3.1, a semicolon (;) may be used to separate *rule* from the rest of the specification. For compatibility with applications that expect the environment variable TZ to be in the format of System V Release 2, the value of TZ should *not* include *rule*.

If the TZ environment variable does not specify a tzfile(5)-format and cannot be interpreted as a direct specification, UTC is used with the standard time abbreviation set to the value of the TZ environment variable (or to the leading characters of its value if it is lengthy).

In most installations TZ is set by default, when the user logs on, to the value in the file /etc/TIMEZONE.

Tzsetwall arranges for the system's best approximation to local wall-clock time to be delivered by subsequent calls to localtime.

Ctime converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 UTC, January 1, 1970, and returns a pointer to a 26-character string of the form

```
Thu Nov 24 18:22:48 1986
```

All the fields have a constant width.

localtime and gmtime return pointers to tm structures, described below. localtime corrects for the time zone and any time-zone adjustments, such as daylight-saving time (DST) in the United States. Before doing so, localtime calls tzset, if tzset has not been called in the current process. After filling in the tm structure, localtime sets the tm_isdst'th element of tzname to a pointer to an ASCII string that's the time-zone abbreviation to be used with the return value of localtime.

Gmtime converts to UTC.

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

`mktime` converts the broken-down time, expressed as local time, in the structure pointed to by *tm* into a calendar time value with the same encoding as that of the values returned by the `time` function. The original values of the `tm_wday` and `tm_yday` components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges. (A positive or zero value for `tm_isdst` causes `mktime` to presume initially that summer time (for example, daylight-saving time in the United States) respectively, is or is not in effect for the specified time. A negative value for `tm_isdst` causes the `mktime` function to attempt to divine whether summer time is in effect for the specified time.) On successful completion, the values of the `tm_wday` and `tm_yday` components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to their normal ranges. The final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined. `Mktime` returns the specified calendar time; if the calendar time cannot be represented, it returns –1.

`Difftime` returns the difference between two calendar times, *time1* - *time0*, expressed in seconds.

Declarations of all the functions and externals, and the `tm` structure, are in the `<time.h>` header file. The structure (of type) `struct tm` includes the following fields:

```
int tm_sec;   /* seconds (0 - 60) */
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; /* is DST ("summer" time) in effect? */
```

`Tm_isdst` is nonzero if a timezone adjustment such as daylight-savings time or summer time is in effect.

The external variable `timezone` contains the difference, in seconds, between UTC and local standard time (in Pacific Standard Time, `timezone` is 5*60*60). The external variable `daylight` is nonzero if a timezone adjustment such as DST or summer time is in effect.

**FILES**

| | |
|---|---|
| `/etc/zoneinfo` | Time-zone information directory |
| `/etc/zoneinfo/localtime` | Local time zone file |
| `/etc/zoneinfo/posixrules` | Used with POSIX-style TZ's |
| `/etc/zoneinfo/GMT` | For UTC leap seconds |

If `/etc/zoneinfo/GMT` is absent, UTC leap seconds are loaded from `/etc/zoneinfo/posixrules`.

**SEE ALSO**

`tzfile`(4), `getenv`(3), `time`(2), `environ`(5)

**NOTES**

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

## NAME

`isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii` — classify characters

## SYNOPSIS

```
#include <ctype.h>

int isalpha(c)
int c;

...
```

## DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. `isascii` is defined on all integer values; the rest are defined only where `isascii` is true and on the single non-ASCII value EOF (–1); see `intro`(3)).

| | |
|---|---|
| `isalpha` | *c* is a letter. |
| `isupper` | *c* is an upper-case letter. |
| `islower` | *c* is a lower-case letter. |
| `isdigit` | *c* is a digit [0-9]. |
| `isxdigit` | *c* is a hexadecimal digit [0-9], [A-F] or [a-f]. |
| `isalnum` | *c* is an alphanumeric (letter or digit). |
| `isspace` | *c* is a space, tab, carriage return, *newline*, vertical tab, or form-feed. |
| `ispunct` | *c* is a punctuation character (neither control nor alphanumeric). |
| `isprint` | *c* is a printing character, code 040 (space) through 0176 (tilde). |
| `isgraph` | *c* is a printing character, similar to `isprint` except false for space. |
| `iscntrl` | *c* is a delete character (0177) or an ordinary control character (less than 040). |
| `isascii` | *c* is an ASCII character, code less than 0200. |

**RETURN VALUE**

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

**SEE ALSO**

intro(3), ascii(5).

## NAME

curses — CRT screen handling and optimization package

## SYNOPSIS

```
#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 terminal 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` as a C structure. These data structures are manipulated with functions described later, 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`.) Eventually `refresh()` must be called, telling the routines to make the users 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`.

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 the file `/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/paul/myterms`, `curses`

will first check /usr/paul/myterms/v/vt100, and if that fails, will then check /usr/lib/terminfo/v/vt100. This is useful for developing experimental definitions or when write permission in /usr/lib/terminfo is not available.

FUNCTIONS

Routines listed here may be called when using the full curses. Those marked with a plus (+) are macros. Those marked with an asterisk (*) may be called when using Mini-Curses.

addch (*ch*) *+
Add a character to stdscr (like putchar) (wraps to next line at end of line).

addstr (*str*) *+
Calls addch with each character in *str*.

attroff (*attrs*) *+
Turn off attributes named in *attrs*.

attron (*attrs*) *+
Turn on attributes named in *attrs*.

attrset (*attrs*) *+
Set current attributes to *attrs*.

baudrate () *
Current terminal speed.

beep () *
Sound beep on terminal.

box (*win, vert, hor*)
Draw a box around edges of *win*. *vert* and *hor* are characters to use for vertical and horizontal edges of box.

clear () +
Clear stdscr.

clearok (*win, bf*)
Clear screen before next redraw of *win*.

clrtobot () +
Clear to bottom of stdscr.

clrtoeol () +
Clear to end-of-line on stdscr.

cbreak () *
Set cbreak mode.

`delay_output(`*ms*`)*`
Insert *ms* millisecond pause in output.

`delch()+`
Delete a character.

`deleteln()+`
Delete a line.

`delwin(`*win*`)`
Delete *win*.

`doupdate()`
Update screen from all `wnooutrefresh`.

`echo()*`
Set echo mode.

`endwin()*`
End window modes.

`erase()+`
Erase `stdscr`.

`erasechar()`
Return user's erase character.

`fixterm()`
Restore tty to "in `curses`" state.

`flash()`
Flash screen or sound 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*) coordinates.

`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 character at current (*y,x*) coordinates.

`initscr()`*
Initialize screens.

`insch(`*c*`)`
Insert a character.

`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 remain at current position.

`longname()`
Return verbose name of terminal.

`meta(`*win, flag*`)` *
Allow metacharacters 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*`)`+
`move(`*y, x*`)`, then `addstr(`*str*`)`.

`mvcur(`*oldrow, oldcol, newrow, newcol*`)`
Low level cursor motion.

`mvdelch(`*y, x*`)`+
Like `delch`, but `move(`*y, x*`)` first.

mvgetch (*y*,*x*) +
Like getch, but move (*y*,*x*) first.

mvgetstr (*y*,*x*) +
Like getstr, but move (*y*,*x*) first.

mvinch (*y*,*x*) +
Like inch, but move (*y*,*x*) first.

mvinsch (*y*,*x*,*c*)
Like insch, but move (*y*,*x*) first.

mvprintw (*y*,*x*,*fmt*,*args*) +
Like printw, but move (*y*,*x*) first.

mvscanw (*y*,*x*,*fmt*,*args*)
Like scanw, but move (*y*,*x*) first.

mvwaddch (*win*,*y*,*x*,*ch*) +
Like addch, but move (*y*,*x*) first.

mvwaddstr (*win*,*y*,*x*,*str*) +
Like waddstr, but move (*y*,*x*) first.

mvwdelch (*win*,*y*,*x*) +
Like wdelch, but move (*y*,*x*) first.

mvwgetch (*win*,*y*,*x*) +
Like wgetch, but move (*y*,*x*) first.

mvwgetstr (*win*,*y*,*x*) +
Like wgetstr, but move (*y*,*x*).

mvwin (*win*, *by*, *bx*)
Like win, but move (*y*,*x*).

mvwinch (*win*,*y*,*x*) +
Like winch, but move (*y*,*x*).

mvwinsch (*win*,*y*,*x*,*c*) +
Like winsch, but move (*y*,*x*).

mvwprintw (*win*,*y*,*x*,*fmt*,*args*) +
Like wprintw, but move (*y*,*x*).

mvwscanw (*win*,*y*,*x*,*fmt*,*args*) +
Like wscanw, but move (*y*,*x*).

newpad (*nlines*,*ncols*)
Create a new pad with given dimensions.

`newterm` (*type*, *fd*)
Set up new terminal of a given type to output on *fd*.

`newin` (*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` is called.

`prefresh` (*pad*, *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*)
Refresh from pad, starting from given upper left corner of pad, with output to the given portion of screen.

`printw` (*fmt*, *arg1*, *arg2*, ...)
`printf` on `stdscr`.

`raw` ()*
Set raw mode.

`refresh` ()*+
Make current screen look like `stdscr`.

`resetterm` ()*
Set tty modes to "out of `curses`" state.

`resetty()*`
Reset tty flags to stored value.

`saveterm()*`
Save current modes as "in `curses`" state.

`savetty()*`
Store current tty flags.

`scanw` (*fmt, arg1, arg2, ...*)
`scanf` through `stdscr`.

`scroll` (*win*)
Scroll *win* one line.

`scrollok` (*win, flag*)
Allow terminal to scroll if *flag*!=0.

`set_term` (*new*)
Now talk to terminal *new*.

`setscrreg` (*t, b*)+
Set user scrolling region to lines *t* through *b*.

`setterm` (*type*)
Establish terminal with given type.

`standend()*+`
Clear standout mode attribute.

`standout()*+`
Set standout mode attribute.

`subwin` (*win, lines, cols, begin_y, begin_x*)
Create a subwindow.

`touchwindow` (*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 character 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.*

`wattrst` (*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 character from *win.*

`wdeleteln` (*win*)
Delete line from *win.*

`werase` (*win*)
Erase *win.*

`wgetch` (*win*)
Get a character through *win.*

`wgetstr` (*win, str*)
Get a string through *win.*

`winch` (*win*) +
Get character at current (*y,x*) in *win.*

`winsch` (*win, c*)
Insert character into *win.*

`winsertln` (*win*)
Insert line into *win.*

`wmove` (*win, y, x*)
Set current (*y,x*) coordinates on *win.*

`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*.

## THE `terminfo` LEVEL ROUTINES

These routines should be called by programs wishing to deal directly with the `terminfo` database; however, due to the low level of this interface, it is discouraged. Initially, `setupterm` should be called, which will define the set of terminal dependent variables defined in `terminfo`(4). The include files `<curses.h>` and `<term.h>` should be included to get the definitions for these strings, numbers, and flags. 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`. Before exiting, `resetterm` should be called to restore the tty modes. (Programs desiring shell escapes or suspention with CONTROL-Z can call `resetterm` before the shell is called and `fixterm` after returning from the shell.)

`fixterm()`
Restore tty modes for `terminfo` use (called by `setupterm`).

`resetterm()`
Reset tty modes to state before program entry.

`setupterm` (*term*, *fd*, *rc*)
Read in database. Terminal type is the character string *term* and all output is to file descriptor *fd*. A status value is returned in the integer pointed to by *rc*: 1 is normal. The sim-

plest call would be

