Notice

The information in this document is subject to change without notice.

THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, Bright Star Engineering DOES NOT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Restricted Rights Legend

Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

This document may not, in whole or in part, be copied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from Bright Star Engineering, Inc.

© Bright Star Engineering, Inc.
All Rights Reserved
Printed in the USA
October 1998
Acknowledgements

Our objective in designing the pKernel was to leverage as much existing proven code as possible. To this end the pKernel incorporates software from a number of different sources. Our initial PTHREADS implementation was based on an implementation by John Birrell. While not much of the original code is left, it was invaluable as a starting point. The networking stack and many POSIX system calls are derived from the FreeBSD version of the 4.4BSD UNIX kernel. The zlib compression library is by Jean-loup Gailly and Mark Adler with zip support by Gilles Vollant. The C and Math Libraries are based on Cygnus Solutions NEWLIB. NEWLIB is composed of software from a variety of sources including software developed by DJ Delorie, David M. Gray, AMD, C.W. Sandmann, Eric Backus, Sun Microsystems, HP, and Cygnus Solutions.
CHAPTER 1 POSIX CORE.................................................................................................................. 1-1

MESSAGE QUEUES...................................................................................................................... 1-1
mq_open---Open a Message Queue.......................................................................................... 1-1
mq_close---Close a Message Queue....................................................................................... 1-4
mq_unlink---Remove a Message Queue................................................................................. 1-5
mq_send---Send a Message to a Message Queue ................................................................. 1-6
mq_receive---Receive a Message From A Message Queue ................................................. 1-8
mq_getattr---Get Message Queue Attributes....................................................................... 1-11
mq_notify---Notify Process That a Message is Available on a Queue............................... 1-12

SEMAPHORES............................................................................................................................. 1-13
sem_init---Initialize an Unnamed Semaphore...................................................................... 1-13
sem_destroy---Destroy an Unnamed Semaphore................................................................ 1-14
sem_open---Initialize/Open a Named Semaphore............................................................... 1-15
sem_close---Close a Named Semaphore............................................................................. 1-16
sem_unlink---Remove a Named Semaphore....................................................................... 1-17
sem_wait, sem_trywait---Lock a Semaphore...................................................................... 1-18
sem_post---Unlock a Semaphore......................................................................................... 1-19
sem_getvalue---Get the Value of a Semaphore.................................................................. 1-20

CONDITIONAL VARIABLE ....................................................................................................... 1-21
pthread_cond_broadcast---Unblock All Threads.................................................................... 1-21
pthread_cond_destroy---Destroy Conditional Variable....................................................... 1-22
pthread_cond_init---Initialize a Conditional Variable....................................................... 1-23
pthread_cond_signal---Unblock Thread.............................................................................. 1-24
pthread_cond_timedwait---Block on Conditional Variable with Timeout............................. 1-25
pthread_cond_wait---Block on Conditional Variable......................................................... 1-26
pthread_condattr_destroy---Destroy Conditional Variable Attributes.......................... 1-27
pthread_condattr_init---Create Conditional Variable Attributes..................................... 1-28

MUTEXES.................................................................................................................................... 1-29
pthread_mutexattr_destroy---Destroy Mutex Attributes...................................................... 1-29
pthread_mutexattr_getprioceiling---Get Mutex Attributes Priority Ceiling........................ 1-30
pthread_mutexattr_getprotocol---Get Mutex Attributes Protocol....................................... 1-31
pthread_mutexattr_init---Create Mutex Attributes.................................................................. 1-32
pthread_mutexattr_setprioceiling---Set Mutex Attributes Priority Ceiling......................... 1-33
pthread_mutexattr_setprotocol---Set Mutex Attributes Protocol........................................ 1-34
pthread_mutex_destroy---Destroy Mutex Attributes............................................................ 1-35
pthread_mutex_getprioceiling---Get Mutex Priority Ceiling.............................................. 1-36
pthread_mutex_init---Create a Mutex.................................................................................. 1-37
pthread_mutex_lock---Lock a Mutex..................................................................................... 1-38
pthread_mutex_setprioceiling---Set Mutex Priority Ceiling .............................................. 1-39
pthread_mutex_trylock---Try to Unlock a Mutex .............................................................. 1-40
pthread_mutex_unlock---Unlock a Mutex ......................................................................... 1-41

THREAD CANCELLATION ......................................................................................................... 1-42
pthread_cancel---Cancel Execution of a Thread .................................................................. 1-42
pthread_setcanceltype---Set Cancellation Type ................................................................ 1-44
pthread_setcancelstate---Set Cancellation State ................................................................ 1-45
pthread_testcancel---Create cancellation point ................................................................ 1-46
pthread_cleanup_push---Add handler to cleanup stack .................................................... 1-47
pthread_cleanup_pop---Remove/Execute cleanup handler ............................................... 1-48

THREAD MANAGEMENT .......................................................................................................... 1-49
pthread_attr_destroy---Destroy Thread Attributes.............................................................. 1-49
pthread_attr_getdetachstate---Get Thread Attribute Detach State.................................... 1-50
pthread_attr_setdetachstate---Set Thread Attribute Detach State..................................... 1-51
pthread_attr_getstackaddr---Set Thread Attribute Stack Address.................................... 1-52
pthread_attr_getstacksize---Get Thread Attribute Stack Size ............................................ 1-53
pthread_attr_setstackaddr---Set Thread Attribute Stack Address ..................................... 1-54
pthread_attr_setstacksize---Set the thread creation stacksize ........................................... 1-55
pthread_attr_init---Initialize a thread attributes object..................................................... 1-56
pthread_create---Create a thread...................................................................................... 1-57
pthread_detach---Detach a thread .................................................................................... 1-58
pthread_equal---Compare Threads ................................................................................... 1-59
pthread_exit---Terminate Current Thread......................................................................... 1-60
pthread_join---Wait for Thread Termination..................................................................... 1-61

SCHEDULING ........................................................................................................................... 1-63
sched_yield---Yield Processor ........................................................................................... 1-63
sleep---suspend process execution for an interval measured in seconds......................... 1-64
usleep---suspend process execution for an interval measured in microseconds ............. 1-65

FILE I/O ............................................................................................................................ ....... 1-66
opendir readdir telldir seekdir rewinddir closedir dirfd---directory operations ............... 1-66
chdir fchdir---change current working directory............................................................... 1-68
getcwd getwd---get working directory pathname .............................................................. 1-70
open---open or create a file for reading or writing ........................................................... 1-72
read readv---read input ..................................................................................................... 1-77
write writev---write output................................................................................................. 1-80
close---delete a descriptor ................................................................................................. 1-83
lseek---reposition read/write file offset .............................................................................. 1-85
fcntl---file control........................................................................................................... .... 1-87
ioctl---control device......................................................................................................... .1 - 94
creat---create a new file..................................................................................................... 1-96
link---make a hard file link ............................................................................................... 1-97
mkdir---make a directory file ........................................................................................... 1-100
unlink---remove directory entry ....................................................................................... 1-103
rmdir---remove a directory file ......................................................................................... 1-105
rename---change the name of a file ................................................................................. 1-107
stat lstat fstat---get file status........................................................................................... 1-111
access---check access permissions of a file or pathname ................................................ 1-116
truncate ftruncate---truncate or extend a file to a specified length .................................... 1-119
dup dup2---duplicate an existing file descriptor .............................................................. 1-122
gethostname sethostname---get/set name of current host ............................................. 1-124
gettimeofday settimeofday---get/set date and time ....................................................... 1-126
select---synchronous I/O multiplexing .......................................................................... 1-128
brk sbrk---change data segment size .............................................................................. 1-131

CHAPTER 2 NETWORKING................................................................................................... 2-1

accept---accept a connection on a socket ........................................................................ 2-1
bind---bind a name to a socket ........................................................................................... 2-4
connect---initiate a connection on a socket ..................................................................... 2-7
getdomainname setdomainname---get/set domain name of current host ........................... 2-10
gethostbyname gethostbyname2 gethostbyaddr gethostent sethostent endhostent herror
hstrerror---get network host entry .................................................................................. 2-12
getpeername---get name of connected peer ..................................................................... 2-17
getprotoent getprotobynumber getprotobyname setprotoent endprotoent---get protocol entry
.......................................................................................................................................... 2-19
getservent getservbyport getservbyname setservent endservent---get service entry .... 2-21
getsockname---get socket name ......................................................................................... 2-24
getsockopt setsockopt---get and set options on sockets ..................................................... 2-26
htonl htons ntohs ntohl ﬂoat or double to or from binary order ........................................ 2-31
inet_aton inet_addr inet_network inet_ntoa inet_makeaddr inet_lnaof inet_netof---Internet
address manipulation routines ......................................................................................... 2-32
listen---listen for connections on a socket ...................................................................... 2-35
rcmd rresvport iruserok ruserok---routines for returning a stream to a remote command2-37
recv recvfrom recvmsg---receive a message from a socket .............................................. 2-40
rexec---return stream to a remote command .................................................................. 2-45
send sendto sendmsg---send a message from a socket ...................................................... 2-47
shutdown---shut down part of a full-duplex connection .................................................. 2-50
socket---create an endpoint for communication ............................................................. 2-51
socketpair---create a pair of connected sockets .............................................................. 2-55

CHAPTER 3 C LIBRARY ......................................................................................................... 3-1

STANDARD UTILITY FUNCTIONS .................................................................................. 3-1
abort---abnormal termination of a program......................................................................... 3-3
abs---integer absolute value (magnitude) ......................................................................... 3-4
assert---Macro for Debugging Diagnostics ..................................................................... 3-5
atexit---request execution of functions at program exit ................................................ 3-6
atof, atoff---string to double or float ................................................................................ 3-7
atoi, atol---string to integer ............................................................................................... 3-9
bsearch---binary search .................................................................................................... 3-10
calloc---allocate space for arrays .................................................................................. 3-11
div---divide two integers ................................................................................................. 3-12
ecvt,ecvtf,fcvt,fcvtf---double or float to string ................................................................. 3-13
CHARACTER TYPE MACROS AND FUNCTIONS

toupper---translate characters to upper case .................................................. 3-48
toascii---force integers to ASCII range.......................................................... 3-46
isxdigit---hexadecimal digit predicate ........................................................... 3-45
isupper---uppercase character predicate ......................................................... 3-44
isspace---whitespace character predicate ....................................................... 3-43
ispunct---punctuation character predicate ..................................................... 3-42
isprint, isgraph---printable character predicates ........................................... 3-41
islower---lower-case character predicate ....................................................... 3-40
digit---decimal digit predicate ...................................................................... 3-39
iscntrl---control character predicate .............................................................. 3-38
isascii---ASCII character predicate .............................................................. 3-37
isalpha---alphabetic character predicate ....................................................... 3-36
isalnum---alphanumeric character predicate ............................................... 3-35
system---execute command string ................................................................. 3-32
wctomb---minimal wide char to multibyte converter ...................................... 3-33