```
setupterm(0, 1, 0)
```

which uses all defaults.

`tparm(`*str*`,`*p1*`,`*p2*`,...,`*p9*`)`
Instantiate string *str* with parameters *pi*.

`tputs(`*str*`,`*affcnt*`,`*putc*`)`
Apply padding info to string *str*. The argument *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 the terminal in video attribute mode *attrs*, which is any combination of the attributes listed later. Characters are passed to `putchar`-like function, `putc`.

`vidattr(`*attrs*`)`
Like `vidputs` but outputs through `putchar`.

## THE `termcap` COMPATIBILITY ROUTINES

These routines were included as a conversion aid for programs that use `termcap`. Their parameters are the same as for `termcap`, and they are emulated using the `terminfo` database. They may be dispensed with at a later date.

`tgetent(`*bp*`,`*name*`)`
Look up `termcap` entry for *name*.

`tgetflag(`*id*`)`
Get boolean entry for *id*.

`tgetnum(`*id*`)`
Get numeric entry for *id*.

`tgetstr(`*id*`,`*area*`)`
Get string entry for *id*.

`tgoto(`*cap*`,`*col*`,`*row*`)`
Apply parameters to given *cap*.

`tputs(`*cap*`,`*affcnt*`,`*fn*`)`
Apply padding to *cap* calling *fn* as `putchar`.

## ATTRIBUTES

The following video attributes can be passed to the functions `attron`, `attroff`, `attrset`.

| | |
|---|---|
| `A_STANDOUT` | Terminal's best highlighting mode |
| `A_UNDERLINE` | Underlining |
| `A_REVERSE` | Reverse video |
| `A_BLINK` | Blinking |
| `A_DIM` | Half bright |
| `A_BOLD` | Extra bright or bold |
| `A_BLANK` | Blanking (invisible) |
| `A_PROTECT` | Protected |
| `A_ALTCHARSET` | Alternate character set |

## FUNCTION KEYS

The following function keys might be returned by `getch` if `keypad` has been enabled. Note that not all of these are currently supported due to lack of definitions in `terminfo` or the terminal not transmitting a unique code when the key is pressed.

| NAME | VALUE | KEY NAME |
|---|---|---|
| `KEY_BREAK` | 0401 | Break key (unreliable) |
| `KEY_DOWN` | 0402 | Arrow key down |
| `KEY_UP` | 0403 | Arrow key up |
| `KEY_LEFT` | 0404 | Arrow key left |
| `KEY_RIGHT` | 0405 | Arrow key right |
| `KEY_HOME` | 0406 | Home key (upward+left arrow) |
| `KEY_BACKSPACE` | 0407 | Backspace (unreliable) |
| `KEY_F0` | 0410 | Function keys (space for 64 is reserved) |
| `KEY_F(`*n*`)` + | (KEY_F0+(*n*)) | Formula for *fn* |
| `KEY_DL` | 0510 | Delete line |
| `KEY_IL` | 0511 | Insert line |
| `KEY_DC` | 0512 | Delete character |
| `KEY_IC` | 0513 | Insert character or enter insert mode |
| `KEY_EIC` | 0514 | Exit insert character 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 backward (reverse) |
| `KEY_NPAGE` | 0522 | Next page |
| `KEY_PPAGE` | 0523 | Previous page |
| `KEY_STAB` | 0524 | Set tab |
| `KEY_CTAB` | 0525 | Clear tab |

| | | |
|---|---|---|
| KEY_CATAB | 0526 | Clear all tabs |
| KEY_ENTER | 0527 | Enter or send (unreliable) |
| KEY_SRESET | 0530 | Soft (partial) reset (unreliable) |
| KEY_RESET | 0531 | Reset or hard reset (unreliable) |
| KEY_PRINT | 0532 | Print or copy |
| KEY_LL | 0533 | Home down or bottom (lower left) |

**SEE ALSO**

curses5.0(3X), terminfo(4).

**WARNINGS**

The plotting library plot(3X) and the curses library curses(3X) both use the names erase() and move(). The curses versions are macros. If you need both libraries, put the plot(3X) code in a different source file than the curses(3X) code or #undef move() and erase() in the plot(3X)) code. Similarly, move is also a macro in <sys/pcl.h>.

## NAME

curses5.0 — BSD-style screen functions with optimal cursor motion

## SYNOPSIS

cc [*flags*] *files* -lcurses5.0 -ltermcap [*libraries*]

## DESCRIPTION

These routines are a subset of the routines provided in the new curses library and are provided for compatibility with programs that use the old curses and termcap libraries. These routines give the user a method for updating screens with reasonable optimization. They maintain an image of the current screen while the user sets up an image of a new one. Then the refresh() tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine initscr() must be called before any of the other routines that deal with windows and screens are used. The routine endwin() should be called before exiting.

## FUNCTIONS

Routines marked with a plus (+) are macros.

| | |
|---|---|
| addch(*ch*)+ | Add a character to stdscr. |
| addstr(*str*)+ | Add a string to stdscr. |
| box(*win*, *vert*, *hor*) | Draw a box around a window. |
| 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. |
| crmode() | Set cbreak mode. |
| delch()+ | Delete a character. |
| deleteln()+ | Delete a line. |
| delwin(*win*) | Delete *win*. |
| echo() | Set echo mode. |
| endwin() | End window modes. |
| erase()+ | Erase stdscr. |
| getcap(*name*) | Get terminal capability *name*. |
| getch()+ | Get a character through stdscr. |
| getstr(*str*)+ | Get a string through stdscr. |
| gettmode() | Get tty modes. |
| getyx(*win*, *y*, *x*)+ | Get (*y*,*x*) coordinates. |
| inch()+ | Get character at current (*y*,*x*) coordinates. |

| | |
|---|---|
| `initscr()` | Initialize screens. |
| `insch(`*c*`)+` | Insert a character. |
| `insertln()+` | Insert a line. |
| `leaveok(`*win*`,` *boolf*`)+` | Set leave flag for *win*. |
| `longname(`*termbuf*`,` *name*`)` | Get long name from *termbuf*. |
| `move(`*y*`,` *x*`)+` | Move to (*y*,*x*) on `stdscr`. |
| `mvcur(`*lasty*`,` *lastx*`,` *newy*`,` *newx*`)` | Actually move cursor. |
| `newwin(`*lines*`,` *cols*`,` *begin_y*`,` *begin_x*`)` | Create a new window. |
| `nl()` | Set newline mapping. |
| `nocrmode()` | Unset `cbreak` mode. |
| `noecho()` | Unset echo mode. |
| `nonl()` | Unset newline mapping. |
| `noraw()` | Unset raw mode. |
| `overlay(`*win1*`,` *win2*`)` | Overlay *win1* on *win2*. |
| `overwrite(`*win1*`,` *win2*`)` | Overwrite *win1* on top of *win2*. |
| `printw(`*fmt*`,` *arg1*`,` *arg2*`,...)` | `printf` on `stdscr`. |
| `raw()` | Set raw mode. |
| `refresh()+` | Make current screen look like `stdscr`. |
| `resetty()` | Reset tty flags to stored value. |
| `savetty()` | Stored current tty flags. |
| `scanw(`*fmt*`,` *arg1*`,` *arg2*`,...)` | `scanf` through `stdscr`. |
| `scroll(`*win*`)` | Scroll *win* one line. |
| `scrollok(`*win*`,` *boolf*`)+` | Set scroll flag. |
| `setterm(`*name*`)` | Set term variables for *name*. |
| `standend()+` | End standout mode. |
| `standout()+` | Start standout mode. |
| `subwin(`*win*`,` *lines*`,` *cols*`,` *begin_y*`,` *begin_x*`)` | Create a subwindow. |
| `touchwin(`*win*`)` | Change all of *win*. |
| `unctrl(`*ch*`)` | Printable version of *ch*. |
| `waddch(`*win*`,` *ch*`)` | Add character *ch* to *win*. |
| `waddstr(`*win*`,` *str*`)` | Add string to *win*. |
| `wclear(`*win*`)` | Clear *win*. |

| | |
|---|---|
| `wclrtobot(`*win*`)` | Clear to bottom of *win*. |
| `wclrtoeol(`*win*`)` | Clear to end-of-line on *win*. |
| `wdelch(`*win*`,`*c*`)` | Delete character from *win*. |
| `wdeleteln(`*win*`)` | Delete line from *win*. |
| `werase(`*win*`)` | Erase *win*. |
| `wgetch(`*win*`)` | Get a character through *win*. |
| `wgetstr(`*win*`,`*str*`)` | Get a string through *win*. |
| `winch(`*win*`)+` | Get character at current (*y*,*x*) in *win*. |
| `winsch(`*win*`,`*c*`)` | Insert character into *win*. |
| `winsertln(`*win*`)` | Insert line into *win*. |
| `wmove(`*win*`,`*y*`,`*x*`)` | Set current (*y*,*x*) coordinates on *win*. |
| `wprintw(`*win*`,`*fmt*`,`*arg1*`,`*arg2*`,...)` | `printf` on *win*. |
| `wrefresh(`*win*`)` | Make screen look like *win*. |
| `wscanw(`*win*`,`*fmt*`,`*arg1*`,`*arg2*`,...)` | `scanf` through *win*. |
| `wstandend(`*win*`)` | End standout mode on *win*. |
| `wstandout(`*win*`)` | Start standout mode on *win*. |

**SEE ALSO**

`ioctl`(2), `curses`(3X), `getenv`(3), `termcap`(4), `terminfo`(4), `tty`(4).

## NAME

cuserid — get character login name of the user

## SYNOPSIS

```
#include <stdio.h>

char *cuserid(s)
char *s;
```

## DESCRIPTION

cuserid generates a character string representation of the effective user ID of the current process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least L_cuserid characters; the representation is left in this array. The constant L_cuserid is defined in the <stdio.h> header file.

If multiple character strings are associated with a user ID, the first one encountered in /etc/passwd will be returned.

## RETURN VALUE

If the character string representing the login name associated with the effective user ID of the current process cannot be found, cuserid returns a NULL pointer; if *s* is not a NULL pointer, a null character (\0) is placed at *s*[0].

## FILES

/etc/passwd

## SEE ALSO

geteuid within getuid(2), cuserid(3S), getlogin(3C), getpwent(3C).

## BUGS

cuserid uses getpwnam(3C); thus the results of a user's call to the latter will be obliterated by a subsequent call to the former.

The name cuserid is rather a misnomer.

## NAME

cuserid — get character login name of the user

## SYNOPSIS

```
#include <stdio.h>

char *cuserid(s)
char *s;
```

## DESCRIPTION

cuserid generates a character string representation of the login name of the owner of the current process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least L_cuserid characters; the representation remains in this array. The constant L_cuserid is defined in the <stdio.h> header file.

## RETURN VALUE

If the login name cannot be found, cuserid returns a NULL pointer; if *s* is not a NULL pointer, a null character (\0) is placed at *s*[0].

## SEE ALSO

getuid(2), cuserid(3P), getlogin(3C), getpwent(3C).

## BUGS

cuserid uses getpwnam(3C); therefore, the results of a user's call to the latter will be obliterated by a subsequent call to the former.

The name cuserid is rather a misnomer.

## NAME

dbminit, fetch, store, delete, firstkey, nextkey — database subroutines

## SYNOPSIS

```
typedef struct {
    char *dptr;
    int dsize;
} datum;

dbminit(file)
char *file;

datum *fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

## DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option `-ldbm`.

*key*s and *content*s are described by the `datum typedef`. A `datum` specifies a string of `dsize` bytes pointed to by `dptr`. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has `.dir` as its suffix. The second file contains all data and has `.pag` as its suffix.

Before a database can be accessed, it must be opened by `dbminit`. At the time of this call, the files *file*`.dir` and *file*`.pag` must exist. (An empty database is created by creating zero-length `.dir` and `.pag` files.)

Once open, the data stored under a key is accessed by `fetch` and data is placed under a key by `store`. A key (and its associated contents) is deleted by `delete`. A linear pass through all keys in a database may be made, in an (apparently) random order, by use

of `firstkey` and `nextkey`. `firstkey` will return the first key in the database. With any key `nextkey` will return the next key in the database. This code will traverse the data base:

```
for(key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

## RETURN VALUE

All functions that return an `int` indicate errors with negative values. A zero return indicates okay. Routines that return a `datum` indicate errors with a null (0) `dptr`.

## BUGS

The `.pag` file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (`cp`, `cat`, `tp`, `tar`, `ar`) without filling in the holes.

`dptr` pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. `store` will return an error in the event that a disk block fills with inseparable data.

`delete` does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by `firstkey` and `nextkey` depends on a hashing function, not on anything interesting.

## NAME

ddp_open, ddp_close — AppleTalk Datagram Delivery Protocol (DDP) interface

## SYNOPSIS

```
#include <at/appletalk.h>
#include <at/ddp.h>
cc [flags] files -lat [libraries]

int ddp_open(socket)
at_socket *socket;

int ddp_close(fd)
int fd;
```

## DESCRIPTION

The DDP interface provides applications with access to the AppleTalk DDP operations.

The ddp_open routine opens a static or dynamic DDP socket and returns an AppleTalk socket file descriptor that can be used to read and write DDP datagrams. The parameters are

*socket* A pointer to the DDP socket number to open. If the AppleTalk socket number is 0, or if *socket* is NULL, a DDP socket is dynamically assigned. If *socket* is non-NULL, the socket number is returned in *socket*.

An error condition results if there are no more dynamic DDP sockets available, if the maximum number of open files has been exceeded at a process or system level, or if the network is offline.

Only the superuser can open a static DDP socket.

*fd* The AppleTalk file descriptor of the DDP socket to be closed by the ddp_close routine.

Datagrams are always read and written with the long DDP header format, using standard A/UX read(2) and write(2) system calls. The long header DDP datagram is defined by this structure in <at/ddp.h>:

```
typedef struct {
        u_short     unused : 2,
                    hopcount : 4,
                    length : 10;
        u_short     checksum;
```

```
        at_net      dst_net;
        at_net      src_net;
        at_node     dst_node;
        at_node     src_node;
        at_socket   dst_socket;
        at_socket   src_socket;
        u_char      type;
        u_char      data[DDP_DATA_SIZE];
} at_ddp_t;
```

When a datagram is written, only the fields `checksum`, `dst_net`, `dst_node`, `dst_socket`, `type`, and `data` need to be set. The rest of the fields may be left uninitialized, because DDP sets them.

The `length` field is the DDP packet length. The `checksum` field contains the DDP checksum. When datagrams are sent, a checksum is computed only if this field is nonzero.

Datagrams can be sent and received asynchronously using standard A/UX facilities: `select`(2N) or `O_NDELAY` `fcntl`(2).

## ERRORS

All routines return −1 on error with detailed error code in `errno`:

| | |
|---|---|
| `[EACCES]` | A nonsuperuser attempted to open a static AppleTalk socket. |
| `[EADDRINUSE]` | The static socket is in use, or all dynamic sockets are in use. |
| `[EINVAL]` | An attempt was made to open an invalid AppleTalk socket number. |
| `[EMSGSIZE]` | A datagram is too large or too small. |
| `[ENETDOWN]` | The network interface is down. |
| `[ENOBUFS]` | DDP is out of buffers. |

Routines also return any additional error codes returned by standard A/UX `open`(2), `close`(2), `read`(2), and `write`(2) system calls.

## FILES

`/dev/appletalk/ddp/*`

**SEE ALSO**

close(2), fcntl(2), open(2), read(2), select(2N), write(2), atp(3N), ddp(3N), nbp(3N), pap(3N), rmtp(3N), fcntl(5), termio(7), *Inside AppleTalk;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming*.

## NAME

dial — establish an out-going terminal line connection

## SYNOPSIS

```
#include <dial.h>

int dial(call)
CALL call;

void undial(fd)
int fd;
```

## DESCRIPTION

dial returns a file descriptor for a terminal line open for read/write. The argument to dial is a CALL structure (defined in the <dial.h> header file).

When finished with the terminal line, the calling program must invoke undial to release the semaphore that has been set during the allocation of the terminal device.

The CALL typedef in the <dial.h> header file is:

```
typedef struct {
      struct termio *attr;  /* pointer to termio
                               attribute struct */
      int           baud;   /* transmission data rate */
      int           speed;  /* 212A modem: low=300,
                               high=1200 */
      char          *line;  /* device name for
                               out-going line */
      char          *telno  /* pointer to tel-no digits
                               string */
      int           modem;  /* specify modem control for
                               direct lines */
      char         *device; /* 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;
```

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-speed 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 2121 modem transmits and receives at 1200 bits per second only. The CALL element baud is for the desired transmis-

sion 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 termination symbol will be supplied by the dial function, and should not be included in the telno string passed to dial in the CALL structure.

The CALL element modem is used to specify modem control for direct lines. This element should be nonzero 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 important for attributes such as parity and baud rate.

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.

## ERRORS

On failure, a negative value indicating the reason for the failure is returned. Mnemonics for these negative indices as listed here are defined in the <dial.h> header file.

```
INTRPT  -1  /* interrupt occurred */
D_HUNG  -2  /* dialer hung (no return from write) */
NO_ANS  -3  /* no answer within 10 seconds */
ILL_BD  -4  /* illegal baud-rate */
A_PROB  -5  /* acu problem (open() failure) */
L_PROB  -6  /* line problem (open() failure) */
NO_Ldv  -7  /* can't 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 requested baud */
```

```
NO_BD_K -11 /* no device known at requested baud */
```

**FILES**

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..*tty-device*

**SEE ALSO**

uucp(1C), alarm(2), read(2), write(2), termio(7).

**WARNINGS**

Including the <dial.h> header file automatically includes the <termio.h> header file.

Because the above routine uses <stdio.h>, the size of programs not otherwise using standard I/O is increased more than might be expected.

**BUGS**

An alarm(2) system call for 3,600 seconds is made (and caught) within the dial module for the purpose of "touching" the LCK.. file and constitutes the device allocation semaphore for the terminal device. Otherwise, uucp(1C) may simply delete the LCK.. entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a read(2) or write(2) system call, causing an apparent error return. If the user program is to run for an hour or more, error returns from reads should be checked for (errno==EINTR), and the read possibly reissued.

**NAME**

dim, ddim, idim — Fortran positive difference intrinsic functions

**SYNOPSIS**

```
integer a1, a2, a3
a3=idim(a1, a2)

real a1, a2, a3
a3=dim(a1, a2)

double precision a1, a2, a3
a3=ddim(a1, a2)
```

**DESCRIPTION**

These functions return:

| | |
|---|---|
| *a1*-*a2* | if *a1* > *a2* |
| 0 | if *a1* <= *a2* |

## NAME

opendir, readdir, telldir, seekdir, rewinddir, closedir — directory operations

## SYNOPSIS

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

DIR *opendir(filename)
char *filename;

struct direct *readdir(dirp)
DIR *dirp;

void rewinddir(dirp)
DIR *dirp;

int closedir(dirp)
DIR *dirp;
```

## DESCRIPTION

`opendir` opens the directory named by *filename* and associates a directory stream with it. `opendir` returns a pointer to be used to identify the directory stream in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed, or if it cannot allocate enough memory to hold the whole thing.

`readdir` returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory or detecting an invalid `seekdir` operation.

The `rewinddir` macro resets the position of the named directory stream to the beginning of the directory. It also causes the directory stream to refer to the current state of the directory.

`closedir` closes the named directory stream and frees the structure associated with the `DIR` pointer.

Sample code that searches a directory for an entry *name* is:

```
len = strlen(name);
dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
    if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
        closedir(dirp);
        return FOUND;
}
closedir(dirp);
return NOT_FOUND;
```

The result of using a directory stream after an exec(2) is undefined. After a fork(2), either the parent or the child (but not both) may continue processing the directory stream by using readdir or rewinddir, or both.

**SEE ALSO**

ls(1), open(2), close(2), getdirentries(2), read(2), lseek(2), dir(4).

## NAME

opendir, readdir, telldir, seekdir, rewinddir, closedir — directory operations

## SYNOPSIS

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

DIR *opendir(filename)
char *filename;

struct direct *readdir(dirp)
DIR *dirp;

void rewinddir(dirp)
DIR *dirp;

int closedir(dirp)
DIR *dirp;
```

## DESCRIPTION

`opendir` opens the directory named by *filename* and associates a *directory stream* with it. `opendir` returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed, or if it cannot `malloc`(3) enough memory to hold the whole thing.

`readdir` returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory or detecting an invalid `seekdir` operation.

The `rewinddir` macro resets the position of the named *directory stream* to the beginning of the directory. It also causes the directory stream to refer to the current state of the directory.

`closedir` closes the named *directory stream* and frees the structure associated with the `DIR` pointer.

Sample code which searchs a directory for entry *name* is:

```
len = strlen(name);
dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
    if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
        closedir(dirp);
        return FOUND;
}
closedir(dirp);
return NOT_FOUND;
```

The result of using a directory stream after an exec(2) is undefined. After a fork(2), either the parent of the child (but not both) may continue processing the directory stream using readdir and/or rewinddir .

**SEE ALSO**

ls(1), open(2), close(2), getdirentries(2), read(2), lseek(2), dir(4).

## NAME

dprod — Fortran double precision product intrinsic function

## SYNOPSIS

```
real a1, a2
double precision a3
a3=dprod(a1, a2)
```

## DESCRIPTION

dprod returns the double precision product of its real arguments.

## NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 — generate uniformly distributed pseudo-random numbers

## SYNOPSIS

```
double drand48()

double erand48(xsubi)
unsigned short xsubi[3];

long lrand48()

long nrand48(xsubi)
unsigned short xsubi[3];

long mrand48()

long jrand48(xsubi)
unsigned short xsubi[3];

void srand48(seedval)
long seedval;

unsigned short *seed48(seed16v)
unsigned short seed16v[3];

void lcong48(param)
unsigned short param[7];
```

## DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions drand48 and erand48 return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions lrand48 and nrand48 return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

Functions mrand48 and jrand48 return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$. Functions srand48,seed48, and lcong48 are initialization entry points, one of which should be invoked before drand48,lrand48, or mrand48 is called. (Although it is not recommended practice, constant default initializer values are supplied automatically if drand48,lrand48, or mrand48 is called without a prior call to an initialization entry point.) Functions erand48, nrand48,

and jrand48 do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, $X_i$, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \qquad n \geq 0$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless lcong48 has been invoked, the multiplier value $a$ and the addend value $c$ are given by

$$a = 5\text{DEECE66D}_{16} = -273673163155_8$$
$$c = \text{B}_{16} = 13_8.$$

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, i.e., the sequence of numbers in each stream does *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function srand48 sets the high-order 32 bits of $X_i$ to the 32 bits contained in its argument. The low-order 16 bits of $X_i$ are set to the arbitrary value $330\text{E}_{16}$.

The initializer function seed48 sets the value of $X_i$ to the 48-bit value specified in the argument array. The previous value of $X_i$ is copied into a 48-bit internal buffer, used only by seed48. A pointer to this buffer is the value returned by seed48. The returned pointer, which can be ignored if not needed, is useful if a program is to be restarted from a given point at some future time.

Use the pointer to get and store the last $X_i$ value; then use this value to reinitialize via `seed48` when the program is restarted.

The initialization function `lcong48` allows the user to specify the initial $X_i$, the multiplier value $a$, and the addend value $c$. Argument array elements *param*[0-2] specify $X_i$, elements *param*[3-5] specify the multiplier $a$, and *param*[6] specifies the 16-bit addend $c$. After `lcong48` has been called, a subsequent call to either `srand48` or `seed48` will restore the "standard" multiplier and addend values, $a$ and $c$, specified on the previous page.

NOTES

The routines are coded in portable C. The source code for the portable version can even be used on computers which do not have floating-point arithmetic. In such a situation, functions `drand48` and `erand48` do not exist; instead, they are replaced by the following two functions:

```
long irand48 (m)
unsigned short m;

long krand48 (xsubi,m)
unsigned short xsubi[3],m;
```

Functions `irand48` and `krand48` return non-negative long integers uniformly distributed over the interval [0, $m$−1].

SEE ALSO

`rand`(3C).

## NAME

dup2 — duplicate a descriptor

## SYNOPSIS

```
dup2 (oldd, newd)
int oldd, newd;
```

## DESCRIPTION

dup2 causes *newd* to become a duplicate of *oldd*. If *newd* is already in use, the descriptor is first deallocated as if a close(2) call had been done first.

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

## RETURN VALUE

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

## ERRORS

dup2 fails if:

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

## SEE ALSO

accept(2N), close(2), dup(2), fcntl(2), getdtablesize(2N), open(2), pipe(2), socket(2N).

## NAME

ecvt, fcvt, gcvt — convert floating-point number to string

## SYNOPSIS

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

## DESCRIPTION

ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to this string. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero; otherwise it is zero.

fcvt is identical to ecvt, except that the correct digit has been rounded for printf %f (Fortran F-format) output of the number of digits specified by *ndigit*.

gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in Fortran F-format, ready for printing; E-format is produced when F-format is not possible. A minus sign, if there is one, or a decimal point is included as part of the returned string. Trailing zeros are suppressed.

## SEE ALSO

printf(3S).

## BUGS

The values returned by ecvt and fcvt point to a single static data array.

## NAME

end, etext, edata — last locations in program

## SYNOPSIS

```
extern end;
extern etext;
extern edata;
```

## DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of `etext` is the first address above the program text, `edata` above the initialized data region, and `end` above the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with `end`, but the program break may be reset by the routines of `brk`(2), `malloc`(3C), standard input/output, the profile (`-p`) option of `cc`(1), and so on. Thus, the current value of the program break should be determined by `sbrk(0)` (see `brk`(2)).

## SEE ALSO

`cc`(1), `brk`(2), `intro`(3), `malloc`(3C).

## NAME

erf, erfc — error function and complementary error function

## SYNOPSIS

```
#include <math.h>

double erf(x)
double x;

double erfc(x)
double x;
```

## DESCRIPTION

The erf function returns the error function of $x$ (the precise formula is available in at standard calculus text).

erfc, which returns 1.0 − erf($x$), is provided because of the extreme loss of relative accuracy if erf($x$) is called for large $x$ and the result subtracted from 1.0 (e.g. for $x$ = 5, 12 places are lost).

## SEE ALSO

exp(3M).

## NAME

ethers, ether_ntoa, ether_aton, ether_ntohost, ether_hostton, ether_line — Ethernet address mapping operations

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>

char *ether_ntoa(e)
struct ether_addr *e;

struct ether_addr *ether_aton(s)
char *s;

ether_ntohost(hostname, e)
char *hostname;
struct ether_addr *e;

ether_hostton(hostname, e)
char *hostname;
struct ether_addr *e;

ether_line(l, e, hostname)
char *l;
struct ether_addr *e;
char *hostname;
```

## DESCRIPTION

These routines are useful for mapping 48-bit Ethernet numbers to their ASCII representations or their corresponding host names, and vice versa.

The function ether_ntoa converts a 48-bit Ethernet number pointed to by *e* to its standard ACSII representation; it returns a pointer to the ASCII string. The representation is of the form:

*x*:*x*:*x*:*x*:*x*:*x*:

where *x* is a hexadecimal number between 0 and 255. The function ether_aton converts an ASCII string in the standard representation back to a 48-bit Ethernet number; the function returns NULL if the string cannot be scanned successfully.

The function ether_ntohost maps an Ethernet number (pointed to by *e*) to its associated hostname. The string pointed to by *hostname* must be long enough to hold the hostname and a null character. The function returns zero upon success and non-zero upon failure. Inversely, the function ether_hostton maps a hostname string to its corresponding Ethernet number; the function modifies the Ethernet number pointed to by *e*. The function also returns zero upon success and non-zero upon failure.

The function ether_line scans a line (pointed to by *l*) and sets the hostname and the Ethernet number (pointed to by *e*). The string pointed to by *hostname* must be long enough to hold the hostname and a null character. The function returns zero upon success and non-zero upon failure. The format of the scanned line is described by ethers(4).

**FILES**

| | |
|---|---|
| /etc/ethers | |
| /etc/ethers.byaddr | Yellow Pages control file |
| /etc/ethers.byname | Yellow Pages control file |

**SEE ALSO**

ethers(4).

**NAME**

exp, dexp, cexp — Fortran exponential intrinsic function

**SYNOPSIS**

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2=exp(r1)

dp2=dexp(dp1)
dp2=exp(dp1)

cx2=cexp(cx1)
cx2=exp(cx1)
```

**DESCRIPTION**

exp returns the real exponential function $e^x$ of its real argument. dexp returns the double-precision exponential function of its double-precision argument. cexp returns the complex exponential function of its complex argument. The generic function exp becomes a call to dexp or cexp, as required, depending on the type of its argument.

**SEE ALSO**

exp(3M).

## NAME

exp, log, log10, pow, sqrt — exponential, logarithm, power, and square root functions

## SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x, y;

double sqrt(x)
double x;
```

## DESCRIPTION

The exp function returns *e* raised to the power of *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.

The pow function returns *x* raised to the power of *y*. If *x* is zero, *y* must be positive. If *x* is negative, *y* must be an integer.

sqrt returns the nonnegative square root of *x*. The value of *x* may not be negative.