INPUT AND OUTPUT.......................................................................................... 3-49

clearerr---clear file or stream error indicator .................................................. 3-52
close---close a file.......................................................................................... 3-53
feof---test for end of file ............................................................................... 3-54
ferror---test whether read/write error has occurred ....................................... 3-55
fflush---flush buffered file output ................................................................. 3-56
fgetc---get a character from a file or stream .................................................. 3-57
fgetpos---record position in a stream or file ................................................... 3-58
fgets---get character string from a file or stream .......................................... 3-59
fopen---open a file .......................................................................................... 3-60
fopen---turn open file into a stream ............................................................... 3-62
fputc---write a character on a stream or file ................................................. 3-63
fputs---write a character string in a file or stream ......................................... 3-64
fread---read array elements from a file ......................................................... 3-65
freopen---open a file using an existing file descriptor .................................... 3-66
fseek---set file position .................................................................................. 3-67
strpbrk---find chars in string .......................................................... 3-132
strncpy---counted copy string .......................................................... 3-131
strncmp---character string compare .................................................. 3-130
strncat---concatenate strings ............................................................. 3-129
strlwr---force string to lower case ..................................................... 3-128
strlen---character string length .......................................................... 3-127
strerror---convert error number to string ........................................... 3-121
strcspn---count chars not in string ..................................................... 3-120
strcpy---copy string .......................................................................... 3-119
strcoll---locale specific character string compare ................................ 3-118
strcmp---character string compare ..................................................... 3-117
strchr---search for character in string ............................................... 3-116
rindex---reverse search for character in string .................................... 3-114
strcat---concatenate strings ............................................................. 3-115
strchr---search for character in string ............................................... 3-116
strcmp---character string compare ..................................................... 3-117
strncpy---character string compare .................................................... 3-130
strncpy---counted copy string .......................................................... 3-131
strpbrk---find chars in string ............................................................. 3-132
CHAPTER 5 COMPRESSION LIBRARY

zlibVersion---get ZLIB version .............................................. 5-1
deflateInit---initialize stream for compression .......... 5-2
deflate---compress stream ....................................................... 5-3
deflateEnd---end compression on stream ..................... 5-6
inflateInit---initialize decompression on stream ......... 5-7
inflate---decompress stream ................................................... 5-8
inflateEnd---end decompression on stream .................. 5-10
deflateInit2---detailed initialization of compression on stream 5-11
deflateSetDictionary---initialize compression dictionary 5-13
inflateInit2---detailed initialization of decompression stream 5-18
deflateSetDictionary---set decompression dictionary ..... 5-19
inflateSync---skip invalid data .............................................. 5-20
inflateReset---end decompression and reset ................. 5-21
compress---compress memory buffer ......................... 5-22
compress2---detailed compress memory buffer ........... 5-23
umcompress---uncompress Buffer ............................. 5-24
gzopen---open gzip file ...................................................... 5-25
gzdopen---open gzip file from file descriptor .......... 5-26
gzsetparams---set parameters for gzip file .............. 5-27
gzread---read from gzip file ......................................... 5-28
gzwrite---write to gzip file ............................................. 5-29
gzprintf---printf to gzip file ....................................... 5-30
gzputs---put string to gzip file .................................. 5-31
gzgets---get line from gzip file ............................... 5-32
gzgetc---get character from gzip file ...................... 5-33
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzflush</td>
<td>flush gzip file</td>
<td>5-34</td>
</tr>
<tr>
<td>gzseek</td>
<td>seek on gzip file</td>
<td>5-35</td>
</tr>
<tr>
<td>gzrewind</td>
<td>rewind gzip file</td>
<td>5-36</td>
</tr>
<tr>
<td>gztell</td>
<td>get gzip file position</td>
<td>5-37</td>
</tr>
<tr>
<td>gzeof</td>
<td>detect gzip file EOF</td>
<td>5-38</td>
</tr>
<tr>
<td>gzclose</td>
<td>close gzip file</td>
<td>5-39</td>
</tr>
<tr>
<td>gzerror</td>
<td>gzip file error</td>
<td>5-40</td>
</tr>
<tr>
<td>Adler32</td>
<td>compute Adler-32 checksum</td>
<td>5-41</td>
</tr>
<tr>
<td>crc32</td>
<td>compute 32 bit CRC</td>
<td>5-42</td>
</tr>
</tbody>
</table>
Chapter 1 POSIX Core

Message Queues

mq_open---Open a Message Queue

Synopsis

```c
#include <mqueue.h>
mqd_t mq_open (const char *name, int oflags, int mode, struct mq_attr *attr);
```

Description

Provides message queue descriptor establishing connection between calling thread and the named message queue. Creates a new message queue descriptor data structure returning a reference to this descriptor to be used in subsequent function calls to access and control the referenced message queue.

The `name` argument points to a string specifying a specific named message queue. Calls to `mq_open()` with the same `name` argument shall refer to the same named message queue. An error is returned if an attempt is made to open a nonexistent message queue and queue creation is not specified via the `oflag` argument. The `oflag` argument also specifies read and/or write access of the message queue. The `oflag` argument is formed by bitwise exclusive OR of the following values:

- **O_RDONLY**
  After opening, only `mq_receive()` may be used with the descriptor to read messages from the queue. Attempts to send messages with `mq_send()` shall result in the return of an error code.

- **O_WRONLY**
  After opening, only `mq_send()` may be used with the descriptor to send messages to the queue. Attempts to receive messages with `mq_receive()` shall result in the return of an error code.

- **O_RDWR**
  Both sending and receiving operations with the opened message queue are supported.
(Note: Only one of these first three values shall be specified)

**O_CREAT** Indicates that a new, empty message is to be opened. If the named message queue already exists, then this flag value has no effect unless the O_EXCL flag is also set in which case the mq_open() function shall fail returning EEXIST. Upon successful creation of a new message queue, the `attr` argument shall be used to define the attributes (mq_maxmsg, mq_msgsize) of the new message queue. If `attr` is NULL, implementation defined default attributes are assigned to the message queue.

**O_EXCL** When set along with O_CREAT, indicates that the open is to be exclusive of any other attempt to create the message queue. If both of these flags are set and the named message queue already exists, then EEXIST shall be returned and the mq_open() shall fail.

**O_NONBLOCK** Determines whether subsequent mq_send() or mq_receive() operations shall cause thread suspension waiting for a message queue slot or a enqued message, respectively. If this flag is set, conditional suspension is not allowed for mq_send() or mq_receive.

The `mode` argument has no effect.

The `attr` argument shall either point to a valid `mq_attribute` data structure specifying the number of messages and maximum message size for a newly created message queue, or the argument shall be NULL in which case the implementation specified default attributes shall be used for a newly opened message queue.

**Returns**

Returns a valid descriptor on successful opening of a message queue. Otherwise, returns –1 and sets `errno` to indicate the error.

- **EACCES** - Message queue was unlinked with pending destroy and no creation
requested

**EEXIST** - File exists

**EINVAL** - Invalid Argument *attr*

**ENAMETOOLONG** - File or path name too long

**ENOENT** - No such file or directory

**ENOSPC** - No space left
mq_close---Close a Message Queue

Synopsis
#include <mqueue.h>

int mq_close (mqd_t mqdes);

Description
Removes association of specified message queue descriptor and the opened message queue.

Removes attachment for notification request if present.

If this is the last descriptor associated with this message queue and an mq_unlink() is pending on the message queue, then the message queue is destroyed.

Returns
Return 0 on successful closing of a message queue. Otherwise, returns –1 and sets errno to indicate the error.
EBADF - Bad file number
mq_unlink---Remove a Message Queue

Synopsis

#include <mqueue.h>

int mq_unlink (const char *name);

Description

Destroys the named message queue unless at least one open descriptor is associated with the queue, in which case, the unlink operation is set pending until the last open descriptor is closed. Once destroyed, all accesses to the named message queue shall fail until a mq_open() is performed with the flags argument set with O_CREAT.

Returns

Return 0 on successful unlinking of a message. Otherwise, returns –1 and sets errno to indicate the error.

ENOENT - No such file or directory
mq_send---Send a Message to a Message Queue

Synopsis

#include <mqueue.h>

int mq_send (mqd_t mqdes, const char *buf, size_t len, unsigned int prio);

Description

The message pointed to by the buf argument and of length (in bytes) len is added to the message queue denoted by the descriptor argument mqdes. The message is inserted into the message queue if space is available. The messages are enqueued in priority order with the new message’s priority specified by the prio argument; messages of higher numeric value are inserted with higher priority and therefore made available in a subsequent mq_receive() function call before messages with priorities of lower numeric value. Messages of equal priority are handled in FIFO order.

If the message queue is full, functional behavior is conditional on the current state of the queue’s O_NONBLOCK attribute flag. If the flag is set, the message is not queued and the function returns an error. If the O_NONBLOCK flag is not set, the calling thread shall block until space is made available in the queue. If multiple threads are blocked waiting for “send” access to the queue, threads of higher priority are resumed first and threads of equal priority are resumed in FIFO order.

Returns

Return 0 on successful sending of a message. Otherwise, returns −1 and sets errno to indicate the error.

EAGAIN – No room available on the queue and no blocking allowed

EBADF – The mqdes argument is not a valid message queue descriptor open for writing

EINVAL - Invalid Argument prio
EMSGSIZE - Message Too Large
mq_receive---Receive a Message From A Message Queue

Synopsis

#include <mqueue.h>

ssize_t mq_receive (mqd_t mqdes, char *buf, size_t len, unsigned int *prio);

Description

Receives the next highest priority message from the message queue specified by the mqdes argument. The message is copied to the location pointed to by the buf argument unless the size of the receiving buffer, specified by the len argument, is smaller than the maximum message size of the queue as specified by the attr.mq_msgsize parameter on mq_open(). The priority of the received message shall be written to the location pointed to by the *prio argument unless this argument is NULL.

If the message queue is empty, the functional behavior is conditional on the current state of the queue’s O_NONBLOCK flag. If the flag is set, mq_receive() will return with an error and no message shall be received. If the flag is not set, the calling thread shall be blocked until a new message is written to the queue. Multiple threads waiting on an empty queue shall be resumed in thread priority order and threads of equal priority shall be resumed in FIFO order.

Returns

For successful message reception, returns the actual number of bytes in the message copied to buf. Otherwise the return value is –1 and errno is set to indicate the error.

EAGAIN – Attempt to receive from an empty queue and blocking not allowed

EBADF - The mqdes argument is not a valid message queue descriptor open for reading
EMSGSIZE – Length of message buffer not at least $mq_{-}msgsiz e$ in length
mq_setattr---Set Message Queue Attributes

Synopsis

#include <mqueue.h>

int mq_setattr (mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr *omqstat);

Description

Modifies the open message queue denoted by the mqdes argument. Only the O_NONBLOCK flag bit within the *mqstat mq_attr structure’s mq_flags element is used to modify the flag attributes of the target message queue. The mq_msgsize and mq_maxmsg members of the mq_attr structure are ignored within mq_setattr(). If not NULL, the *omqstat argument pointer shall be used to receive the message queue’s previous attributes and current status.

Returns

Upon successful update of the queue’s attribute, a value of 0 shall be returned. Otherwise a value of –1 shall be returned and errno updated to indicate the error condition.

EBADF - Bad mqdes message queue descriptor or the mqstat argument is NULL.
mq_getattr---Get Message Queue Attributes

Synopsis
#include <mqueue.h>

int mq_getattr (mqd_t mqdes, struct mq_attr *omqstat);

Description
Acquires the current status (mq_curmsgs the number of messages currently in the queue) and attributes (mq_flags, mq_maxmsg, and mq_msgsize) of the message queue denoted by the mqdes argument. The return information is placed in the structure pointed to by the omqstat argument.

Returns
The function shall return 0 on successful completion of the getattr. Otherwise, a value of –1 shall be returned and errno shall be set with the appropriate error indicator.

EBADF - Bad mqdes message queue descriptor or the omqstat argument is NULL.
mq_notify---Notify Process That a Message is Available on a Queue

Synopsis

```c
#include <mqueue.h

int mq_notify(mqd_t mqdes, const struct sigevent *ev);
```

Description

This function is not supported.

Returns

Returns a –1 and sets `errno` to `ENOSYS`.

`ENOSYS` - Function not implemented
Semaphores

sem_init---Initialize an Unnamed Semaphore

Synopsis
#include <semaphore.h>

int sem_init(sem_t *location, int pshared, unsigned int value);

Description
Creates an unnamed semaphore which can be subsequently referenced via the object pointed to by the location argument. The newly created semaphore is initialized with the value argument and semaphore’s thread queue is initialized to an empty state. The pshared argument is not used and is for future extension of sem_init()’s functionality.