## RETURN VALUE

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 nonpositive. 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 nonpositive, 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(3M).

**SEE ALSO**

intro(2), hypot(3M), matherr(3M), sinh(3M).

## NAME

fclose, fflush — close or flush a stream

## SYNOPSIS

```
#include <stdio.h>

int fclose(stream)
FILE *stream;

int fflush(stream)
FILE *stream;
```

## DESCRIPTION

fclose causes any buffered data for the named *stream* to be written out and the *stream* to be closed.

fclose is performed automatically for all open files upon calling exit(2).

fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

## RETURN VALUE

These functions return 0 for success, and EOF if any error (such as trying to write to a file that has not been opened for writing) was detected.

## SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S).

## NAME

ferror, feof, clearerr, fileno — stream status inquiries

## SYNOPSIS

```
#include <stdio.h>

int feof(stream)
FILE *stream;

int ferror(stream)
FILE *stream;

void clearerr(stream)
FILE *stream;

int fileno(stream)
FILE *stream;
```

## DESCRIPTION

feof returns nonzero when EOF has previously been detected reading the named input *stream*; otherwise, it returns zero.

ferror returns nonzero when an I/O error has previously occurred reading from or writing to the named *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(2).

## NOTES

All these functions are implemented as macros; they cannot be declared or redeclared.

## SEE ALSO

open(2), fopen(3S).

## NAME

floor, ceil, fmod, fabs — floor, ceiling, remainder, absolute value functions

## SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fmod(x, y)
double x, y;

double fabs(x)
double x;
```

## DESCRIPTION

floor returns the largest integer (as a double-precision number) not greater than $x$.

ceil returns the smallest integer not less than $x$.

fmod returns the floating-point remainder of the division of $x$ by $y$: $x$ if $y$ is zero or if $x/y$ would overflow; otherwise the number is $f$ with the same sign as $x$, such that $x = iy + f$ for some integer $i$, and $|f| < |y|$.

fabs returns the absolute value of $|x|$.

## SEE ALSO

abs(3C).

## NAME

fopen, freopen, fdopen — open a stream

## SYNOPSIS

```
#include <stdio.h>

FILE *fopen (filename, type)
char *filename, *type;

FILE *freopen (filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

## DESCRIPTION

fopen opens the file named by *filename* and associates a *stream* with it. fopen returns a pointer to the FILE structure associated with the *stream*.

*filename* points to a character string that contains the name of the file to be opened.

*type* is a character string having one of the following values:

| | |
|---|---|
| r | open for reading |
| w | truncate or create for writing |
| a | append; open for writing at end of file, or create for writing |
| r+ | open for update (reading and writing) |
| w+ | truncate or create for update |
| a+ | append; open or create for update at end-of-file |

freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. freopen returns a pointer to the FILE structure associated with *stream*.

freopen is typically used to attach the preopened *streams* associated with stdin, stdout, and stderr to other files.

fdopen associates a *stream* with a file descriptor by formatting a file structure from the file descriptor. Thus, fdopen can be used to access the file descriptors returned by open(2), dup(2), creat(2), or pipe(2). (These calls open files but do not return

pointers to a FILE structure.) The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening fseek or rewind, and input may not be directly followed by output without an intervening fseek, rewind, or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when *type* is a or a+), it is impossible to overwrite information already in the file. fseek may be used to reposition the file pointer to any position in the file, but when output is written to the file the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

**RETURN VALUE**

fopen and freopen return a NULL pointer on failure.

**SEE ALSO**

creat(2), dup(2), open(2), pipe(2), fclose(3S), fseek(3S).

## NAME

fread, fwrite — binary input/output

## SYNOPSIS

```
#include <stdio.h>

int fread(ptr, size, nitems, stream)
char *ptr;
int  size, nitems;
FILE *stream;

int fwrite(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

## DESCRIPTION

`fread` copies *nitems* items of data from the named input *stream* into an array beginning at *ptr*. An item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. `fread` stops appending bytes if an end-of-file or error condition is encountered while reading *stream* or if *nitems* items have been read. `fread` leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. `fread` does not change the contents of *stream*.

`fwrite` appends at most *nitems* items of data from the the array pointed to by *ptr* to the named output *stream*. `fwrite` stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. `fwrite` does not change the contents of the array pointed to by *ptr*.

The variable *size* is typically `sizeof(*ptr)` where the pseudo-function `sizeof` specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

## RETURN VALUE

`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(2), write(2), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S).

## NAME

frexp, ldexp, modf — manipulate parts of floating-point numbers

## SYNOPSIS

```
double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;
int exp;

double modf(value, iptr)
double value, *iptr;
```

## DESCRIPTION

Every nonzero number can be written uniquely as $x$* pow(2,$n$), where the "mantissa" (fraction) $x$ is in the range $0.5 \le |x| < 1.0$, and the "exponent" $n$ is an integer. frexp returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by frexp are zero.

ldexp returns the quantity *value** pow(2,*exp*).

modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

## ERRORS

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 ERRANGE.

## SEE ALSO

exp(3M).

## NAME

fseek, rewind, ftell — reposition a file pointer in a stream

## SYNOPSIS

```
#include <stdio.h>

int fseek(stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind(stream)
FILE *stream;

long ftell(stream)
FILE *stream;
```

## DESCRIPTION

fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, when the value of *ptrname* is 0, 1, or 2, respectively.

rewind(*stream*) is equivalent to fseek(*stream*, 0L, 0), except that no value is returned.

fseek and rewind undo any effects of ungetc(3S).

After fseek or rewind, the next operation on a file opened for update may be either input or output.

ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

## RETURN VALUE

fseek returns non-zero for improper seeks; otherwise it returns 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(3S).

## SEE ALSO

lseek(2), fopen(3S), popen(3S), ungetc(3S).

**WARNINGS**

On A/UX an offset returned by `ftell` is measured in bytes, and it is permissible to seek to positions relative to that offset; however, portability to systems other than A/UX requires that an offset be used by `fseek` directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

## NAME

`fstyp` — determine the file-system type

## SYNOPSIS

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

struct fstypent *fstype(name)
char *name;
```

## DESCRIPTION

`fstyp` determines the type of the file system associated with the file *name*. The file-system types supported on A/UX® are listed in `/etc/fstypes`. See `fstypes`(4).

If the file *name* is a block or character device and is readable by the calling process, `fstyp` attempts to determine the file-system type by executing (through `exec`(2)) the file-system-dependent versions of the `fstyp`(1) command. These commands read the file-sytem superblocks and perform consistency checks on the file-system data. See `fs`(4). Otherwise, `fstyp` uses the information returned by `statfs`(2).

## RETURN VALUE

The return value is a pointer to a `fstypent` structure. If the file-system type cannot be determined, a NULL pointer is returned.

## FILES

```
/etc/fstab
/etc/fstypes
/etc/mtab
/etc/fs/*/fstyp
```

## NOTES

A/UX currently supports System V file systems (SVFS) and Berkeley Fast File Systems (UFS) as local systems.

## SEE ALSO

`stat`(2), `statfs`(2), `fstypent`(3), `getmntent`(3), `fs`(4), `fstab`(4), `fstypes`(4), `mtab`(4).

## NAME

fstypent — get file-system-type entry

## SYNOPSIS

```
#include <stdio.h>
#include <sys/fstypent.h>

struct fstypent *fstypent(filep)
FILE *filep;
```

## DESCRIPTION

fstypent reads the next line from the file indicated by *filep* and returns a pointer to a struct fstypent containing this information.

The fstypent structure is defined in <sys/fstypent.h>:

```
struct fstypent {
        int fstype;
        char **typelist;
        char *pathlist;
};
```

In the above structure, fstype indicates the file-system type, and this value is used by fsmount(2).

typelist is a null-terminated list of pointers; each one points to a character string describing a file-system type. At least one of these types is defined in <mntent.h>.

pathlist is a colon-separated list of pathnames. The pathnames indicate the directories where file-system-dependent utilities may be found. This entry in /etc/fstypes is not required. pathlist is a NULL pointer if the entry is missing.

The data in this structure and referenced static data are overwritten by a subsequent call to fstypent or typefs.

## RETURN VALUE

fstypent returns a pointer of type struct fstypent. See fstyp(3). A NULL pointer is returned on end-of-file or error.

## FILES

/etc/fstypes

**SEE ALSO**

getmntent(3), typefs(3), fs(4), fstab(4), fstypes(4), ufs(4).

## NAME

ftok — standard interprocess communication package

## SYNOPSIS

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

key_t ftok(path, id)
char *path;
char id;
```

## DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the msgget(2), semget(2), and shmget(2) system calls to obtain interprocess communication identifiers. One method for forming a key is to use the ftok subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If a standard is not adhered to, unrelated processes may 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 pathname of an existing file that is accessible to the process. *id* is a character that uniquely identifies a project. ftok returns the same key for linked files when called with the same *id*; it returns different keys when called with the same filename but different *ids*.

## SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

## DIAGNOSTICS

ftok returns (key_t) -1 if *path* does not exist or if it is not accessible to the process.

## WARNINGS

If the file whose *path* is passed to ftok is removed when keys still refer to the file, future calls to ftok with the same *path* and *id* will return an error. If the same file is recreated, ftok is likely to return a different key than it did the original time it was called.

## NAME

ftw — walk a file tree

## SYNOPSIS

```
#include <ftw.h>

int ftw(path, fn, depth)
char *path;
int (*fn)();
int depth;
```

## DESCRIPTION

ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, ftw calls *fn*, passing it a pointer to a nullterminated character string containing the name of the object, a pointer to a stat structure (see stat(2)) containing information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which stat could not be executed successfully. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the stat structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* is a file in a directory with read permission but not execute (search) permission.

ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or an error is detected within ftw (such as an I/O error). If the tree is exhausted, ftw returns zero. If *fn* returns a nonzero value, ftw stops its tree traversal and returns whatever value was returned by *fn*. If ftw detects an error, it returns –1, and sets the error type in errno.

ftw uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *depth* must not be greater than the number of file descriptors currently available for use. ftw runs more quickly if *depth* is at least as large as the number of levels in the tree.

## RETURN VALUE

The tree traversal continues until the tree is exhausted, and invocation of *fn* returns a nonzero value or some error is detected within ftw (such as an I/O error). If the tree is exahusted, ftw

returns 0. If *fn* returns a nonzero value, `ftw` stops its tree traversal and returns whatever value was returned by *fn*.

If `ftw` encounters an error other than `EACCESS`, it returns −1 and `errno` is set to indicate the error. The external variable `errno` may contain the error values that are possible when a directory is opened or when `stat`(2) is executed on a directory or file.

**SEE ALSO**

`stat`(2), `malloc`(3C).

**BUGS**

Because `ftw` is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

`ftw` could be made to run faster and use less storage on deep structures at the cost of considerable complexity.

`ftw` uses `malloc`(3C) to allocate dynamic storage during its operation. If `ftw` is forcibly terminated, such as by `longjmp` being executed by *fn* or an interrupt routine, `ftw` does not have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

## NAME

int, ifix, idint, real, float, sngl, dble, cmplx, dcmplx, ichar, char — explicit Fortran type conversion

## SYNOPSIS

```
integer i, j
real r, s
double precision dp, dq
complex cx
double complex dcx
character *1 ch

i=int(r)
i=int(dp)
i=int(cx)
i=int(dcx)
i=ifix(r)
i=idint(dp)

r=real(i)
r=real(dp)
r=real(cx)
r=real(dcx)
r=float(i)
r=sngl(dp)

dp=dble(i)
dp=dble(r)
dp=dble(cx)
dp=dble(dcx)

cx=cmplx(i)
cx=cmplx(i, j)
cx=cmplx(r)
cx=cmplx(r, s)
cx=cmplx(dp)
cx=cmplx(dp, dq)
cx=cmplx(dcx)

dcx=dcmplx(i)
dcx=dcmplx(i, j)
dcx=dcmplx(r)
dcx=dcmplx(r, s)
dcx=dcmplx(dp)
dcx=dcmplx(dp, dq)
dcx=dcmplx(cx)
```

*i*=ichar(*ch*)
*ch*=char(*i*)

DESCRIPTION

These functions perform conversion from one data type to another.

int converts to integer form its real, double precision, complex, or double complex argument. If the argument is real or double precision, int returns the integer whose magnitude is the largest integer that does not exceed the magnitude of the argument and whose sign is the same as the sign of the argument (i.e., truncation). For complex types, the above rule is applied to the real part. ifix and idint convert only real and double precision arguments respectively.

real converts to real form an integer, double precision, complex, or double complex argument. If the argument is *double precision* or *double complex*, as much precision is kept as is possible. If the argument is one of the complex types, the real part is returned. float and sngl convert only integer and double precision arguments, respectively.

dble converts any integer, real, complex, or double complex argument to double precision form. If the argument is of a complex type, the real part is returned.

cmplx converts its integer, real, double precision, or double complex argument(s) to complex form.

dcmplx converts its integer, real, double precision, or complex argument(s) to double complex form.

Either one or two arguments may be supplied to cmplx and dcmplx. If there is only one argument, it is taken as the real part of the complex type and a imaginary part of zero is supplied. If two arguments are supplied, the first is taken as the real part and the second as the imaginary part.

ichar converts from a character to an integer depending on the character's position in the collating sequence.

char returns the character in the *i*th position in the processor collating sequence, where *i* is the supplied argument.

For a processor capable of representing *n* characters,

`ichar(char(`*i*`))` = *i* for 0 <= *i* < *n*, and

`char(ichar(`*ch*`))` = *ch* for any representable character *ch*.

## NAME

gamma — log gamma function

## SYNOPSIS

```
#include <math.h>

extern int signgam;

double gamma(x)
double x;
```

## DESCRIPTION

gamma returns the natural log of gamma as a function of the absolute value of a given value. gamma returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as

$$\int_0^\infty e^{-t}t^{x-1}dt.$$

The sign of $\Gamma(x)$ is returned in the external integer signgam. The argument $x$ may not be a nonpositive 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 exp(3M) to return a range error, and is defined in the <values.h> header file.

## RETURN VALUE

For non-negative integer arguments HUGE is returned, and errno is set to EDOM. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, gamma returns HUGE and sets errno to ERANGE.

These error-handling procedures may be changed with the function matherr(3M).

## SEE ALSO

exp(3M), matherr(3M), values(5).

## NAME

getarg — return Fortran command-line argument

## SYNOPSIS

```
character *N c
integer i
getarg(i, c)
```

## DESCRIPTION

getarg returns the *i*th command-line argument of the current process. Thus, if a program were invoked with:

```
foo arg1 arg2 arg3
```

then the call

```
getarg(2, c)
```

would return the string *arg2* in the character variable c.

## SEE ALSO

getopt(3C).

## NAME

getc, getchar, fgetc, getw — get character or word from a stream

## SYNOPSIS

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

## DESCRIPTION

The getc macro returns the next character (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. The getchar macro is defined as getc(stdin).

fgetc behaves like getc, but is a function rather than a macro. fgetc runs more slowly than getc, but takes less space per invocation and its name can be passed as an argument to a function.

getw returns the next word (32-bit integer on a Macintosh II) from the named input *stream*. getw increments the associated file pointer, if defined, to point to the next word. getw assumes no special alignment in the file.

## RETURN VALUE

These functions return the constant EOF at end-of-file or upon an error. Because EOF is a valid integer, ferror(3S) should be used to detect getw errors.

## SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S), ungetc(3S).

## WARNINGS

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 integer is machine-dependent.

## BUGS

Because it is implemented as a macro, `getc` treats incorrectly 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 `putw` are machine-dependent, and may not be read using `getw` on a different processor.

## NAME

getcwd — get the pathname of the current working directory

## SYNOPSIS

```
char *getcwd(buf, size)
char *buf;
int size;
```

## DESCRIPTION

getcwd returns a pointer to the current directory pathname. The value of *size* must be at least two greater than the length of the pathname to be returned.

If *buf* is a NULL pointer, getcwd obtains *size* bytes of space by using malloc(3C). In this case, the pointer returned by getcwd may be used as the argument in a subsequent call to free.

The function is implemented by using popen(3S) to pipe the output of the pwd(1) command into the specified string space.

## EXAMPLES

```
#include <limits.h>
char *cwd, *getcwd();
.
.
.
if ((cwd=getcwd((char *)NULL, PATH-MAX))==NULL) {
        perror(``pwd'');
        exit(1);
}
printf("%s\n", cwd);
```

## RETURN VALUE

If any of the following conditions occur, getcwd returns NULL and sets errno to the corresponding value:

| | |
|---|---|
| [EINVAL] | The size is less than or equal to 0. |
| [ERANGE] | The size is not large enough to contain the pathname. |
| [EACCES] | The read or search permission was denied for a component of the pathname. |

## ERRORS

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

| | |
|---|---|
| [EINVAL] | The size is less than or equal to 0. |

| | |
|---|---|
| [ERANGE] | The size is not large enough to contain the pathname. |
| [EACCES] | The read or search permission was denied for a component of the pathname. |

**SEE ALSO**

pwd(1), malloc(3C), popen(3S).

## NAME

getenv — return value for environment name

## SYNOPSIS

```
char *getenv(name)
char *name;
```

## DESCRIPTION

getenv searches the environment list (see environ(5)) for a string of the form *name=value*, and returns a pointer to the *value* in the current environment if such a string is present; otherwise a NULL pointer is returned.

## SEE ALSO

exec(2), putenv(3C), environ(5).

**NAME**

getenv — return Fortran environment variable

**SYNOPSIS**

```
character *N c
getenv(tmpdir, c)
```

**DESCRIPTION**

getenv returns the character-string value of the environment variable represented by its first argument into the character variable of its second argument. If no such environment variable exists, all blanks are returned.

**SEE ALSO**

getenv(3C), environ(5).

## NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent — obtain group file entry from a group file

## SYNOPSIS

```
#include <grp.h>

struct group *getgrent()

struct group *getgrgid(gid)
gid_t gid;

struct group *getgrnam(name)
char *name;

void setgrent()

struct group *fgetgrent(f)
FILE *f;

void endgrent()
```

## DESCRIPTION

getgrent, getgrgid, and getgrnam return pointers to an object with the following structure containing the broken-out fields of a line in the /etc/group file. Each line contains a group structure, defined in the <grp.h> header file:

```
struct  group {
        char    *gr_name;    /* the name of the group */
        char    *gr_passwd;  /* the encrypted group
                                password */
        int       gr_gid;    /* the numeric 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; therefore, successive calls may be used to search the entire file. getgrgid searches from the beginning of the file until a numeric group ID matching *gid* is found; it returns a pointer to the particular structure in which the match was found. getgrnam searches from the beginning of the file until a group name matching *name* is found; it returns a pointer to the particular structure in which the match was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to `setgrent` has the effect of rewinding the group file to allow repeated searches. `endgrent` may be called to close the group file when processing is complete.

`fgetgrent` returns a pointer to the next group structure in the stream *f*, which matches the format of `/etc/group`.

**RETURN VALUE**

If an end-of-file or an error is encountered, a NULL pointer is returned.

**FILES**

`/etc/group`

**SEE ALSO**

`getlogin`(3C), `getpwent`(3C), `group`(4).

**WARNINGS**

The routines use `<stdio.h>`. Therefore, the size of programs not otherwise using standard I/O is increased more than might be expected.

**BUGS**

All information is contained in a static area, so it must be copied if it is to be saved.

## NAME

getgroups — get the group access list

## SYNOPSIS

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

int getgroups(gidsetlen, gidset)
int gidsetlen;
gid_t *gidset;
```

## DESCRIPTION

getgroups gets the current group access list of the user process and stores it in the array *gidset*. The parameter *gidsetlen* indicates the number of entries that may be placed in *gidset*.

getgroups returns the actual number of groups returned in *gidset*. No more than NGROUPS_MAX, as defined in <limits.h>, are ever returned. If *gidsetlen* is 0, getgroups returns the number of supplementary group IDs associated with the calling process without modifying the array pointed to by *gidset*.

## RETURN VALUE

A successful call returns the number of groups in the group set. If an error is detected, -1 is returned and the error code is stored in the global variable errno.

## ERRORS

The possible error values for getgroups are:

| | |
|---|---|
| [EINVAL] | The argument *gidsetlen* is smaller than the number of groups in the group set. |
| [EFAULT] | The argument *gidset* specifies an invalid address. |

## SEE ALSO

setgroups(2), initgroups(3X).

## NAME

gethostbyaddr, gethostbyname — get network host entry

## SYNOPSIS

```
#include <netdb.h>

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr;
int len, type;
```

## DESCRIPTION

gethostbyaddr and gethostbyname each return a pointer to an object with the following structure containing the broken-out fields of a line in the network host data base, /etc/hosts.

```
struct  hostent {
   char  *h_name;       /*official name of host*/
   char  **h_aliases;   /*alias list*/
   int   h_addrtype;    /*address type*/
   int   h_length;      /*length of address*/
   char  **h_addr_list; /*address list*/
};
#define h_addr h_addr_list[0] /* backward compatibility */
```

The members of this structure are

| | |
|---|---|
| h_name | official name of the host. |
| h_aliases | A zero terminated array of alternate names for the host. |
| h_addrtype | The type of address being returned; currently always AF_INET. |
| h_length | The length, in bytes, of the address. |
| h_addr | A pointer to the network address for the host. Host addresses are returned in network byte order. |

gethostbyname and gethostbyaddr sequentially search from the beginning of the file until a matching host name or host address is found, or until EOF is encountered. Host addresses are supplied in network order.

**RETURN VALUE**

NULL pointer (0) returned on EOF or error.

**FILES**

`/etc/hosts`

**SEE ALSO**

`hosts`(4N).

**BUGS**

All information is contained in a static area, so it must be copied if it is to be saved. Only the Internet address format is currently understood.

## NAME

getlogin — get login name

## SYNOPSIS

```
char *getlogin();
```

## DESCRIPTION

getlogin returns a pointer to the login name as found in /etc/utmp. It may be used in conjunction with 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 getlogin. If getlogin fails, call getpwuid.

## RETURN VALUE

getlogin returns the NULL pointer if *name* is not found.

## FILES

/etc/utmp

## SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

## BUGS

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

## NAME

setmntent, getmntent, addmntent, endmntent, hasmntopt — get file system descriptor file entry

## SYNOPSIS

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

FILE *setmntent(filep, type)
char *filep;
char *type;

struct mntent *getmntent(filep)
FILE *filep;

int addmntent(filep, mnt)
FILE *filep;
struct mntent *mnt;

char *hasmntopt(mnt, opt)
struct mntent *mnt;
char *opt;

int endmntent(filep)
FILE *filep;
```

## DESCRIPTION

These routines replace the getfsent(3) routines for accessing the file system description file /etc/fstab, and the mounted file system description file /etc/mtab.

setmntent opens a file system description file and returns a file pointer for use with getmntent, addmntent, or endmntent. The *type* argument is the same as in fopen(3). getmntent reads the next line from filep and returns a pointer to an object with the following structure containing broken-out fields of a line in the file system description file, <mntent.h>. The fields have meanings described in fstab(4).

```
struct mntent {
    char    *mnt_fsname; /* file system name */
    char    *mnt_dir;    /* file system path prefix */
    char    *mnt_type;   /* 4.2, 5.2, nfs, swap, or ignore */
    char    *mnt_opts;   /* ro, rw, quota, noquota, hard, soft */
    int     mnt_freq;    /* dump frequency, in days */
    int     mnt_passno;  /* pass number on parallel fsck */
};
```

`addmntent` adds the `mntent` structure *mnt* to the end of the open file *filep*. Note that *filep* has to be opened for writing if this is to work. `hasmntopt` scans the `mnt_opts` field of the `mntent` structure `mnt` for a substring that matches *opt*. It returns the address of the substring if a match is found, 0 otherwise. `endmntent` closes the file.

**RETURN VALUE**

NULL pointer (0) returned on EOF or error.

**FILES**

`/etc/fstab`
`/etc/mtab`

**SEE ALSO**

`fstab`(4), `mtab`(4).

**BUGS**

The returned `mntent` structure points to static information that is overwritten in each call.

## NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent — get network entry

## SYNOPSIS

```
#include <netdb.h>

struct netent *getnetent()

struct netent *getnetbyname(name)
char *name;

struct netent *getnetbyaddr(net)
long net;

setnetent(stayopen)
int stayopen;

endnetent()
```

## DESCRIPTION

getnetent, getnetbyname, and getnetbyaddr each return a pointer to an object with the following structure, containing the broken-out fields of a line in the network data base, /etc/networks.

```
struct netent {
        char    *n_name;      /* official name of net */
        char    **n_aliases;  /* alias list */
        int     n_addrtype;   /* net number type */
        long    n_net;        /* net number */
};
```

The members of this structure are:

| | |
|---|---|
| n_name | The official name of the network. |
| n_aliases | A zero terminated list of alternate names for the network. |
| n_addrtype | The type of the network number returned; currently only AF_INET. |
| n_net | The network number. Network numbers are returned in machine byte order. |

getnetent reads the next line of the file, opening the file if necessary.

setnetent opens and rewinds the file. If the *stayopen* flag is nonzero, the net data base will not be closed after each call to getnetent (either directly, or indirectly through one of the oth-

er "getnet" calls).

endnetent closes the file.

getnetbyname and getnetbyaddr sequentially search from the beginning of the file until a matching net name or net address is found, or until EOF is encountered. Network numbers are supplied in host order.

RETURN VALUE

NULL pointer (0) returned on EOF or error.

FILES

/etc/networks

SEE ALSO

networks(4N).

BUGS

All information is contained in a static area, so it must be copied if it is to be saved. Only Internet network numbers are currently understood. Expecting network numbers to fit in no more than 32 bits is probably naive.

## NAME

getnetgrent, setnetgrent, endnetgrent, innetgr — get network group entry

## SYNOPSIS

```
innetgr(netgroup, machine, user, domain)
char *netgroup, *machine, *user, *domain;

int setnetgrent(netgroup)
char *netgroup

int endnetgrent()

getnetgrent(machinep, userp, domainp)
char **machinep, **userp, **domainp;
```

## DESCRIPTION

inngetgr returns 1 or 0, depending on whether netgroup contains the machine, user, domain triple as a member. Any of the three strings machine, user, or domain can be NULL, in which case it signifies a wild card.

getnetgrent returns the next member of a network group. After the call, *machinep* will contain a pointer to a string containing the name of the machine part of the network group member, and similarly for *userp* and *domainp*. getnetgrent will malloc space for the name. This space is released when a endnetgrent call is made. getnetgrent returns 1 if it succeeding in obtaining another member of the network group, 0 if it has reached the end of the group.

setnetgrent establishes the network group from which getnetgrent will obtain members, and also restarts calls to getnetgrent from the beginning of the list. If the previous setnetgrent call was to a different network group, a endnetgrent call is implied. endnetgrent frees the space allocated during the getnetgrent calls.