Returns
Upon successful creation of the new semaphore, a value of zero is returned. Otherwise a value of –1 shall be returned.
sem_destroy---Destroy an Unnamed Semaphore

Synopsis
#include <semaphore.h>

int sem_destroy (sem_t *location );

Description
The memory allocated to the unnamed semaphore data object pointed to by the location argument is freed. The destruction of a semaphore upon which threads are currently enqueued will not take place.

Returns
A zero return value shall be returned upon a successful destruction of the semaphore. Otherwise, a value of –1 shall be returned and errno set to reflect the error condition.

EINVAL - Invalid location argument

EBUSY – Threads are enqueued on the semaphore
sem_open---Initialize/Open a Named Semaphore

Synopsis

#include <semaphore.h>

sem_t *sem_open (const char *name, int oflag, int mode, int value);

Description

If the object does not already exist, creates a named semaphore object and associates the object with the name provided in the name argument. The newly created semaphore is initialized with the value argument and semaphore’s thread queue is initialized to an empty state. Creation of a new semaphore requires that the oflag argument include a set O_CREAT bit. If both O_CREATE and O_EXCL bits are set in the oflag argument and the named semaphore already exits, the sem_open() shall fail.

If the object does already exists, the returned semaphore pointer shall pointed to the named semaphore. Multiple sem_open() calls with the same name argument shall all return pointers to the same semaphore object.

Returns

Upon a successful open, a valid semaphore pointer shall be returned. Otherwise, a value of –1 shall be returned and errno set to reflect the error condition.

EEXIST – Both the O_CREAT and O_EXCL bits are set in the oflag argument and the named semaphore already exits

ENFILE – Memory not available to create a new semaphore

ENOENT – Named semaphore does not exist and OCrear not set.
sem_close---Close a Named Semaphore

Synopsis
#include <semaphore.h>

int sem_close (sem_t *sem);

Description
Not Implemented.

Returns
EINVAL - Invalid Argument
sem_unlink---Remove a Named Semaphore

Synopsis
#include <semaphore.h>

int sem_unlink (const char *name);

Description
Removes the association between the name and the previously opened semaphore.

Returns
Always returns zero
sem_wait, sem_trywait---Lock a Semaphore

Synopsis
#include <semaphore.h>

int sem_wait ( sem_t *sem );

int sem_trywait ( sem_t *sem );

Description
The sem_wait() function evaluates the value of the semaphore referenced by the sem argument and if the value is greater than zero, the value is decremented. If the value is equal to or less than zero, the calling thread is suspended to wait until a subsequent sem_post() operation is performed on the semaphore. Threads are inserted on the semaphore’s wait queue in priority order and threads of equal priority are inserted in FIFO order.

The sem_trywait() function evaluates the specified semaphore’s value and if it is greater than zero, the value is decremented. If the value is zero or less, the function returns with a return errno of EAGAIN.

Returns
Both functions return a value of zero if the semaphore lock (decrement value to zero) is successful. If the semaphore is not successfully locked, a value of –1 is returned and errno set to indicate the error.

EAGAIN – sem_trywait() found the semaphore’s value to be zero or less

EINVAL - Invalid sem argument
sem_post---Unlock a Semaphore

Synopsis

```
#include <semaphore.h

int sem_post (sem_t *sem);
```

Description

Unlocks semaphore referenced `sem` argument. If the semaphore’s wait queue is empty, its value is incremented. If the wait queue is not empty, the thread waiting at the head of the queue is released for execution.

Returns

If successful, the `sem_post()` function shall return 0. Otherwise the function shall return a value of –1 and set `errno` to indicate the error.

- **EINVAL** - Invalid `sem` argument
sem_getvalue—Get the Value of a Semaphore

Synopsis

#include <semaphore.h>

int sem_getvalue (sem_t *sem, int *value);

Description
The value of the semaphore referenced by the sem argument is written into the data object pointed to by the value object.

Returns
If successful, the function shall return 0. Otherwise the function shall return a value of −1 and set errno to indicate the error.

EINVAL - Invalid sem argument
Conditional Variable

pthread_cond_broadcast---Unblock All Threads

Synopsis
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);

Description
Removes all threads currently suspended on the condition variable referenced by the cond argument.

Returns
If the function is successful, a value of 0 is returned. Otherwise, a value of –1 is returned and errno set to indicate the error.

EINVAL - Invalid cond argument
pthread_cond_destroy---Destroy Conditional Variable

Synopsis

```c
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t * cond);
```

Description

The conditional variable data object reference by the `cond` argument is destroyed with a freeing of its memory resources.

Returns

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

- **EINVAL** - Invalid `cond` argument
pthread_cond_init---Initialize a Conditional Variable

Synopsis

#include <pthread.h>

int pthread_cond_init(pthread_cond_t * cond, const
pthread_condattr_t * cond_attr);

Description

A conditional variable data object referenced via the cond argument is initialized. The type of the conditional variable can be specified via the cond_attr argument. Note, only COND_TYPE_FAST conditional variables are supported.

Returns

If the function is successful, a value of zero is returned. Otherwise a non-zero error value is returned.

EINVAL - Invalid Argument
pthread_cond_signal--- Unblock Thread

Synopsis

#include <pthread.h>

int pthread_cond_signal(pthread_cond_t * cond);

Description
If there is a thread blocked on the condition variable referenced by the cond argument, it is released from suspension and made ready for execution.

Returns
A value of zero is returned upon success. Otherwise a non-zero error value shall be returned to indicate the error.

EINVAL - Invalid cond argument
**pthread_cond_timedwait---Block on Conditional Variable with Timeout**

**Synopsis**
#include <pthread.h>

```c
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
```

**Description**
The calling thread is enqueued on the condition variable referenced by the `cond` argument and suspended awaiting a signaling via `pthread_cond_signal()` or a timeout of the period specified by the `abstime` argument. The function `pthread_cond_timedwait()` should only be called with the mutex referenced by the `mutex` argument locked or undesirable thread behavior may result. Before suspension of the thread, the referenced mutex is unlocked. The function always returns with the mutex locked.

**Returns**
A return value of zero shall be presented on a successful execution of the function. A non-zero return value shall indicate an error.

- **EINVAL** - Invalid `cond` argument

- **ETIMEDOUT** - The time specified by the `abstime` argument passed
pthread_cond_wait---Block on Conditional Variable

Synopsis
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex);

Description
The calling thread is enqueued on the condition variable referenced by the cond argument and suspended awaiting a signaling via pthread_cond_signal(). The function pthread_cond_wait() should only be called with the mutex referenced by the mutex argument locked or undesirable thread behavior may result. Before suspension of the thread, the referenced mutex is unlocked. The function always returns with the mutex locked.

Returns
A return value of zero shall be presented on a successful execution of the function. A non-zero return value shall indicate an error.

EINVAL - Invalid cond argument
pthread_condattr_destroy---Destroy Conditional Variable Attributes

Synopsis
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);

Description
The condition variable attribute object pointed to by the attr argument is destroyed (i.e., its memory is freed) and sets the reference value to NULL.

Returns
Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

EINVAL - Invalid Argument
**pthread_condattr_init---Create Conditional Variable Attributes**

**Synopsis**

```c
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);
```

**Description**

Allocates storage for a conditional variable attribute data object and initializes the object with implementation-defined default values (see pthread.h).

**Returns**

Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

- **ENOMEM** – memory not available for creation of new data object
Mutexes

pthread_mutexattr_destroy---Destroy Mutex Attributes

Synopsis
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

Description
The mutex attribute object pointed to by the attr argument is destroyed (i.e., its memory is freed) and sets the reference value to NULL.

Returns
Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

EINVAL - Invalid Argument
pthread_mutexattr_getprioceiling---Get Mutex Attributes Priority Ceiling

Synopsis
#include <pthread.h>

int pthread_mutexattr_getprioceiling(pthread_mutexattr_t *attr, int *prioceiling);

Description
The prioceiling value for the mutex attribute object referenced by the attr argument is copied to the location referenced by the prioceiling argument.

Returns
Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

EINVAL - Invalid Argument
pthread_mutexattr_getprotocol---Get Mutex Attributes Protocol

Synopsis
#include <pthread.h>

int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr, int *protocol);

Description
The protocol value for the mutex attribute object referenced by the attr argument
is copied to the location referenced by the protocol argument.

Returns
Upon success, a value of zero is returned. Otherwise a non-zero error value is
returned.

EINVAL - Invalid Argument
**pthread_mutexattr_init---Create Mutex Attributes**

**Synopsis**

```c
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

**Description**

Allocates storage for a mutex attribute data object and initializes the object with implementation-defined default values (see pthread.h).

**Returns**

Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

- **ENOMEM** – memory not available for creation of new data object
pthread_mutexattr_setprioceiling---Set Mutex Attributes Priority Ceiling

Synopsis
#include <pthread.h>

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);

Description
Copies the prioceiling value passed by the prioceiling argument into the mutex attribute object referenced by the attr argument.

Returns
Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

EINVAL - Invalid Argument
pthread_mutexattr_setprotocol---Set Mutex Attributes Protocol

Synopsis
#include <pthread.h>

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);

Description
Copies the protocol value passed by the protocol argument into the mutex attribute object referenced by the attr argument.

Returns
Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

EINVAL - Invalid Argument
pthread_mutex_destroy---Destroy Mutex

Attributes

Synopsis

#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);

Description

The mutex object referenced by the mutex argument is re-initialized with a count of zero, empty thread queue, no owner thread, and all flags set to zero. Destroying a locked mutex is permitted and results in undefined thread behavior.

Returns

Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

EINVAL - Invalid Argument
**Synopsis**

```c
#include <pthread.h>

int pthread_mutex_getprioceiling(pthread_mutex_t *mutex, int *prioceiling);
```

**Description**

The `prioceiling` value for the mutex object referenced by the `mutex` argument is copied to the location referenced by the `prioceiling` argument.

**Returns**

Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

**EINVAL - Invalid Argument**
**pthread_mutex_init---Create a Mutex**

**Synopsis**

```c
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *mutex_attr);
```

**Description**

A mutex data object is created and a reference to the new data structure is passed back via the `mutex` argument. The new mutex is either initialized with implementation-defined default values or via the attributes specified via the `mutex_attr` argument.

**Returns**

Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

- **EINVAL** - Invalid Argument
- **ENOMEM** – No memory available for creation of a new object
pthread_mutex_lock---Lock a Mutex

Synopsis
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

Description
The calling thread attempts to lock and gain ownership of the mutex referenced by the mutex argument. If the mutex is found to be already locked, the calling thread will be enqueued on the mutex and suspended until the mutex is unlocked. The priority of the thread may be conditionally modified depending on the mutex priority protocol and relative priorities of this thread and other threads interacting with the mutex. A successful return indicates that the calling thread has gained exclusive mutex ownership.

Returns
Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

EINVAL - Invalid Argument


pthread_mutex_setprioceiling---Set Mutex Priority Ceiling

Synopsis

```c
#include <pthread.h>

int pthread_mutex_setprioceiling(pthread_mutex_t *mutex, int prioceiling, int *old_ceiling);
```

Description

Copies the prioceiling value passed by the `prioceiling` argument into the mutex object referenced by the `mutex` argument. Before updating the mutex, its current value of prioceiling is first copied out through the `old_ceiling` argument if this argument is not NULL. The priority of the thread currently owning the mutex is re-evaluated and the thread’s position on the run queue (if the thread is on the run queue) is also adjusted. This thread priority adjustment is only performed if the mutex’s protocol is PTHREAD_PRIO_PROTECT.

Returns

Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

EINVAL - Invalid Argument
**Synopsis**

```c
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

**Description**

The calling thread attempts to lock and gain ownership of the mutex referenced by the `mutex` argument. If the mutex is found to be already locked, the calling thread will return with the error value EBUSY and the calling thread will not have mutex ownership. A successful return indicates that the calling thread has gained exclusive mutex ownership. The priority of the thread may be conditionally modified upon gaining ownership depending on the mutex priority protocol and relative priorities of this thread and other threads interacting with the mutex.