## FILES

/etc/netgroup

## NAME

getopt — get option letter from argument vector

## SYNOPSIS

```
int getopt(argc, argv, optstring)
int argc;
char **argv, *optstring;

extern char *optarg;
extern int optind, opterr;
```

## DESCRIPTION

getopt returns the next option letter in *argv* that matches a letter in *optstring*. optstring is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. optarg is set to point to the start of the option argument on return from getopt.

getopt places in optind the *argv* index of the next argument to be processed. Because optind is external, it is normally initialized to zero automatically before the first call to getopt.

When all options have been processed (i.e., up to the first non-option argument), getopt returns EOF. The special option -- may be used to delimit the end of the options; EOF will be returned, and -- will be skipped.

## DIAGNOSTICS

getopt prints an error message on stderr and returns a question mark (?) when it encounters an option letter not included in *optstring*. This error message may be disabled by setting opterr to 0.

## EXAMPLES

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options a and b, and the options f and o, both of which require arguments:

```
main(argc, argv)
int argc;
char **argv;
{
   int c;
   extern int optind;
   extern char *optarg;
```

```
...

while ((c = getopt (argc, argv, "abf:o:")) != EOF)
   switch (c) {
   case 'a':
      if (bflg)
         errflg++;
      else
         aflg++;
      break;
   case 'b':
      if (aflg)
         errflg++;
      else
         bproc( );
      break;
   case 'f':
      ifile = optarg;
      break;
   case 'o':
      ofile = optarg;
      break;
   case '?':
      errflg++;
   }
if (errflg) {
   fprintf (stderr, "usage: ..fl. ");
   exit (2);
}
for ( ; optind < argc; optind++) {
   if (access (argv[optind], 4)) {

...
```

**SEE ALSO**

getopt(1).

## NAME

getpass — read a password

## SYNOPSIS

```
char *getpass (prompt)
char *prompt;
```

## DESCRIPTION

getpass reads up to a newline or EOF from the file /dev/tty, after prompting on the standard error output with the null-terminated string *prompt* and disabling echo. 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 terminates input and sends an interrupt signal to the calling program before returning.

## FILES

/dev/tty

## SEE ALSO

crypt(3C).

## WARNINGS

The above routine uses <stdio.h>. This causes the size of programs not otherwise using standard I/O to increase more than might be expected.

## BUGS

The return value points to static data whose content is overwritten by each call.

## NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent — get protocol entry

## SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

int setprotoent(stayopen)
int stayopen

int endprotoent()
```

## DESCRIPTION

getprotoent, getprotobyname, and getprotobynumber each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, /etc/protocols.

```
struct protoent {
        char *p_name;       /* official name of protocol */
        char **p_aliases;   /* alias list */
        long p_proto;       /* protocol number */
};
```

The members of this structure are:

| | |
|---|---|
| p_name | The official name of the protocol. |
| p_aliases | A zero terminated list of alternate names for the protocol. |
| p_proto | The protocol number. |

getprotoent reads the next line of the file, opening the file if necessary.

setprotoent opens and rewinds the file. If the *stayopen* flag is nonzero, the net data base will not be closed after each call to getprotoent (either directly, or indirectly through one of the other "getproto" calls).

`endprotoent` closes the file.

`getprotobyname` and `getprotobynumber` sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

**RETURN VALUE**

NULL pointer (0) returned on EOF or error.

**FILES**

`/etc/protocols`

**SEE ALSO**

`protocols`(4N).

**BUGS**

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

## NAME

getptabent, addptabent, endptabent, setptabent, numbptabent — get partition table file entry

## SYNOPSIS

```
#include <stdio.h>
#include <apple/ptabent.h>

struct ptabent *getptabent (filep)
FILE *filep;

int addptabent (filep, ptab)
FILE *filep;
struct ptabent *ptab;

int endptabent (filep)
FILE *filep;

FILE *setptabent (fname, type)
char *fname;
char *type;

int numptabent (filep)
FILE *filep;

cc [flags] files -lptab [libraries]
```

## DESCRIPTION

`setptabent` opens a partition table file and returns a file pointer which can then be used with `getptabent` or `addptabent`. The *type* argument is the same as in `fopen`(3). `getptabent` returns a pointer to an object with the following structure containing the broken-out fields of a line in the partition table file. The fields have meanings described in `ptab`(4).

```
struct ptabent {
        char *ptab_name;     /* partition name */
        char *ptab_type;     /* partition type */
        int  ptab_ctrl;      /* controller number */
        int  ptab_disk;      /* disk number */
        int  ptab_part;      /* partition number */
};
```

`addptabent` adds the `ptabent` structure `ptab` to the end of the open file *filep*. `numptabent` returns the number of partition table file entries and has the effect of rewinding the partition table file to allow repeated searches. `endptabtent` closes the file.

**FILES**

/etc/ptab

**RETURN VALUE**

A NULL pointer (0) is returned on EOF or error by setptabent and getptabent. addptabent, endptabent, and numbptabent return EOF on error.

**BUGS**

The returned ptabent structure points to static information that is overwritten in each call.

**SEE ALSO**

pname(1M), ptab(4).

## NAME

getpw — get name from UID

## SYNOPSIS

```
int getpw(uid, buf)
int uid;
char *buf;
```

## DESCRIPTION

getpw searches the password file for a user ID number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. The line is null terminated. getpw returns nonzero if *uid* cannot be found.

This routine is included only for compatibility with prior systems and should not be used; see getpwent(3C) for routines to use instead.

## RETURN VALUE

getpw returns nonzero on error.

## FILES

/etc/passwd

## SEE ALSO

getpwent(3C), passwd(4).

## WARNINGS

The above routine uses <stdio.h>. Therefore, the size of programs not otherwise using standard I/O is increased more than might be expected.

## NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent — get the password file entry

## SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent()

struct passwd *getpwuid(uid)
uid_t uid;

struct passwd *getpwnam(name)
char *name;

void setpwent()

void endpwent()

struct passwd *fgetpwent(f)
FILE *f;
```

## DESCRIPTION

getpwent, getpwuid, and getpwnam 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;
};
```

Because this structure is declared in <pwd.h>, it is not necessary to redeclare it.

The pw_comment field is unused; the others have meanings described in passwd(4).

When first called, `getpwent` returns a pointer to the first `passwd` structure in the file. Thereafter, it returns a pointer to the next `passwd` structure in the file; therefore, successive calls can be used to search the entire file. `getpwuid` searches from the beginning of the file until a numeric user ID matching *uid* is found; it returns a pointer to the particular structure in which the match was found. `getpwnam` searches from the beginning of the file until a login name matching *name* is found; it returns a pointer to the particular structure in which the match 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.

`fgetpwent` returns a pointer to the next `passwd` structure in the stream *f*, which matches the format of `/etc/passwd`.

## RETURN VALUE

If an end-of-file or an error is encountered, a NULL pointer is returned.

## FILES

`/etc/passwd`

## SEE ALSO

`cuserid`(3S), `getlogin`(3C), `getgrent`(3C), `putpwent`(3C), `passwd`(4).

## WARNINGS

The routines use `<stdio.h>`. Therefore the size of programs not otherwise using standard I/O is increased more than might be expected.

## BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

## NAME

endrpcent, getrpcent, getrpcbyname, getrpcbynumber, setrpcent — get RPC entry

## SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcent()

struct rpcent *getrpcbyname(name)
char *name;

struct rpcent *getrpcbynumber(number)
int number;

setrpcent(stayopen)
int stayopen;

int endrpcent()
```

## DESCRIPTION

getrpcent, getrpcbyname, and getrpcbynumber each return a pointer to an object with the following structure containing the broken-out fields of a line in the RPC program number data base, /etc/rpc.

```
struct rpcent {
    char *r_name;      /* name of server */
    char **r_aliases; /* alias list */
    long r_number;     /* rpc program number */
};
```

The members of this structure are

| | |
|---|---|
| r_name | The name of the server for this RPC program. |
| r_aliases | A zero terminated list of alternate names for the RPC program. |
| r_number | The RPC program number for this service. |

getrpcent reads the next line of the file, opening the file if necessary.

setrpcent opens and rewinds the file. If the *stayopen* flag is nonzero, the net data base will not be closed after each call to getrpcent (neither directly nor indirectly through one of the other getrpc calls).

`endrpcent` closes the file.

`getrpcbyname` and `getrpcbynumber` sequentially search from the beginning of the file until a matching RPC program name or program number is found or until EOF is encountered.

**FILES**

`/etc/rpc`
`/etc/yp/`*domainname*`/rpc.bynumber`

**SEE ALSO**

`rpcinfo`(1M).

**DIAGNOSTICS**

Null pointer (0) returned on EOF or error.

**BUGS**

All information is contained in a static area, so it must be copied if it is to be saved.

## NAME

getrpcport — get RPC port number

## SYNOPSIS

```
int getrpcport(host, prognum, versnum, proto)
char *host;
int prognum, versnum, proto;
```

## DESCRIPTION

getrpcport returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact the portmapper, or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it will return that port number.

## NAME

gets, fgets — get a string from a stream

## SYNOPSIS

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;
```

## DESCRIPTION

gets reads characters from the standard input stream, stdin, into the array pointed to by *s*, until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

fgets reads characters from the *stream* into the array pointed to by *s* until *n*–1 characters are read, or a newline character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

## RETURN VALUE

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 (for example, trying to use these functions on a file that has not been opened for reading) occurs, a NULL pointer is returned. Otherwise *s* is returned.

## SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S).

## NOTES

gets deletes the newline ending its input, but fgets keeps it.

## NAME

getservent, getservbyport, getservbyname, setservent, endservent — get service entry

## SYNOPSIS

```
#include <netdb.h>

struct servent *getservent()

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port;
char *proto;

int setservent(stayopen)
int stayopen;

int endservent()
```

## DESCRIPTION

`getservent`, `getservbyname`, and `getservbyport` each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, `/etc/services`.

```
struct servent {
   char *s_name;     /* official name of service */
   char **s_aliases; /* alias list */
   long s_port;      /* port service resides at */
   char *s_proto;    /* protocol to use */
};
```

The members of this structure are:

| | |
|---|---|
| `s_name` | The official name of the service. |
| `s_aliases` | A zero terminated list of alternate names for the service. |
| `s_port` | The port number at which the service resides. Port numbers are returned in network byte order. |
| `s_proto` | The name of the protocol to use when contacting the service. |

`getservent` reads the next line of the file, opening the file if necessary.

`setservent` opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getservent` (either directly, or indirectly through one of the other `getserv` calls).

`endservent` closes the file.

`getservbyname` and `getservbyport` sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

**RETURN VALUE**

NULL pointer (0) returned on EOF or error.

**FILES**

`/etc/services`

**SEE ALSO**

`getprotoent`(3N), `services`(4N).

**BUGS**

All information is contained in a static area, so it must be copied if it is to be saved. Expecting port numbers to fit in a 32 bit quantity is probably naive.

## NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname — access utmp file entry

## SYNOPSIS

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

struct utmp *getutent()

struct utmp *getutid(id)
struct utmp *id;

struct utmp *getutline(line)
struct utmp *line;

void pututline(utmp)
struct utmp *utmp;

void setutent()

void endutent()

void utmpname(file)
char *file;
```

## DESCRIPTION

getutent, getutid, and getutline each return a pointer to a structure of the following type:

```
struct utmp {
     char   ut_user[8];     /* User login name */
     char   ut_id[4];       /* /etc/inittab ID
                               (usually line#) */
     char   ut_line[12];    /* device name (console, lnxx) */
     short  ut_pid;         /* process ID */
     short  ut_type;        /* type of entry */
     struct exit_status {
     short  e_termination; /* Process termination status */
     short  e_exit;         /* Process exit status */
     } ut_exit;             /* Exit status of a process
                            /* marked as DEAD_PROCESS */
     time_t  ut_time;       /* time entry was made */
     char    ut_host[16];   /* host name, if remote */
};
```

getutent reads in the next entry from a utmp-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

`getutid` searches forward from the current point in the `utmp` file until it finds an entry with a `ut_type` matching `id->ut_type` if the type specified is `RUN_LVL`, `BOOT_TIME`, `OLD_TIME`, or `NEW_TIME`. If the type specified in *id* is `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS`, or `DEAD_PROCESS`, `getutid` will return a pointer to the first entry whose type is one of these four and whose `ut_id` field matches `id->ut_id`. `getutid` fails if the end of file is reached without a match.

`getutline` searches forward from the current point in the `utmp` file until it finds an entry of the type `LOGIN_PROCESS` or `USER_PROCESS` which also has a `ut_line` string matching the `line->ut_line` string. If the end of file is reached without a match, it fails.

`pututline` writes out the supplied `utmp` structure into the `utmp` file. It uses `getutid` to search forward for the proper place if it finds that it is not already at the proper place. It is assumed that the user of `pututline` has searched for the proper entry using one of the `getut` routines. If this has been done, `pututline` will not search. If `pututline` does not find a matching slot for the new entry, it will add a new entry to the end of the file.

`setutent` resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

`endutent` closes the currently open file.

`utmpname` allows the user to change the name of the file examined from `/etc/utmp` to any other filename. It is expected that most often this other file will be `/etc/wtmp`. If the file doesn't exist, this will not be apparent until the first attempt to reference the file is made. `utmpname` does not open the file. It just closes the old file, if it is currently open, and saves the new filename.

## RETURN VALUE

A NULL pointer is returned upon failure to read or write. Failure to read may be due to permissions or because end-of-file has been reached.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

COMMENTS

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either getutid or getutline sees the routine examine the static structure before performing more I/O. If the search of the static structure results in a match, no further search is performed. To use getutline to search for multiple occurences, zero out the static structure after each success; otherwise getutline will 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. If the implicit read done by pututline finds that it isn't already at the correct place in the file, the contents of the static structure returned by the getutent, getutid, or getutline routines are not harmed, if the user has just modified those contents and passed the pointer back to pututline.