**Returns**

Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

- **EINVAL** - Invalid Argument
- **EBUSY** - Resource busy
pthread_mutex_unlock---Unlock a Mutex

Synopsis

#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);

Description

The calling thread relinquishes exclusive ownership of the mutex referenced by the mutex argument. If a thread is enqueued on the mutex and waiting to take ownership, it is released from the queue and made ready for execution. The priority of the calling thread may be modified depending on the mutex’s priority protocol.

Returns

Upon success, a value of zero is returned. Otherwise a non-zero error value is returned.

EINVAL - Invalid Argument or calling thread does not own mutex
Thread Cancellation

pthread_cancel---Cancel Execution of a Thread

Synopsis

```c
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Description

Requests that the thread referenced by the `thread` argument be cancelled. The specific actions taken in this function are determined by the cancellation state and type for the specified thread to be cancelled. If the cancellation state is disabled, then a pending cancellation request is set for the thread and `pthread_cancel()` returns with no further action at this time. If the cancellation state is enabled, the action taken is dependant on the cancellation type setting for the thread. If the cancellation type is set for deferred and the target thread is not currently suspended at a cancellation waiting point, a pending cancellation request is set for the thread and `pthread_cancel()` returns with no further action at this time. A thread with enabled cancellation state and deferred type and suspended at a cancellation waiting point is actively cancelled within the `pthread_cancel()` function call. Cancellation waiting points are waiting for conditional variable, waiting for expiration of a sleep interval, waiting to complete a thread join operation, waiting on a semaphore, or waiting on a mutex. Immediate cancellation is also performed for a thread with enabled cancellation state and asynchronous cancellation type.

Active and immediate cancellation entails several actions. If the target thread is currently waiting for a conditional variable, the mutex associated with the conditional variable is locked before the thread’s cleanup routines are executed. All cleanup routines are executed. If there is thread-specific data, the thread-specific data destruction routine is executed. If there are threads waiting to `join` with the thread being cancelled, they are released for execution and the return value from the joined thread is set to PTHREAD_CANCELED. If the cancelled thread is currently waiting on the run queue or any other wait queue, it is removed from this queue. If the cancelled thread is currently set to a detached state,
storage associated with the thread is reclaimed, otherwise the thread is simply suspended without storage reclamation. The cancelled thread is set to the DEAD state and if it is the currently executing thread, a rescheduling operation is performed.

**Returns**
Upon a successful execution of this function, a value of 0 is returned. (Note, successful execution does not imply that the target thread was actively cancelled.) Otherwise, a non-zero return value shall be returned to indicate the error.

**ESRCH** – No thread found matching the *thread* argument.
pthread_setcanceltype---Set Cancellation Type

Synopsis
#include <pthread.h>
int pthread_setcanceltype(int newtype, int *oldtype);

Description
This function first copies out the current cancel type value for the calling thread,
and then sets the cancel type to the value of the newtype argument. The cancel
type value can be either PTHREAD_CANCEL_DEFERRED or
PTHREAD_CANCEL_ASYNCHRONOUS. The oldtype argument references
the object to receive the original cancel type value.

This function will result in immediate cancellation of the calling thread if the
cancellation state of the thread is enabled, there is a pending cancellation request,
and the new cancellation type is being set to asynchronous.

Returns
If successful, the function returns a value of zero. Otherwise a non-zero value is
returned to indicate the error.
EINVAL - Invalid newtype argument
pthread_setcancelstate---Set Cancellation State

Synopsis
#include <pthread.h>
int pthread_setcancelstate(int newstate, int *oldstate);

Description
This function first copies out the current cancel state value for the calling thread, and then sets the cancel state to the value of the newstate argument. The cancel state value can be either PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE. The oldstate argument references the object to receive the original cancel state value.

This function will result in immediate cancellation of the calling thread if the cancellation type of the thread is asynchronous, there is a pending cancellation request, and the new cancellation state is being set to enable.

Returns
If successful, the function returns a value of zero. Otherwise a non-zero value is returned to indicate the error.
EINVAL - Invalid newstate argument
pthread_testcancel---Create cancellation point

Synopsis

```
#include <pthread.h>
void pthread_testcancel(void);
```

Description

Checks if there is a pending cancellation and the cancellation state of the calling thread is set to enable. If these conditions are met, the calling thread is actively cancelled as described for `pthread_cancel()`.

Returns

No return value provided.
pthread_cleanup_push---Add handler to cleanup stack

Synopsis
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);

Description
Pushes the cleanup routine referenced via the routine argument and the routine’s arguments referenced via the arg argument on the calling thread’s cleanup routine stack. The pushed routine shall be subsequently executed with its specified arguments when the thread exits, the thread acts upon a cancellation request, or the pthread_cleanup_pop() function is called with arguments requesting execution of the routine being popped.

Returns
No return value provided.
**Synopsis**

```c
#include <pthread.h>
void pthread_cleanup_pop(int execute);
```

**Description**

Removes the cleanup routine from the top of the calling thread’s cleanup stack. If the stack is not empty and the `execute` argument is non-zero, the popped cleanup routine is executed.

**Returns**

No return value provided.
Thread Management

pthread_attr_destroy---Destroy Thread Attributes

Synopsis
#include <pthread.h>
int pthread_attr_destroy(pthread_attr_t *attr);

Description
The storage allocated for the thread attributes object referenced by the attr argument is freed. The pointer to this object is set to NULL.

Returns
A value of zero is returned upon successful completion. Otherwise a non-zero error value is returned.

EINVAL – Illegal attr argument
pthread_attr_getdetachstate---Get Thread Attribute Detach State

Synopsis

#include <pthread.h>
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);

Description

Returns the detach state from the thread attribute object referenced by the attr argument. The returned state value is copied to the object referenced by the detachstate argument.

Returns

A value of zero is returned upon successful completion. Otherwise a non-zero error value is returned.

EINVAL – Illegal argument
**pthread_attr_setdetachstate---Set Thread Attribute Detach State**

**Synopsis**

```c
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

**Description**

Sets the detach state in the thread attribute object referenced by the `attr` argument to the value provided by the `detachstate` argument. Legal values are `PTHREAD_CREATE_DETACHED` and `PTHREAD_CREATE_JOINABLE`.

**Returns**

A value of zero is returned upon successful completion. Otherwise a non-zero error value is returned.

`EINVAL` – Illegal argument
**pthread_attr_getstackaddr---Set Thread Attribute Stack Address**

**Synopsis**

```c
#include <pthread.h>
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);
```

**Description**

Returns the stack address attribute from the thread attribute object referenced by the `attr` argument. The stack address is returned via the `stackaddr` argument.

**Returns**

A value of zero is returned upon successful completion. Otherwise a non-zero error value is returned.

EINVAL – Illegal argument
pthread_attr_getstacksize---Get Thread Attribute Stack Size

Synopsis
#include <pthread.h>
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);

Description
Returns the stack size attribute from the thread attribute object referenced by the attr argument. The stack size is returned via the stacksize argument. The stack size attributes defines the minimum thread stack size in bytes.

Returns
A value of zero is returned upon successful completion. Otherwise a non-zero error value is returned.

EINVAL – Illegal argument
**pthread_attr_setstackaddr---Set Thread Attribute Stack Address**

**Synopsis**

```c
#include <pthread.h>
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

**Description**

Sets the stack address value passed via the `stackaddr` argument into the stack address attribute field of the thread attribute object referenced by the `attr` argument. The stack address specifies a location for allocating a thread’s new stack.

**Returns**

A value of zero is returned upon successful completion. Otherwise a non-zero error value is returned.

EINVAL – Illegal argument
pthread_attr_setstacksize---Set the thread creation stacksize

Synopsis
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

Description
Sets the stack size attribute for the thread attribute object referenced by the attr argument to the value passed by the stacksize argument. This attribute defines the minimum stack size, in bytes, for the thread created with these attributes. The stacksize argument must be at least PTHREAD_STACK_MIN in value.

Returns
A value of zero is returned upon successful completion. Otherwise a non-zero error value is returned.

EINVAL – Illegal argument
**pthread_attr_init---Initialize a thread attributes object**

**Synopsis**

```c
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```

**Description**

Allocates storage for a thread attribute object, initializes the object to default values, and returns a reference to this object via the `attr` argument.

**Returns**

A value of zero is returned upon successful completion. Otherwise a non-zero error value is returned.

- **ENOMEM** – Insufficient memory is available for allocating the object.
#include <pthread.h>
int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void *(*start_routine) (void *), void *arg);

Description

Creates a new thread and provides a thread identification object referenced by the thread argument. The attributes of the newly created thread are determined by the attr argument. Default thread attributes are used in creating the new thread if attr is NULL. Execution of the newly created thread begins with the function specified by the start_routine argument which is provided with the arg argument. A call to pthread_exit() shall be made upon return from the start_routine() function with the return value from start_routine() provided as an input to pthread_exit().

Returns

A value of zero is returned upon successful completion. Otherwise a non-zero error value is returned.

EAGAIN - Insufficient memory to create a thread
**pthread_detach---Detach a thread**

**Synopsis**

```c
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

**Description**

The “detached” state of the thread referenced by the `thread` argument is set to indicate that memory resources used by the thread are to be reclaimed when the thread terminates. If the “detached” state of the thread is not already set, then all threads waiting to join with the target thread are released for execution.

**Returns**

A value of zero is returned upon successful completion. Otherwise a non-zero error value is returned.

- **EINVAL** - Invalid Argument
- **ESRCH** – thread is already detached
pthread_equal---Compare Threads

Synopsis

```c
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

Description

Determines if the two thread `t1` and `t2` arguments reference the same thread object.

Returns

Returns a non-zero value if the two thread arguments reference the same thread object. Otherwise returns a zero value.


pthread_exit---Terminate Current Thread

Synopsis

#include <pthread.h>
void pthread_exit(void *value_ptr);

Description

Terminates the calling thread and passes the value provided by the \textit{value\_ptr} argument on to any thread successfully joined to the calling thread. All cleanup routines pushed on the stack of cleanup routines for the terminating thread are popped and executed in LIFO (Last-in, First-out) order. If the thread has thread-specific data, a \_\texttt{thread\_cleanupspecific}() routine is called to destroy this data. All threads waiting to join with the terminated thread are released for execution. If the terminating thread \textit{detach} attribute flag is set, then memory resources (thread stack, thread control block) are freed and standard file descriptors are closed. If the \textit{detach} attribute flag is not set, the thread is added to a list of “dead threads” with no reclamation of its memory resources.

The next available thread to execute is allowed to run at the end of this function.

Returns

Does not return.
**pthread_join---Wait for Thread Termination**

**Synopsis**

```c
#include <pthread.h>
int pthread_join(pthread_t pthread, void **thread_return);
```

**Description**

Synchronizes execution of calling thread with the termination of the thread referenced by the `pthread` argument. If the targeted thread is detached, `pthread_join()` returns an error value. If the targeted thread has already terminated, the calling thread is not suspended. Otherwise, the calling thread is suspended until the targeted thread has called `pthread_exit()` or has been cancelled. Upon return from `pthread_join()`, the terminated thread’s return value is copied back through the `thread_return` argument if this argument is non-NULL.

**Returns**

`pthread_join()` returns a zero value if successful. Otherwise, a non-zero return value shall be returned to indicate the error.