These routines use buffered standard I/O for input, but pututline uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the utmp and wtmp files.

## NAME

getwd — get current working directory pathname

## SYNOPSIS

```
char *getwd(pathname)
char *pathname;
```

## DESCRIPTION

getwd copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result.

Maximum pathname length is PATH_MAX characters (see intro(2)).

## DIAGNOSTICS

getwd returns zero and places a message in *pathname* if an error occurs.

## NAME

hsearch, hcreate, hdestroy — manage hash search tables

## SYNOPSIS

```
#include <search.h>

ENTRY *hsearch(item, action)
ENTRY item;
ACTION action;

int hcreate(nel)
unsigned nel;

void hdestroy()
```

## DESCRIPTION

hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *item* is a structure of type ENTRY (defined in the <search.h> header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. ENTER indicates that the item should be inserted in the table at an appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

hcreate allocates sufficient space for the table and must be called before hsearch is used. *nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

hdestroy destroys the search table, and may be followed by another call to hcreate.

## NOTES

hsearch uses "open addressing" with a "multiplicative" hash function. However, its source code has many other options available which the user may select by compiling the hsearch source with the following symbols defined to the preprocessor:

DIV — Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.

USCR — Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named `hcompar` and should behave in a manner similar to `strcmp` (see `string`(3C)).

CHAINED — Use a linked list to resolve collisions. If this option is selected, the following other options become available.

- START — Place new entries at the beginning of the linked list (default is at the end).
- SORTUP — Keep the linked list sorted by key in ascending order.
- SORTDOWN — Keep the linked list sorted by key in descending order.

Additionally, there are preprocessor flags for obtaining debugging printout (`-DDEBUG`) and for including a test driver in the calling routine (`-DDRIVER`). The source code should be consulted for further details.

## RETURN VALUE

`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.

## EXAMPLES

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

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

struct info {     /* this is the info stored in
  int age, room;     the table other than the key */
};
#define NUM_EMPL 5000 /* # of elements in search
                         table */
```

```
main( )
{
    /* space to store strings */
    char string_space[NUM_EMPL*20];

    /* space to store employee info */
    struct info info_space[NUM_EMPL];

    /* next avail space in string_space */
    char *str_ptr = string_space;

    /* next avail space in info_space */
    struct info *info_ptr = info_space;
    ENTRY item, *found_item, *hsearch( );

    /* name to look for in table */
    char name_to_find[30];
    int i = 0;

    /* create table */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
       &info_ptr->room) != EOF && i++ < NUM_EMPL) {

      /* put info in structure,
         and structure in item */
      item.key = str_ptr;
      item.data = (char *)info_ptr;
      str_ptr += strlen(str_ptr) + 1;
      info_ptr++;

      /* put item into table */
      (void) hsearch(item, ENTER);
 }

 /* access table */
 item.key = name_to_find;
 while (scanf("%s", item.key) != EOF) {
    if ((found_item = hsearch(item, FIND)) != NULL) {

      /* if item is in the table */
      (void)printf("found %s, age = %d, room = %d\n",
          found_item->key,
         ((struct info *)found_item->data)->age,
         ((struct info *)found_item->data)->room);
     } else {
      (void)printf("no such employee %s\n",
          name_to_find)
     }
 }
}
```

**SEE ALSO**

bsearch(3C), lsearch(3C), malloc(3C), malloc(3X), string(3C), tsearch(3C).

**WARNINGS**

hsearch and hcreate use malloc(3C) to allocate space.

**BUGS**

Only one hash search table may be active at any given time.

## NAME

hypot — Euclidean distance function

## SYNOPSIS

```
#include <math.h>

double hypot(x, y)
double x, y;
```

## DESCRIPTION

hypot returns the following, taking precautions against unwarranted overflows:

```
sqrt(x * x + y * y)
```

## RETURN VALUE

When the correct value would overflow, hypot returns HUGE and sets errno to ERANGE.

These error-handling procedures may be changed with the function matherr(3M).

## SEE ALSO

matherr(3M).

**NAME**

iargc — return command line arguments

**SYNOPSIS**

```
integer i
i=iargc()
```

**DESCRIPTION**

The iargc function returns the number of command line arguments passed to the program. Thus, if a program were invoked via

foo *arg1 arg2 arg3*

iargc() would return "3".

**SEE ALSO**

getarg(3F).

## NAME

index — return location of Fortran substring

## SYNOPSIS

```
character *N1 ch1
character *N2 ch2
integer i
i=index(ch1, ch2)
```

## DESCRIPTION

index returns the location of substring *ch2* in string *ch1*. The value returned is either the position at which substring *ch2* starts or 0 if *ch2* is not present in string *ch1*.

## NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof — Internet address manipulation routines

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(cp)
char *cp;

unsigned long inet_network(cp)
char *cp;

char *inet_ntoa(in)
struct in_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

int inet_lnaof(in)
struct in_addr in;

int inet_netof(in)
struct in_addr in;
```

## DESCRIPTION

The routines inet_addr and inet_network each interpret character strings representing numbers expressed in the Internet standard . notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine inet_ntoa takes an Internet address and returns an ASCII string representing the address in . notation. The routine inet_makeaddr takes an Internet network number and a local network address and constructs an Internet address from it. The routines inet_netof and inet_lnaof break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet address are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

## INTERNET ADDRESSES

Values specified using the . notation take one of the following forms.

*a.b.c.d*
*a.b.c*
*a.b*
*a*

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as `128.net.host`.

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as `net.host`.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a . notation may be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

## RETURN VALUE

The value −1 is returned by `inet_addr` and `inet_network` for malformed requests.

## SEE ALSO

`getnetent`(3N), `hosts`(4N), `networks`(4N).

## BUGS

The problem of host byte ordering versus network byte ordering is confusing. A simple way to specify Class C network addresses in a manner similar to Class B and Class A is needed. The string returned by `inet_ntoa` resides in a static memory area.

## NAME

initgroups — initialize group access list

## SYNOPSIS

```
initgroups(name, basegid)
char * name;
int basegid;
```

## DESCRIPTION

initgroups reads through the group file and sets up, using the setgroups(2) call, the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file.

## RETURN VALUE

initgroups returns –1 if it was not invoked by the superuser.

## FILES

/etc/group
/etc/passwd

## SEE ALSO

setgroups(2).

## BUGS

initgroups uses the routines based on getgrent(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to initgroups.

## NAME

insque, remque — insert/remove element from a queue

## SYNOPSIS

```
#include <vax/vaxque.h>

int insque(elem,pred)
struct qelem *elem, *pred;

int remque(elem)
struct qelem *elem;
```

## DESCRIPTION

The insque and remque macros manipulate queues built from doubly-linked lists. Each element in the queue must be in the form of struct qelem.

```
struct qelem {
        struct   qelem *q_forw;
        struct   qelem *q_back;
        char     q_data[];
};
```

insque inserts *elem* in a queue immediately after *pred*; remque removes an entry *elem* from a queue.

## FILES

/usr/include/vax/vaxque.h

## NAME

killpg — send signal to a process group

## SYNOPSIS

```
int killpg(pgrp, sig)
int pgrp, sig;
```

## DESCRIPTION

killpg sends the signal *sig* to the process group *pgrp*.

The sending process and members of the process group must have the same effective user ID, otherwise this call is restricted to the superuser. As a single special case the continue signal SIGCONT may be sent to any process which is a descendant of the current process.

## RETURN VALUE

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

## ERRORS

killpg will fail and no signal will be sent if any of the following occur:

| | |
|---|---|
| [EINVAL] | *sig* is not a valid signal number. |
| [ESRCH] | No process can be found corresponding to that specified by *pgrp*. |
| [EPERM] | The sending process is not the superuser and one or more of the target processes has an effective user ID different from that of the sending process. |

## SEE ALSO

kill(2), getpid(2).

## NAME

l3tol, ltol3 — convert between 3-byte integers and long integers

## SYNOPSIS

```
void l3tol(lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3(cp, lp, n)
char *cp;
long *lp;
int n;
```

## DESCRIPTION

l3tol converts a list of *n* 3-byte integers (packed into a character string pointed to by *cp*) into a list of long integers pointed to by lp.

ltol3 performs the reverse conversion from long integers (*lp*) to 3-byte integers (*cp*).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

## SEE ALSO

fs(4).

## BUGS

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

## NAME

lap_default — AppleTalk Link Access Protocol (LLAP/ELAP) interface

## SYNOPSIS

```
char *lap_default()
```

## DESCRIPTION

The lap_default routine returns a character pointer to the LAP interface name of the default interface as defined in /etc/appletalkrc. It returns NULL on error.

## ERRORS

If an error occurs, lap_default returns NULL, with a detailed error code in errno.

[ENOENT] No AppleTalk interface exists.

## FILES

/dev/appletalk/lap/*/...
/etc/appletalkrc

## SEE ALSO

atp(3N), ddp(3N), nbp(3N), pap(3N), rtmp(3N), appletalkrc(4), appletalk(7); *Inside AppleTalk;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming.*

## NAME

ldahread — read the archive header of a member of an archive file

## SYNOPSIS

```
#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, ldahread reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

Programs using this routine should be loaded with the object file access library libld.a.

## RETURN VALUE

ldahread returns SUCCESS or FAILURE. ldahread fails if TYPE(*ldptr*) does not represent an archive file or if it cannot read the archive header.

## SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(3X), ar(4).

## NAME

ldclose, ldaclose — close a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldclose(ldptr)
LDFILE *ldptr;

int ldaclose(ldptr)
LDFILE *ldptr;
```

## DESCRIPTION

ldopen(3X) and ldclose 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 closes the file and frees 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 reinitializes OFFSET(*ldptr*) to the file address of the next archive member and returns FAILURE. The LDFILE structure is prepared for a subsequent ldopen(3X). 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.

Programs using this routine must be loaded with the object file access library libld.a.

## SEE ALSO

fclose(3S), ldfcn(3X), ldopen(3X).

## NAME

ldfcn — common object file access routines

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>
```

## DESCRIPTION

The common object file access routines are a collection of functions for reading an object file that is in common object file form. Although the calling program must know the detailed structure of the parts of the object file that it processes, the routines effectively insulate the calling program from knowledge of the overall structure of the object file.

The interface between the calling program and the object file access routines is based on the defined type LDFILE (defined as struct ldfile), which is declared in the header file <ldfcn.h>. The primary purpose of this structure is to provide uniform access to both simple object files and object files that are members of an archive file.

The function ldopen(3X) allocates and initializes the LDFILE structure and returns a pointer to the structure to the calling program. The fields of the LDFILE structure may be accessed individually through macros defined in <ldfcn.h> and contain the following information:

LDFILE **ldptr;*

TYPE(*ldptr*) The file magic number, used to distinguish between archive members and simple object files.

IOPTR(*ldptr*) The file pointer returned by fopen(3S) and used by the standard input/output functions.

OFFSET(*ldptr*) The file address of the beginning of the object file; the offset is nonzero if the object file is a member of an archive file.

HEADER(*ldptr*) The file header structure of the object file.

The object file access functions may be divided into four categories:

(1) functions that open or close an object file

`ldopen`(3X) and `ldaopen`
: open a common object file

`ldclose`(3X) and `ldaclose`
: close a common object file

(2) functions that read header or symbol table information

`ldahread`(3X)
: read the archive header of a member of an archive file

`ldfhread`(3X)
: read the file header of a common object file

`ldshread`(3X) and `ldnshread`
: read a section header of a common object file

`ldtbread`(3X)
: read a symbol table entry of a common object file

`ldgetname`(3X)
: retrieve a symbol name from a symbol table entry or from the string table

(3) functions that position an object file at (seek to) the start of the section, relocation, or line number information for a particular section.

`ldohseek`(3X)
: seek to the optional file header of a common object file

`ldsseek`(3X) and `ldnsseek`
: seek to a section of a common object file

`ldrseek`(3X) and `ldnrseek`
: seek to the relocation information for a section of a common object file

`ldlseek`(3X) and `ldnlseek`
: seek to the line number information for a section of a common object file

`ldtbseek`(3X)
: seek to the symbol table of a common object file

(4) the function `ldtbindex`(3X) which returns the index of a particular common object file symbol table entry

These functions are described in detail in the manual pages identified for each function.

All the functions except ldopen, ldgetname(3X), ldaopen, and ldtbindex return either SUCCESS or FAILURE, which are constants defined in <ldfcn.h>. ldopen and ldaopen both return pointers to a LDFILE structure.

Programs using this routine must be loaded with the object file access library libld.a.

## MACROS

Additional access to an object file is provided through a set of macros defined in <ldfcn.h>. These macros parallel the standard input/output file reading and manipulating functions, translating a reference of the LDFILE structure into a reference to its file descriptor field.

The following macros are provided:

GETC (*ldptr*)
FGETC (*ldptr*)
GETW (*ldptr*)
UNGETC (*c, ldptr*)
FGETS (*s, n, ldptr*)
FREAD (*ptr, size, nitems, ldptr*)
FSEEK (*ldptr, offset, ptrname*)
FTELL (*ldptr*)
REWIND (*ldptr*)
FEOF (*ldptr*)
FERROR (*ldptr*)
FILENO (*ldptr*)
SETBUF (*ldptr, buf*)
STROFFSET (*ldptr*)

The STROFFSET macro calculates the address of the string table in an object file. See the manual entries for the corresponding standard input/output library functions for details on the use of these macros. (The functions are identified as 3S in this manual.)

## WARNINGS

The macro FSEEK defined in the header file <ldfcn.h> translates into a call to the standard input/output function fseek(3S). FSEEK should not be used to seek from the end of an archive file since the end of an archive file may not be the same as the end of one of its object file members.

**SEE ALSO**

fopen(3S), fseek(3S), ldahread(3X), ldclose(3X), ldfhread(3X), ldgetname(3X), ldlread(3X), ldlseek(3X), ldohseek(3X), ldopen(3X), ldrseek(3X), ldlseek(3X), ldshread(3X), ldtbindex(3X), ldtbread(3X), ldtbseek(3X).
"COFF Reference" and "C Object Library" *A/UX Programming Languages and Tools, Volume 1.*

## NAME

ldfhread — read the file header of a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread(ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

## DESCRIPTION

ldfhread 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. ldfhread fails 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(3)). The information in any field of the file header may be accessed by applying the dot operator to the value returned by the HEADER macro; for example:

```
HEADER(ldptr).f_timdat
```

The program using this routine must be loaded with the object file access library libld.a.

## SEE ALSO

ldclose(3X), ldfcn(3X), ldopen(3X), filehdr(4).

## NAME

ldgetname — retrieve symbol name for object file symbol table entry

## SYNOPSIS

```
#include <stdio.h> #include <filehdr.h>
#include <syms.h> #include <ldfcn.h>

char *ldgetname(ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;
```

## DESCRIPTION

ldgetname returns a pointer to the name associated with *symbol* as a string. The string is contained in a static buffer local to ldgetname. Because the buffer is overwritten by each call to ldgetname, it must be copied by the caller if the name is to be saved.

The common object file format has been extended to handle arbitrary length symbol names with the addition of a "string table." ldgetname returns the symbol name associated with a symbol table entry for either an object file or a preobject file. Thus, ldgetname can be used to retrieve names from object files without any backward compatibility problems.

Typically, ldgetname is called immediately after a successful call to ldtbread to retrieve the name associated with the symbol table entry filled by ldtbread.

Programs using this routine should be loaded with the object file access library libld.a.

## ERRORS

ldgetname returns NULL (defined in <stdio.h>) for an object file if the name cannot be retrieved. This occurs when:

the string table cannot be found.

not enough memory can be allocated for the string table.

the string table appears not to be a string table (e.g., if an auxiliary entry is handed to ldgetname that looks like a reference to a name in a nonexistent string table).

the name's offset into the string table is beyond the end of the string table.

**SEE ALSO**

ldclose(3X), ldfcn(3X), ldopen(3X), ldtbseek(3X), ldtbread(3X).

## NAME

ldlread, ldlinit, ldlitem — manipulate line number entries of a common object file function

## SYNOPSIS

```
#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

ldlread 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*.

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* into *linent*.

Programs using this routine should be loaded with the object file access library libld.a.

**ERRORS**

`ldlread`, `ldlinit`, and `ldlitem` each return either `SUCCESS` or `FAILURE`. `ldlread` fails 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` fails 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` fails if it finds no line number equal to or greater than *linenum*.

**SEE ALSO**

`ldclose`(3X), `ldfcn`(3X), `ldopen`(3X), `ldtbindex`(3X).

## NAME

ldlseek, ldnlseek — seek to line number entries of a section of a common object file

## SYNOPSIS

```
#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

ldlseek 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 fails if *sectindx* is greater than the number of sections in the object file; ldnlseek fails if there is no section name corresponding to **sectname*. Either function fails 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.

Programs using this routine must be loaded with the object file access library libld.a.

## SEE ALSO

ldclose(3X), ldfcn(3X), ldopen(3X), ldshread(3X).

## NAME

ldohseek — seek to the optional file header of a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek(ldptr)
LDFILE *ldptr;
```

## DESCRIPTION

ldohseek seeks to the optional file header of the common object file currently associated with *ldptr*.

ldohseek returns SUCCESS or FAILURE. ldohseek fails if the object file has no optional header or if it cannot seek to the optional header.

Programs using this routine should be loaded with the object file access routine library libld.a.

## SEE ALSO

ldclose(3X), ldfcn(3X), ldopen(3X), ldfhread(3X).

## NAME

ldopen, ldaopen — open a common object file for reading

## SYNOPSIS

```
#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

ldopen and ldclose(3X) 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, ldopen opens *filename*, allocates and initializes the LDFILE structure, and returns a pointer to the structure to the calling program.

If *ldptr* is valid and TYPE(*ldptr*) is the archive magic number, ldopen reinitializes the LDFILE structure for the next archive member of *filename*.

ldopen and ldclose are designed to work in concert. ldclose returns 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.

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 opens *filename* anew and allocates and initializes 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.

Programs using this routine must be loaded with the object file access library libld.a.

**SEE ALSO**

fopen(3S), ldclose(3X), ldfcn(3X).

## NAME

ldrseek, ldnrseek — seek to relocation entries of a section of a common object file

## SYNOPSIS

```
#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

ldrseek 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*.

The routines ldrseek and ldnrseek return SUCCESS or FAILURE. ldrseek fails if *sectindx* is greater than the number of sections in the object file; ldnrseek fails if there is no section name corresponding with *sectname*. Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note that the first section has an index of one.

Programs using this routine should be loaded with the object file access library libld.a.

## SEE ALSO

ldclose(3X), ldfcn(3X), ldopen(3X), ldshread(3X).

## NAME

ldshread, ldnshread — read an indexed/named section header of a common object file

## SYNOPSIS

```
#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

ldshread 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 fails if *sectindx* is greater than the number of sections in the object file; ldnshread fails if there is no section name corresponding with *sectname*. Either function fails if it cannot read the specified section header.

Note that the first section header has an index of one.

Programs using this routine must be loaded with the object file access library libld.a.

## SEE ALSO

ldclose(3X), ldfcn(3X), ldopen(3X).

## NAME

ldsseek, ldnsseek — seek to an indexed/named section of a common object file

## SYNOPSIS

```
#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

ldsseek seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

ldnsseek seeks to the section specified by *sectname*.

ldsseek and ldnsseek return SUCCESS or FAILURE. ldsseek fails if *sectindx* is greater than the number of sections in the object file; ldnsseek fails if there is no section name corresponding with *sectname*. Either function fails 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.

Programs using this routine should be loaded with the object file access library libld.a.

## SEE ALSO

ldclose(3X), ldfcn(3X), ldopen(3X), ldshread(3X).

## NAME

ldtbindex — compute index of a symbol table entry of a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex(ldptr)
LDFILE *ldptr;
```

## DESCRIPTION

ldtbindex 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 ldtbread(3X). 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 particular symbol table entry has been read, it returns the the index of the next entry.

ldtbindex fails 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.

Programs using this routine should be loaded with the object file access library libld.a.

## SEE ALSO

ldclose(3X), ldfcn(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X).

## NAME

ldtbread — read an indexed symbol table entry of a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread(ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

## DESCRIPTION

ldtbread 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. ldtbread fails 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.

Programs using this routine must be loaded with the object file access library libld.a.

## SEE ALSO

ldclose(3X), ldfcn(3X), ldgetname(3X), ldopen(3X), ldtbseek(3X).

## NAME

ldtbseek — seek to the symbol table of a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek(ldptr)
LDFILE *ldptr;
```

## DESCRIPTION

ldtbseek seeks to the symbol table of the common object file currently associated with *ldptr*.

ldtbseek returns SUCCESS or FAILURE. ldtbseek fails if the symbol table has been stripped from the object file or if it cannot seek to the symbol table.

Programs using this routine must be loaded with the object file access library libld.a.

## SEE ALSO

ldclose(3X), ldfcn(3X), ldopen(3X), ldtbread(3X).

**NAME**

len — return length of Fortran string

**SYNOPSIS**

```
character *N ch
integer i
i=len(ch)
```

**DESCRIPTION**

len returns the length of string *ch*.

## NAME

lge, lgt, lle, llt — string comparision intrinsic functions

## SYNOPSIS

```
character *N a1, a2
logical l

l=lge(a1, a2)
l=lgt(a1, a2)
l=lle(a1, a2)
l=llt(a1, a2)
```

## DESCRIPTION

These functions return TRUE if the inequality holds and FALSE otherwise.

**NAME**

line_push — routine used to push streams line disciplines

**SYNOPSIS**

```
line_push (fildes)
int fildes;
```

**DESCRIPTION**

line_push will push the streams line discipline "line" onto the stream referenced by the file descriptor *fildes*. If *fildes* does not reference a stream or it references a stream that already has a line discipline pushed onto it nothing will happen.

**SEE ALSO**

line_sane(1M), streams(7).

## NAME

lockf — record locking on files

## SYNOPSIS

```
#include <unistd.h>

int lockf(fildes, function, size)
long size;
int fildes, function;
```

## DESCRIPTION

The lockf call will allow sections of a file to be locked (advisory write locks). (Mandatory locking is available via locking(2)). Locking calls from other processes which attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. (See fcntl(2) for more information about record locking.)

*fildes* is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish lock with this function call.

*function* is a control value which specifies the action to be taken. The permissible values for *function* are defined in <unistd.h> 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.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

*size* is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size. If *size* is zero, the section from the current offset through the largest file offset is locked (i.e., from the current offset through the present or any future 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 [EACCES] error if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to 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.

Sleeping on a resource is interrupted with any signal. The alarm(2) command may be used to provide a timeout facility in applications which require this facility.

## RETURN VALUE

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

## ERRORS

The `lockf` utility will fail if one or more of the following are true:

| | |
|---|---|
| `[EBADF]` | *fildes* is not a valid open descriptor. |
| `[EACCES]` | `function` is `F_TLOCK` or `F_TEST` and the section is already locked by another process. |
| `[EDEADLK]` | `function` is `F_LOCK` or `F_TLOCK` and a deadlock would occur. Also the `function` is either of the above or `F_ULOCK` and the number of entries in the lock table would exceed the number allocated on the system. |
| `[EREMOTE]` | *fildes* is a file descriptor referring to a file on a remotely mounted file system. |

## CAVEATS

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The standard I/O package is the most common source of unexpected buffering.

## SEE ALSO

`close`(2), `creat`(2), `fcntl`(2), `intro`(2), `locking`(2), `open`(2), `read`(2), `write`(2).

## NAME

log, alog, dlog, clog — Fortran natural logarithm intrinsic function

## SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2=alog(r1)
r2=log(r1)

dp2=dlog(dp1)
dp2=log(dp1)

cx2=clog(cx1)
cx2=log(cx1)
```

## DESCRIPTION

alog returns the real natural logarithm of its real argument. dlog returns the double-precision natural logarithm of its double-precision argument. clog returns the complex logarithm of its complex argument. The generic function log becomes a call to alog, dlog, or clog depending on the type of its argument.

## SEE ALSO

exp(3M).

## NAME

log10, alog10, dlog10 — Fortran common logarithm intrinsic function

## SYNOPSIS

```
real r1,r2
double precision dp1, dp2

r2=alog10(r1)
r2=log10(r1)

dp2=dlog10(dp1)
dp2=log10(dp1)
```

## DESCRIPTION

alog10 returns the real common logarithm of its real argument. dlog10 returns the double-precision common logarithm of its double-precision argument. The generic function log10 becomes a call to alog10 or dlog10 depending on the type of its argument.

## SEE ALSO

exp(3M).

## NAME

logname — return login name of user

## SYNOPSIS

```
char *logname()
```

## DESCRIPTION

logname returns a pointer to the null-terminated login name; it extracts the $LOGNAME variable from the user's environment.

This routine is kept in /lib/libPW.a.

## FILES

/etc/profile

## SEE ALSO

env(1), login(1), profile(4), environ(5).

## BUGS

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

This method of determining a login name is subject to forgery.

## NAME

lsearch, lfind — linear search and update

## SYNOPSIS

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

char *lsearch(key, base, nelp, width, compar)
char *key;
char *base;
unsigned *nelp;
unsigned *width;
int (*compar)();

char *lfind(key, base, nelp, width, compar)
char *key;
char *base;
unsigned *nelp;
unsigned *width;
int (*compar)();
```

## DESCRIPTION

lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. *key* points to the datum to be sought in the table. *base* points to the first element in the table. *nelp* points to an integer containing the current number of elements in the table. The integer at **nelp* is incremented if the datum is added to the table. *width* is the width of an element in bytes. *compar* is the name of the comparison function which the user must supply (strcmp, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

lfind is the same as lsearch except that if the datum is not found, it is not added to the table. Instead, a −1 pointer is returned.

## RETURN VALUE

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.

## NOTES

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

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

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

## EXAMPLES

This fragment will read in ≤ TABSIZE strings of length ≤ ELSIZE and store them in a table, eliminating duplicates.

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

#define TABSIZE 50
#define ELSIZE 120

    char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
    unsigned nel = 0;
    int strcmp( );
    . . .
    while (fgets(line, ELSIZE, stdin) != NULL &&
          nel < TABSIZE)
             (void) lsearch(line, (char *)tab, &nel,
                       ELSIZE, strcmp);
    . . .
```

## SEE ALSO

bsearch(3C), hsearch(3C), tsearch(3C).

## BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and troff running on A/UX. Proof and final pages were created on Apple LaserWriter® printers. POSTSCRIPT®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Times and Helvetica. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

Writers: J. Eric Akin, Mike Elola, George Towner, and Kathy Wallace

Editor: George Truett

Production Supervisor: Josephine Manuele

Acknowledgments: Lori Falls and Michael Hinkson

Special thanks to Lorraine Aochi, Vicki Brown, Sharon Everson, Pete Ferrante, Kristi Fredrickson, Don Gentner, Tim Monroe, Dave Payne, Henry Seltzer, and John Sovereign

030-0784