- ESRCH – Thread specified by `pthread` argument not found.
- EINVAL -- Thread specified by `pthread` argument can not be joined.
**pthread_self---Get Current Thread**

**Synopsis**

```c
#include <pthread.h>
pthread_t pthread_self();
```

**Description**

Returns the thread ID of the calling thread.

**Returns**

Returns the thread ID of the calling thread.
Scheduling

sched_yield---Yield Processor

Synopsis
#include <pthread.h>

int sched_yield(void);

Description
The calling thread will relinquish the processor if another thread of higher or equal priority is currently ready to run. The calling thread will resume execution once it works its way back to the top of the ready queue.

Returns
Always returns a value of zero.
sleep—suspend process execution for an interval measured in seconds

Synopsis
#include <unistd.h>

unsigned int sleep(unsigned int seconds);

Description
The sleep function suspends execution of the calling process until either seconds seconds have elapsed or a signal is delivered to the process and its action is to invoke a signal-catching function or to terminate the process. System activity may lengthen the sleep by an indeterminate amount.

Return Values
If the sleep function returns because the requested time has elapsed, the value returned will be zero. If the sleep function returns due to the delivery of a signal, the value returned will be the unslept amount (the requested time minus the time actually slept) in seconds.

See Also
usleep

Standards
The sleep function conforms to p1003.1-90.
usleep---suspend process execution for an interval measured in microseconds

Synopsis
#include <unistd.h>

int usleep(unsigned int microseconds);

Description
The `usleep` function suspends execution of the calling process until either `microseconds` microseconds have elapsed or a signal is delivered to the process and its action is to invoke a signal-catching function or to terminate the process. System activity may lengthen the sleep by an indeterminate amount.

Errors
The `usleep` function will fail if:

EINTR

A signal was delivered to the process and its action was to invoke a signal-catching function.

See Also
sleep
File I/O

opendir readdir telldir seekdir rewinddir closedir
dirfd---directory operations

Synopsis
#include <sys/types.h>
#include <dirent.h>

DIR * opendir(const char *filename);
struct dirent * readdir(DIR *dirp);
long telldir(const DIR *dirp);
void seekdir(DIR *dirp, long loc);
void rewinddir(DIR *dirp);
int closedir(DIR *dirp);
int dirfd(DIR *dirp);

Description
The opendir function opens the directory named by filename, associates a
directory stream with it and 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 enough memory to hold the whole thing.

The readdir function 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 \texttt{telldir} function returns the current location associated with the named \textit{directory stream}.

The \texttt{seekdir} function sets the position of the next \texttt{readdir} operation on the \textit{directory stream}. The new position reverts to the one associated with the \textit{directory stream} when the \texttt{telldir} operation was performed. Values returned by \texttt{telldir} are good only for the lifetime of the \texttt{DIR} pointer, \texttt{dirp}, from which they are derived. If the directory is closed and then reopened, the \texttt{telldir} value may be invalidated due to undetected directory compaction. It is safe to use a previous \texttt{telldir} value immediately after a call to \texttt{opendir} and before any calls to \texttt{readdir}.

The \texttt{rewinddir} function resets the position of the named \textit{directory stream} to the beginning of the directory.

The \texttt{closedir} function closes the named \textit{directory stream} and frees the structure associated with the \texttt{dirp} pointer, returning 0 on success. On failure, -1 is returned and the global variable \texttt{errno} is set to indicate the error.

The \texttt{dirfd} function returns the integer file descriptor associated with the named \textit{directory stream} see \texttt{open}.

Sample code which searches a directory for entry "name" is:

```c
len = strlen(name);
dirp = opendir(".");
while ((dp = readdir(dirp)) != NULL)
    if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
        (void)closedir(dirp);
        return FOUND;
    }
(void)closedir(dirp);
return NOT_FOUND;
```

\textbf{See Also}
\texttt{close}, \texttt{lseek}, \texttt{open}, \texttt{read}, \texttt{dir}(5)
chdir fchdir---change current working directory

Synopsis
#include <unistd.h>

int chdir(const char *path);

int fchdir(int fd);

Description
The path argument points to the pathname of a directory. The chdir function causes the named directory to become the current working directory, that is, the starting point for path searches of pathnames not beginning with a slash, `/`

The fchdir function causes the directory referenced by fd to become the current working directory, the starting point for path searches of pathnames not beginning with a slash, `/`

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

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

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

ENOTDIR

A component of the path prefix is not a directory.

ENAMETOOLONG

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

The named directory does not exist.

ELOOP

Too many symbolic links were encountered in translating the pathname.

EACCES

Search permission is denied for any component of the path name.

EFAULT

Path points outside the process’s allocated address space.

EIO

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

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

EACCES

Search permission is denied for the directory referenced by the file descriptor.

ENOTDIR

The file descriptor does not reference a directory.

EBADF

The argument fd is not a valid file descriptor.

See Also

chroot

Standards

The chdir function call is expected to conform to p1003.1-90.
getcwd getwd---get working directory pathname

Synopsis
#include <unistd.h>

char * getcwd(char *buf, size_t size);

char * getwd(char *buf);

Description
The getcwd function copies the absolute pathname of the current working directory into the memory referenced by buf and returns a pointer to buf. The size argument is the size, in bytes, of the array referenced by buf.

If buf is NULL space is allocated as necessary to store the pathname. This space may later be free’d.

The function getwd is a compatibility routine which calls getcwd with its buf argument and a size of MAXPATHLEN (as defined in the include file sys/param.h). Obviously, buf should be at least MAXPATHLEN bytes in length.

These routines have traditionally been used by programs to save the name of a working directory for the purpose of returning to it. A much faster and less error-prone method of accomplishing this is to open the current directory (‘.’) and use the fchdir function to return.

Return Values
Upon successful completion, a pointer to the pathname is returned. Otherwise a NULL pointer is returned and the global variable errno is set to indicate the error. In addition, getwd copies the error message associated with errno into the memory referenced by buf.

Errors
The getcwd function will fail if:
EACCES

Read or search permission was denied for a component of the pathname.

EINVAL

The size argument is zero.

ENOENT

A component of the pathname no longer exists.

ENOMEM

Insufficient memory is available.

ERANGE

The size argument is greater than zero but smaller than the length of the pathname plus 1.

See Also
chdir, fchdir, malloc, strerror

Standards
The getcwd function conforms to ansiC. The ability to specify a NULL pointer and have getcwd allocate memory as necessary is an extension.

Bugs
The getwd function does not do sufficient error checking and is not able to return very long, but valid, paths. It is provided for compatibility.
open---open or create a file for reading or writing

Synopsis
#include <fcntl.h>

int open(const char *path, int flags, mode_t mode);

Description
The file name specified by path is opened for reading and/or writing as specified by the argument flags and the file descriptor returned to the calling process. The flags argument may indicate the file is to be created if it does not exist (by specifying the O_CREAT flag), in which case the file is created with mode mode as described in chmod and modified by the process’ umask value (see umask).

The flags specified are formed by or ’ing the following values

- O_RDONLY        open for reading only
- O_WRONLY        open for writing only
- O_RDWR          open for reading and writing
- O_NONBLOCK      do not block on open
- O_APPEND        append on each write
- O_CREAT         create file if it does not exist
- O_TRUNC         truncate size to 0
- O_EXCL          error if create and file exists
- O_SHLOCK        atomically obtain a shared lock
- O_EXLOCK        atomically obtain an exclusive lock

Opening a file with O_APPEND set causes each write on the file to be appended to the end. If O_TRUNC is specified and the file exists, the file is truncated to zero length. If O_EXCL is set with O_CREAT and the file already exists, open returns an error. This may be used to implement a simple exclusive access locking mechanism. If O_EXCL is set and the last component of the pathname is a symbolic link, open will fail even if the symbolic link points to a non-existent name. If the O_NONBLOCK flag is specified and the open call would result in the process being blocked for some reason (e.g., waiting for carrier on a dialup line), open returns immediately. The first time the process attempts to perform I/O on the open file it will block (not currently implemented).
When opening a file, a lock with flock semantics can be obtained by setting \texttt{O\_SHLOCK} for a shared lock, or \texttt{O\_EXLOCK} for an exclusive lock. If creating a file with \texttt{O\_CREATE} the request for the lock will never fail (provided that the underlying filesystem supports locking).

If successful, \texttt{open} returns a non-negative integer, termed a file descriptor. It returns -1 on failure. The file pointer used to mark the current position within the file is set to the beginning of the file.

When a new file is created it is given the group of the directory which contains it.

The new descriptor is set to remain open across \texttt{execve} system calls; see \texttt{close} and \texttt{fcntl}.

The system imposes a limit on the number of file descriptors open simultaneously by one process. \texttt{Getdtablesize} returns the current system limit.

**Return Values**

If successful, \texttt{open} returns a non-negative integer, termed a file descriptor. It returns -1 on failure, and sets \texttt{errno} to indicate the error.

**Errors**

The named file is opened unless:

\texttt{ENOTDIR}

A component of the path prefix is not a directory.

\texttt{ENAMETOOLONG}

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

\texttt{ENOENT}

\texttt{O\_CREATE} is not set and the named file does not exist.

\texttt{ENOENT}

A component of the path name that must exist does not exist.
EACCES

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

EACCES

The required permissions (for reading and/or writing) are denied for the given flags.

EACCES

**O_CREAT** is specified, the file does not exist, and the directory in which it is
to be created does not permit writing.

ELOOP

Too many symbolic links were encountered in translating the pathname.

EISDIR

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

EROFS

The named file resides on a read-only file system, and the file is to be
modified.

EMFILE

The process has already reached its limit for open file descriptors.

ENFILE

The system file table is full.

ENXIO

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

EINTR
The open operation was interrupted by a signal.

EOPNOTSUPP

**O_SHLOCK** or **O_EXLOCK** is specified but the underlying filesystem does not support locking.

ENOSPC

**O_CREAT** is specified, the file does not exist, and the directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.

ENOSPC

**O_CREAT** is specified, the file does not exist, and there are no free inodes on the file system on which the file is being created.

EDQUOT

**O_CREAT** is specified, the file does not exist, and the directory in which the entry for the new file is being placed cannot be extended because the user’s quota of disk blocks on the file system containing the directory has been exhausted.

EDQUOT

**O_CREAT** is specified, the file does not exist, and the user’s quota of inodes on the file system on which the file is being created has been exhausted.

EIO

An I/O error occurred while making the directory entry or allocating the inode for **O_CREAT**

ETXTBSY

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

EFAULT
Path points outside the process’s allocated address space.

EEXIST

O_CREAT and O_EXCL were specified and the file exists.

EOPNOTSUPP

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

EINVAL

An attempt was made to open a descriptor with an illegal combination of O_RDONLY O_WRONLY and O_RDWR

See Also
chmod, close, dup, getdtablesize, lseek, read, umask, write
read readv---read input

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

ssize_t read(int d, void *buf, size_t nbytes);
ssize_t readv(int d, const struct iovec *iov, int iovcnt);

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

For readv, the iovec structure is defined as:

    struct iovec {
        char   *iov_base;  /* Base address. */
        size_t iov_len;    /* Length. */
    };

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

On objects capable of seeking, the read starts at a position given by the pointer associated with d (see lseek). Upon return from read, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

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

Return Values
If successful, the number of bytes actually read is returned. Upon reading end-of-file, zero is returned. Otherwise, a -1 is returned and the global variable *errno* is set to indicate the error.

Errors
Read and readv will succeed unless:

EBADF

\[ D \] is not a valid file or socket descriptor open for reading.

EFAULT

\[ Buf \] points outside the allocated address space.

EIO

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

EINTR

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

EINVAL

The pointer associated with \( d \) was negative.

EAGAIN

The file was marked for non-blocking I/O, and no data were ready to be read.

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.

EFAULT

Part of the iov points outside the process’s allocated address space.

See Also
dup, fcntl, open, pipe, select, socket, socketpair

Standards
The read function call is expected to conform to p1003.1-90.
write writev---write output

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

ssize_t write(int d, const void *buf, size_t nbytes);
ssize_t writev(int d, const struct iovec *iov, int iovcnt);

Description
Write attempts to write nbytes of data to the object referenced by the descriptor d from the buffer pointed to by buf. Writev performs the same action, but gathers the output data from the iovcnt buffers specified by the members of the iov array: iov[0], iov[1], ..., iov[iovcnt-1].

For writev, the iovec structure is defined as:

    struct iovec {
        char    *iov_base;  /* Base address. */
        size_t  iov_len;    /* Length. */
    };

Each iovec entry specifies the base address and length of an area in memory from which data should be written. Writev will always write a complete area before proceeding to the next.

On objects capable of seeking, the write starts at a position given by the pointer associated with d, see lseek. Upon return from write, the pointer is incremented by the number of bytes which were written.

Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

If the real user is not the super-user, then write clears the set-user-id bit on a file.
This prevents penetration of system security by a user who “captures” a writable set-user-id file owned by the super-user.

When using non-blocking I/O on objects such as sockets that are subject to flow control, write and writev may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

**Return Values**

Upon successful completion the number of bytes which were written is returned. Otherwise a -1 is returned and the global variable *errno* is set to indicate the error.

**Errors**

Write and writev will fail and the file pointer will remain unchanged if:

**EBADF**

\[D\] is not a valid descriptor open for writing.

**EPIPE**

An attempt is made to write to a pipe that is not open for reading by any process.

**EPIPE**

An attempt is made to write to a socket of type **SOCK_STREAM** that is not connected to a peer socket.

**EFBIG**

An attempt was made to write a file that exceeds the process’s file size limit or the maximum file size.

**EFAULT**

Part of *iov* or data to be written to the file points outside the process’s allocated address space.
The pointer associated with $d$ was negative.

**ENOSPC**

There is no free space remaining on the file system containing the file.

**EDQUOT**

The user’s quota of disk blocks on the file system containing the file has been exhausted.

**EIO**

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

**EAGAIN**

The file was marked for non-blocking I/O, and no data could be written immediately.

In addition, `writev` may return one of the following errors:

**EINVAL**

$Iovcnt$ was less than or equal to 0, or greater than `UIO_MAXIOV`

**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**

`fcntl`, `lseek`, `open`, `pipe`, `select`

**Standards**

The `write` function call is expected to conform to `p1003.1-90`. 
close---delete a descriptor

Synopsis

#include <unistd.h>

int close(int d);

Description

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

When a process exits, all associated file descriptors are freed, but since there is a limit on active descriptors per processes, the `close` function call is useful when a large quantity of file descriptors are being handled.

When a process forks (see `fork`), all descriptors for the new child process reference the same objects as they did in the parent before the fork. If a new process is then to be run using `execve`, the process would normally inherit these descriptors. Most of the descriptors can be rearranged with `dup2` or deleted with `close` before the `execve` is attempted, but if some of these descriptors will still be needed if the `execve` fails, it is necessary to arrange for them to be closed if the `execve` succeeds. For this reason, the call `fcntl(d, F_SETFD, 1)` is provided, which arranges that a descriptor will be closed after a successful `execve`; the call `fcntl(d, F_SETFD, 0)` restores the default, which is to not close the descriptor.

Return Values

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

Errors

Close will fail if:
EBADF

D is not an active descriptor.

EINTR

An interrupt was received.

See Also
accept, execve, fcntl, flock, open, pipe, socket, socketpair

Standards
The close function call is expected to conform to p1003.1-90.
lseek---reposition read/write file offset

Synopsis
#include <unistd.h>

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

Description
The lseek function repositions the offset of the file descriptor fildes to the argument offset according to the directive whence. The argument fildes must be an open file descriptor. lseek repositions the file position pointer associated with the file descriptor fildes as follows:

If

    whence is SEEK_SET the offset is set to offset bytes.

If

    whence is SEEK_CUR the offset is set to its current location plus offset bytes.

If

    whence is SEEK_END the offset is set to the size of the file plus offset bytes.

The lseek function allows the file offset to be set beyond the end of the existing end-of-file of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

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

Return Values
Upon successful completion, lseek returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of -1 is returned and errno is set to indicate the error.
Errors
Lseek will fail and the file position pointer will remain unchanged if:

EBADF

\textit{Fildes} is not an open file descriptor.

ESPIPE

\textit{Fildes} is associated with a pipe, socket, or FIFO.

EINVAL

\textit{Whence} is not a proper value.

See Also
dup, open

Bugs
This document’s use of \textit{whence} is incorrect English, but is maintained for historical reasons.

Standards
The lseek function call is expected to conform to p1003.1-90.
fcntl---file control

Synopsis
#include <fcntl.h>

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

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

**F_DUPFD**

Return a new descriptor as follows:

- Lowest numbered available descriptor greater than or equal to *arg*.
- Same object references as the original descriptor.
- New descriptor shares the same file offset if the object was a file.
- Same access mode (read, write or read/write).
- Same file status flags (i.e., both file descriptors share the same file status flags).
- The close-on-exec flag associated with the new file descriptor is set to remain open across execve system calls.

**F_GETFD**

Get the close-on-exec flag associated with the file descriptor *fd*. If the low-order bit of the returned value is 0, the file will remain open across exec, otherwise the file will be closed upon execution of exec (arg is ignored).

**F_SETFD**

Set the close-on-exec flag associated with *fd* to the low order bit of *arg* (0 or 1
F_GETFL

Get descriptor status flags, as described below (*arg* is ignored).

F_SETFL

Set descriptor status flags to *arg*.

F_GETOWN

Get the process ID or process group currently receiving *SIGIO* and *SIGURG* signals; process groups are returned as negative values (*arg* is ignored).

F_SETOWN

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

The flags for the F_GETFL and F_SETFL flags are as follows:

O_NONBLOCK

Non-blocking I/O; if no data is available to a _read_ call, or if a write operation would block, the read or write call returns -1 with the error Err EAGAIN.

O_APPEND

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

O_ASYNC

Enable the SIGIO signal to be sent to the process group when I/O is possible, e.g., upon availability of data to be read.

Several commands are available for doing advisory file locking; they all operate on the following structure:

```c
struct flock {
    off_t   l_start; /* starting offset */
};
```
The commands available for advisory record locking are as follows:

**F_GETLK**

Get the first lock that blocks the lock description pointed to by the third argument, `arg`, taken as a pointer to a `struct flock` (see above). The information retrieved overwrites the information passed to `fcntl` in the `flock` structure. If no lock is found that would prevent this lock from being created, the structure is left unchanged by this function call except for the lock type which is set to **F_UNLCK**

**F_SETLK**

Set or clear a file segment lock according to the lock description pointed to by the third argument, `arg`, taken as a pointer to a `struct flock` (see above). **F_SETLK** is used to establish shared (or read) locks (**F_RDLCK**) or exclusive (or write) locks, (**F_WRLCK**) as well as remove either type of lock (**F_UNLCK**) If a shared or exclusive lock cannot be set, `fcntl` returns immediately with Error **EAGAIN**.

**F_SETLKW**

This command is the same as **F_SETLK** except that if a shared or exclusive lock is blocked by other locks, the process waits until the request can be satisfied. If a signal that is to be caught is received while `fcntl` is waiting for a region, the `fcntl` will be interrupted if the signal handler has not specified the **SA_RESTART** (see sigaction).

When a shared lock has been set on a segment of a file, other processes can set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor was not opened with read access.

An exclusive lock prevents any other process from setting a shared lock or an
exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file was not opened with write access.

The value of $l_{\text{whence}}$ is \texttt{SEEK_SET} \texttt{SEEK_CUR} or \texttt{SEEK_END} to indicate that the relative offset, $l_{\text{start}}$ bytes, will be measured from the start of the file, current position, or end of the file, respectively. The value of $l_{\text{len}}$ is the number of consecutive bytes to be locked. If $l_{\text{len}}$ is negative, the result is undefined. The $l_{\text{pid}}$ field is only used with \texttt{F_GETLK} to return the process ID of the process holding a blocking lock. After a successful \texttt{F_GETLK} request, the value of $l_{\text{whence}}$ is \texttt{SEEK_SET}

Locks may start and extend beyond the current end of a file, but may not start or extend before the beginning of the file. A lock is set to extend to the largest possible value of the file offset for that file if $l_{\text{len}}$ is set to zero. If $l_{\text{whence}}$ and $l_{\text{start}}$ point to the beginning of the file, and $l_{\text{len}}$ is zero, the entire file is locked. If an application wishes only to do entire file locking, the \texttt{flock} system call is much more efficient.

There is at most one type of lock set for each byte in the file. Before a successful return from an \texttt{F_SETLK} or an \texttt{F_SETLKW} request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region is replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an \texttt{F_SETLK} or an \texttt{F_SETLKW} request fails or blocks respectively when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

This interface follows the completely stupid semantics of System V and p1003.1-88 that require that all locks associated with a file for a given process are removed when any file descriptor for that file is closed by that process. This semantic means that applications must be aware of any files that a subroutine library may access. For example if an application for updating the password file locks the password file database while making the update, and then calls \texttt{getpwnam} to retrieve a record, the lock will be lost because \texttt{getpwnam} opens, reads, and closes the password database. The database close will release all locks that the process has associated with the database, even if the library routine never requested a lock on the database. Another minor semantic problem with this interface is that locks are not inherited by a child process created using the \texttt{fork} function. The \texttt{flock} interface has much more rational last close semantics and allows locks to be
inherited by child processes. Flock is recommended for applications that want to ensure the integrity of their locks when using library routines or wish to pass locks to their children. Note that flock and fcntl locks may be safely used concurrently.

All locks associated with a file for a given process are removed when the process terminates.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock the locked region of another process. This implementation detects that sleeping until a locked region is unlocked would cause a deadlock and fails with an Err EDEADLK error.

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

- **F_DUPFD**
  A new file descriptor.

- **F_GETFD**
  Value of flag (only the low-order bit is defined).

- **F_GETFL**
  Value of flags.

- **F_GETOWN**
  Value of file descriptor owner.

- **other**
  Value other than -1.

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

**Errors**
Fcntl will fail if:

EAGAIN
POSIX_CORE

The argument `cmd` is **F_SETLK** the type of lock (`l_type`) is a shared lock (**F_RDLCK**) or exclusive lock (**F_WRLCK**) and the segment of a file to be locked is already exclusive-locked by another process; or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

**EBADF**

*Fildes* is not a valid open file descriptor.

The argument `cmd` is **F_SETLK** or **F_SETLKW** the type of lock (`l_type`) is a shared lock (**F_RDLCK**) and *fildes* is not a valid file descriptor open for reading.

The argument `cmd` is **F_SETLK** or **F_SETLKW** the type of lock (`l_type`) is an exclusive lock (**F_WRLCK**) and *fildes* is not a valid file descriptor open for writing.

**EDEADLK**

The argument `cmd` is **F_SETLKW** and a deadlock condition was detected.

**EINTR**

The argument `cmd` is **F_SETLKW** and the function was interrupted by a signal.

**EINVAL**

`Cmd` is **F_DUPFD** and `arg` is negative or greater than the maximum allowable number (see `getdtablesize`).

The argument `cmd` is **F_GETLK** **F_SETLK** or **F_SETLKW** and the data to which `arg` points is not valid, or *fildes* refers to a file that does not support locking.

**EMFILE**

The argument `cmd` is **F_DUPFD** and the maximum number of file descriptors permitted for the process are already in use, or no file descriptors greater than or equal to `arg` are available.
ENOLCK

The argument cmd is F_SETLK or F_SETLKW and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

ESRCH

Cmd is F_SETOWN and the process ID given as argument is not in use.

In addition, if fd refers to a descriptor open on a terminal device (as opposed to a descriptor open on a socket), a cmd of F_SETOWN can fail for the same reasons as in tcsetpgrp, and a cmd of F_GETOWN for the reasons as stated in tcgetpgrp.

See Also

close, execve, flock, getdtablesiz, open, sigvec, tcgetpgrp, tcsetpgrp
ioctl---control device

Synopsis
#include <sys/ioctl.h>

int ioctl(int d, unsigned long request, char *argp);

Description
The ioctl function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with ioctl requests. The argument d must be an open file descriptor.

An ioctl request has encoded in it whether the argument is an “in” parameter or “out” parameter, and the size of the argument argp in bytes. Macros and defines used in specifying an ioctl request are located in the file Ao Pa sys/ioctl.h Ac.

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

Errors
ioctl will fail if:

EBADF

    d is not a valid descriptor.

ENOTTY

    d is not associated with a character special device.

ENOTTY

    The specified request does not apply to the kind of object that the descriptor d references.
EINVAL

Request or argp is not valid.

See Also
mt(1), execve, fcntl, intro(4), tty(4)
creat---create a new file

Synopsis
#include <fcntl.h>

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

Description
This interface is made obsolete by: open.

Creat is the same as:

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

See Also
open
link---make a hard file link

Synopsis
#include <unistd.h>

int link(const char *name1, const char *name2);

Description
The `link` function call atomically creates the specified directory entry (hard link) `name2` with the attributes of the underlying object pointed at by `name1`. If the link is successful, the link count of the underlying object is incremented; `name1` and `name2` share equal access and rights to the underlying object.

If `name1` is removed, the file `name2` is not deleted and the link count of the underlying object is decremented.

`Name1` must exist for the hard link to succeed and both `name1` and `name2` must be in the same file system. `name1` may not be a directory.

Return Values
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:

ENOTDIR

A component of either path prefix is not a directory.

ENAMETOOLONG

A component of either pathname exceeded 255 characters, or entire length of either path name exceeded 1023 characters.

ENOENT
A component of either path prefix does not exist.

The file system containing the file named by name1 does not support links.

The link count of the file named by name1 would exceed \{LINK_MAX\}.

A component of either path prefix denies search permission.

The requested link requires writing in a directory with a mode that denies write permission.

Too many symbolic links were encountered in translating one of the pathnames.

The file named by name1 does not exist.

The link named by name2 does exist.

The file named by name1 is a directory.

The link named by name2 and the file named by name1 are on different file systems.

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

EDQUOT

The directory in which the entry for the new link is being placed cannot be extended because the user’s quota of disk blocks on the file system containing the directory has been exhausted.

EIO

An I/O error occurred while reading from or writing to the file system to make the directory entry.

EROFS

The requested link requires writing in a directory on a read-only file system.

EFAULT

One of the pathnames specified is outside the process’s allocated address space.

See Also
readlink, symlink, unlink

Standards
The link function call is expected to conform to p1003.1-90.
mkdir---make a directory file

Synopsis
#include <sys/types.h>

#include <sys/stat.h>

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

Description
The directory path is created with the access permissions specified by mode and restricted by the umask of the calling process.

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.

Return Values
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:

ENOTDIR

A component of the path prefix is not a directory.

ENAMETOOLONG

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

ENOENT

A component of the path prefix does not exist.

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

ELOOP

Too many symbolic links were encountered in translating the pathname.

EPERM

The \textit{path} argument contains a byte with the high-order bit set.

EROFS

The named file resides on a read-only file system.

EEXIST

The named file exists.

ENOSPC

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

ENOSPC

There are no free inodes on the file system on which the directory is being created.

EDQUOT

The new directory cannot be created because the user’s quota of disk blocks on the file system that will contain the directory has been exhausted.

EDQUOT

The user’s quota of inodes on the file system on which the directory is being created has been exhausted.

EIO

An I/O error occurred while making the directory entry or allocating the inode.
EIO

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

EFAULT

Path points outside the process’s allocated address space.

See Also
chmod, stat, umask

Standards
The mkdir function call is expected to conform to p1003.1-90.
unlink---remove directory entry

Synopsis
#include <unistd.h>

int unlink(const char *path);

Description
The unlink function removes the link named by path from its directory and decrements the link count of the file which was referenced by the link. If that decrement reduces the link count of the file to zero, and no process has the file open, then all resources associated with the file are reclaimed. If one or more process have the file open when the last link is removed, the link is removed, but the removal of the file is delayed until all references to it have been closed. path may not be a directory.

Return Values
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 unlink succeeds unless:

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

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

ENOENT
The named file does not exist.

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

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

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

EPERM
The named file is a directory.

EPERM
The directory containing the file is marked sticky, and neither the containing directory nor the file to be removed are owned by the effective user ID.

EBUSY
The entry to be unlinked is the mount point for a mounted file system.

EIO
An I/O error occurred while deleting the directory entry or deallocating the inode.

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

EFAULT
Path points outside the process’s allocated address space.

See Also
close, link, rmdir, symlink(7)
rmdir---remove a directory file

Synopsis
#include <unistd.h>

int rmdir(const 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 Values
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:

ENOTDIR
A component of the path is not a directory.

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

ENOENT
The named directory does not exist.

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

ENOTEMPTY
The named directory contains files other than ‘.’ and ‘..’ in it.
EACCES

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

EACCES

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

EPERM

The directory containing the directory to be removed is marked sticky, and neither the containing directory nor the directory to be removed are owned by the effective user ID.

EBUSY

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

EIO

An I/O error occurred while deleting the directory entry or deallocating the inode.

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
mkdir, unlink
rename---change the name of a file

Synopsis
#include <stdio.h>

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

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

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

If the final component of from is a symbolic link, the symbolic link is renamed, not the file or directory to which it points.

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 Values
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 argument files will be affected if:
ENAMETOOLONG
POSIX CORE

A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.

ENOENT

A component of the from path does not exist, or a path prefix of to does not exist.

EACCES

A component of either path prefix denies search permission.

EACCES

The requested link requires writing in a directory with a mode that denies write permission.

EPERM

The directory containing from is marked sticky, and neither the containing directory nor from are owned by the effective user ID.

EPERM

The to file exists, the directory containing to is marked sticky, and neither the containing directory nor to are owned by the effective user ID.

ELOOP

Too many symbolic links were encountered in translating either pathname.

ENOTDIR

A component of either path prefix is not a directory.

ENOTDIR

from is a directory, but to is not a directory.

EISDIR

to is a directory, but from is not a directory.
EXDEV

The link named by *to* and the file named by *from* are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.

ENOSPC

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

EDQUOT

The directory in which the entry for the new name is being placed cannot be extended because the user’s quota of disk blocks on the file system containing the directory has been exhausted.

EIO

An I/O error occurred while making or updating a directory entry.

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*, or an attempt is made to rename ‘.’ or ‘..’

ENOTEMPTY

*To* is a directory and is not empty.

See Also

open, symlink(7)
Standards
The `rename` function call is expected to conform to p1003.1-90.
stat, lstat, fstat---get file status

Synopsis
#include <sys/types.h>

#include <sys/stat.h>

int stat(const char *path, struct stat *sb);

int lstat(const char *path, struct stat *sb);

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

Description
The stat function obtains information about the file pointed to by path. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

lstat 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. Unlike other filesystem objects, symbolic links do not have an owner, group, access mode, times, etc. Instead, these attributes are taken from the directory that contains the link. The only attributes returned from an lstat that refer to the symbolic link itself are the file type (S_IFLNK), size, blocks, and link count (always 1).

The fstat obtains the same information about an open file known by the file descriptor fd.

The sb argument is a pointer to a stat structure as defined by sys/stat.h (shown below) and into which information is placed concerning the file.

struct stat {
    dev_t st_dev;           /* inode’s device */
    ino_t st_ino;           /* inode’s number */
    mode_t st_mode;         /* inode protection mode */
    nlink_t st_nlink;       /* number of hard links */
    uid_t st_uid;           /* user ID of the file’s owner */
}
POSIX_CORE

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

struct stat {
    gid_t st_gid;    /* group ID of the file’s group */
    dev_t st_rdev;   /* device type */

    #ifdef _POSIX_SOURCE
    struct timespec st_atimespec; /* time of last access */
    struct timespec st_mtimespec; /* time of last data modification */
    struct timespec st_ctimespec; /* time of last file status change */
    #else
    time_t st_atime;       /* time of last access */
    long st_atimensec;     /* nsec of last access */
    time_t st_mtime;       /* time of last data modification */
    long st_mtimensec;     /* nsec of last data modification */
    time_t st_ctime;       /* time of last file status change */
    long st_ctimensec;     /* nsec of last file status change */
    #endif

    off_t st_size;        /* file size, in bytes */
    quad_t st_blocks;     /* blocks allocated for file */
    u_long st_blksize;    /* optimal blocksize for I/O */
    u_long st_flags;      /* user defined flags for file */
    u_long st_gen;        /* file generation number */
};
```

The time-related fields of `struct stat` are as follows:

**st_atime**

Time when file data last accessed. Changed by the `mknod`, `utimes` and `read` system calls.

**st_mtime**

Time when file data last modified. Changed by the `mknod`, `utimes` and `write` system calls.

**st_ctime**

Time when file status was last changed (inode data modification). Changed by the `chmod`, `chown`, `link`, `mknod`, `rename`, `unlink`, `utimes` and `write` system calls.

If `_POSIX_SOURCE` is not defined, the time-related fields are defined as:

```c
#ifndef _POSIX_SOURCE
#define st_atime st_atimespec.tv_sec
#define st_mtime st_mtimespec.tv_sec
#define st_ctime st_ctimespec.tv_sec
#endif
```

The size-related fields of the `struct stat` are as follows:
**st_blksize**

The optimal I/O block size for the file.

**st_blocks**

The actual number of blocks allocated for the file in 512-byte units. As short symbolic links are stored in the inode, this number may be zero.

The status information word **st_mode** has the following bits:

```c
#define S_IFMT 0170000 /* type of file */
#define S_IFIFO 0010000 /* named pipe (fifo) */
#define S_IFCHR 0020000 /* character special */
#define S_IFDIR 0040000 /* directory */
#define S_IFBLK 0060000 /* block special */
#define S_IFREG 0100000 /* regular */
#define S_IFLNK 0120000 /* symbolic link */
#define S_IFSOCK 0140000 /* socket */
#define S_IFWHT 0160000 /* whiteout */
#define S_ISUID 0004000 /* set user id on execution */
#define S_ISGID 0002000 /* set group id on execution */
#define S_ISVTX 0001000 /* save swapped text even after use */
#define S_IRUSR 0000400 /* read permission, owner */
#define S_IWUSR 0000200 /* write permission, owner */
#define S_IXUSR 0000100 /* execute/search permission, owner */
#define S_IRGRP 0000040 /* read permission, group */
#define S_IWGRP 0000020 /* write permission, group */
#define S_IXGRP 0000010 /* execute/search permission, group */
#define S_IROTH 0000004 /* read permission, others */
#define S_IWOTH 0000002 /* write permission, others */
#define S_IXOTH 0000001 /* execute/search permission, others */
```

For a list of access modes, see `sys/stat.h`, `access` and `chmod`.

**Return Values**

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

**COMPATIBILITY**

Previous versions of the system used different types for the **st_dev**, **st_uid**, **st_gid**, **st_rdev**, **st_size**, **st_blksize** and **st_blocks** fields.

**Errors**

**Stat** and **lstat** will fail if:

**ENOTDIR**

A component of the path prefix is not a directory.
ENAMETOOLONG

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

ENOENT

The named file does not exist.

EACCES

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

ELOOP

Too many symbolic links were encountered in translating the pathname.

EFAULT

$sb$ or $name$ points to an invalid address.

EIO

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

Fstat will fail if:

EBADF

$fd$ is not a valid open file descriptor.

EFAULT

$sb$ points to an invalid address.

EIO

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

See Also

access, chmod, chown, utimes, symlink(7)
Bugs
Applying `fstat` to a socket (and thus to a pipe) returns a zeroed buffer, except for the blocksize field, and a unique device and inode number.

Standards
The `stat` and `fstat` function calls are expected to conform to p1003.1-90.
access---check access permissions of a file or pathname

Synopsis
#include <unistd.h>

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

Description
The access function checks the accessibility of the file named by path for the access permissions indicated by mode. The value of mode is the bitwise inclusive OR of the access permissions to be checked (R_OK for read permission, W_OK for write permission and X_OK for execute/search permission) or the existence test, F_OK. All components of the pathname path are checked for access permissions (including F_OK).

The real user ID is used in place of the effective user ID and the real group access list (including the real group ID) are used in place of the effective ID for verifying permission.

Even if a process has appropriate privileges and indicates success for X_OK the file may not actually have execute permission bits set. Likewise for R_OK and W_OK.

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

Errors
Access to the file is denied if:
ENOTDIR
   A component of the path prefix is not a directory.
ENAMETOOLONG
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

ENOENT

The named file does not exist.

ELOOP

Too many symbolic links were encountered in translating the pathname.

EROFS

Write access is requested for a file on a read-only file system.

ETXTBSY

Write access is requested for a pure procedure (shared text) file presently being executed.

EACCES

Permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the path prefix. The owner of a file has permission checked with respect to the ‘‘owner’’ read, write, and execute mode bits, members of the file’s group other than the owner have permission checked with respect to the ‘‘group’’ mode bits, and all others have permissions checked with respect to the ‘‘other’’ mode bits.

EFAULT

Path points outside the process’s allocated address space.

EIO

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

See Also

chmod, stat
Standards
The *access* function call is expected to conform to p1003.1-90.

Caveat
*Access* is a potential security hole and should never be used.
truncatem ftruncate---truncate or extend a file to a specified length

Synopsis
#include <unistd.h>

int truncate(const char *path, off_t length);

int ftruncate(int fd, off_t length);

Description
Truncate causes the file named by path or referenced by fd to be truncated or extended to length bytes in size. If the file was larger than this size, the extra data is lost. If the file was smaller than this size, it will be extended as if by writing bytes with the value zero. 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 succeeds unless:

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

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

ENOENT
The named file does not exist.

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

EACCES
The named file is not writable by the user.

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

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.

EIO
An I/O error occurred updating the inode.

EFAULT
Path points outside the process’s allocated address space.

Ftruncate succeeds unless:

EBADF
The fd is not a valid descriptor.

EINVAL
The fd references a socket, not a file.

EINVAL
The fd is not open for writing.
See Also

open

Bugs

These calls should be generalized to allow ranges of bytes in a file to be discarded.

Use of truncate to extend a file is not portable.
dup dup2---duplicate an existing file descriptor

Synopsis
#include <unistd.h>

int dup(int oldd);

int dup2(int oldd, int newd);

Description
Dup duplicates an existing object descriptor and returns its value to the calling process (newd = dup oldd). 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. The new descriptor returned by the call is the lowest numbered descriptor currently not in use by the process.

The object referenced by the descriptor does not distinguish between oldd and newd in any way. Thus if newd and oldd are duplicate references to an open file, read, write and lseek calls all move a single pointer into the file, and append mode, non-blocking I/O and asynchronous I/O options are shared between the references. If a separate pointer into the file is desired, a different object reference to the file must be obtained by issuing an additional open call. The close-on-exec flag on the new file descriptor is unset.

In dup2, the value of the new descriptor newd is specified. If this descriptor is already in use and oldd != newd, the descriptor is first deallocated as if a close call had been used. If oldd is not a valid descriptor, then newd is not closed. If oldd == newd and oldd is a valid descriptor, then dup2 is successful, and does nothing.

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

Errors
Dup and dup2 fail if:
EBADF

Oldd or newd is not a valid active descriptor

EMFILE

Too many descriptors are active.

See Also
accept, close, fcntl, getdtablesize, open, pipe, socket, socketpair

Standards
The dup and dup2 function calls are expected to conform to p1003.1-90.
Miscellaneous

gethostname sethostname---get/set name of current host

Synopsis
#include <unistd.h>

int gethostname(char *name, int namelen);

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

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

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

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

Errors
The following errors may be returned by these calls:

EFAULT

The name or namelen parameter gave an invalid address.

EPERM

The caller tried to set the hostname and was not the super-user.
See Also
gethostid, sysctl

Bugs
Host names are limited to MAXHOSTNAMELEN (from Ao Pa sys/param.h Ac ) characters, currently 256.
gettimeofday settimeofday---get/set date and time

Synopsis
#include <sys/time.h>

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

int settimeofday(const struct timeval *tp, const 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 tp or tzp is NULL, the associated time information will not be returned or set.

The structures pointed to by tp and tzp are defined in Ao Pa sys/time.h Ac as:

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

struct timezone {
  int tz_minuteswest; /* minutes west of Greenwich */
  int tz_dsttime; /* type of dst correction */
};

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

Only the super-user may set the time of day or time zone. If the system is running in secure mode (see init(8)), the time may only be advanced. This limitation is imposed to prevent a malicious super-user from setting arbitrary time stamps on files. The system time can still be adjusted backwards using the adjtime system
call even when the system is secure.

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

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

**EFAULT**

An argument address referenced invalid memory.

**EPERM**

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

**See Also**
date(1), adjtime, ctime, clocks(7), timed(8)
select---synchronous I/O multiplexing

Synopsis

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

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

FD_SET fd &fdset();
FD_CLR fd &fdset();
FD_ISSET fd &fdset();
FD_ZERO &fdset();

Description

Select examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and exceptfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first nfds descriptors are checked in each set; i.e., the descriptors from 0 through nfds Ns No -1 in the descriptor sets are examined. On return, select replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. Select returns the total number of ready descriptors in all the sets.

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

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

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

**Return Values**

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

**Errors**

An error return from `select` indicates:

EBADF

One of the descriptor sets specified an invalid descriptor.

EINTR

A signal was delivered before the time limit expired and before any of the selected events occurred.

EINVAL

The specified time limit is invalid. One of its components is negative or too large.

EINVAL

`nfds` was invalid.

**See Also**
accept, connect, getdtablesize, gettimeofday, read, recv, send, write, clocks(7)

Notes
The default size of **FD_SETSIZE** is currently 1024. In order to accomodate programs which might potentially use a larger number of open files with **select**, it is possible to increase this size by having the program define **FD_SETSIZE** before the inclusion of any header which includes **sys/types.h**.

If **nfds** is greater than the number of open files, **select** is not guaranteed to examine the unused file descriptors. For historical reasons, **select** will always examine the first 256 descriptors.

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

Synopsis

```
#include <unistd.h>

char * brk(const char *addr);

char * sbrk(int incr);
```

Description

The brk and sbrk functions are historical curiosities left over from earlier days before the advent of virtual memory management. The \texttt{brk} function sets the break or lowest address of a process’s data segment (uninitialized data) to \textit{addr} (immediately above bss). Data addressing is restricted between \textit{addr} and the lowest stack pointer to the stack segment. Memory is allocated by \textit{brk} in page size pieces; if \textit{addr} is not evenly divisible by the system page size, it is increased to the next page boundary.

The current value of the program break is reliably returned by \texttt{``sbrk(0)''} (see also \texttt{end}). The \texttt{getrlimit} system call may be used to determine the maximum permissible size of the \textit{data} segment; it will not be possible to set the break beyond the \textit{rlim_max} value returned from a call to \texttt{getrlimit}, e.g. \texttt{``etext + rlp-rlim_max.''} (see \texttt{end} for the definition of \textit{etext} )

Return Values

\texttt{Brk} returns 0 if successful; otherwise -1 with \textit{errno} set to indicate why the allocation failed. The \texttt{sbrk} function returns a pointer to the base of the new storage if successful; otherwise -1 with \textit{errno} set to indicate why the allocation failed.

Errors

\texttt{Brk} or \texttt{sbrk} will fail and no additional memory will be allocated if one of the following are true:

\texttt{ENOMEM}
POSIX CORE

The limit, as set by `setrlimit`, was exceeded.

ENOMEM

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

ENOMEM

Insufficient space existed in the swap area to support the expansion.

See Also
execve, getrlimit, end, malloc

Bugs
Setting the break may fail due to a temporary lack of swap space. It is not possible to distinguish this from a failure caused by exceeding the maximum size of the data segment without consulting `getrlimit`.
Chapter 2 Networking

accept---accept a connection on a socket

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

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

Description
The argument s is a socket that has been created with socket, bound to an address
with bind, and is listening for connections after a listen. The accept argument
extracts the first connection request 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 non-blocking, accept blocks the caller until a connection is
present. If the socket is marked non-blocking and no pending connections are
present on the queue, accept returns an error as described below. The accepted
socket may not be used to accept more connections. The original socket s remains
open.

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

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

For certain protocols which require an explicit confirmation, such as ISO or
DATAKIT accept can be thought of as merely dequeueing the next connection request and not implying confirmation. Confirmation can be implied by a normal read or write on the new file descriptor, and rejection can be implied by closing the new socket.

One can obtain user connection request data without confirming the connection by issuing a recvmsg call with an msg_iovlen of 0 and a non-zero msg_controllen, or by issuing a getsockopt request. Similarly, one can provide user connection rejection information by issuing a sendmsg call with providing only the control information, or by calling setsockopt.

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

Errors
The accept will fail if:

EBADF
The descriptor is invalid.

EMFILE
The per-process descriptor table is full.

ENFILE
The system file table is full.

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.
EWOULD.block

The socket is marked non-blocking and no connections are present to be accepted.

See Also
bind, connect, getpeername, listen, select, socket
bind---bind a name to a socket

Synopsis
#include <sys/types.h>

#include <sys/socket.h>

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

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

Notes
Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using unlink).

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

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

Errors
The bind call will fail if:

EBADF
   S is not a valid descriptor.

ENOTSOCK
   S is not a socket.

EADDRNOTAVAIL
The specified address is not available from the local machine.

EADDRINUSE

The specified address is already in use.

EACCES

The requested address is protected, and the current user has inadequate permission to access it.

EFAULT

The name parameter is not in a valid part of the user address space.

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

ENOTDIR

A component of the path prefix is not a directory.

EINVAL

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

ENAMETOOLONG

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

ENOENT

A prefix component of the path name does not exist.

ELOOP

Too many symbolic links were encountered in translating the pathname.

EIO

An I/O error occurred while making the directory entry or allocating the inode.

EROFS
The name would reside on a read-only file system.

EISDIR

An empty pathname was specified.

See Also
connect, getsockname, listen, socket
connect---initiate a connection on a socket

Synopsis
#include <sys/types.h>

#include <sys/socket.h>

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

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

Return Values
If the connection or binding succeeds, 0 is returned. Otherwise a \texttt{-1} is returned, and a more specific error code is stored in \texttt{errno}.

Errors
The \texttt{connect} call fails if:

\textbf{EBADF}

\texttt{s} is not a valid descriptor.

\textbf{ENOTSOCK}

\texttt{s} is a descriptor for a file, not a socket.

\textbf{EADDRNOTAVAIL}
The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain.
ENOTDIR

A component of the path prefix is not a directory.

ENAMETOOLONG

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

ENOENT

The named socket does not exist.

EACCES

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

EACCES

Write access to the named socket is denied.

ELOOP

Too many symbolic links were encountered in translating the pathname.

See Also
accept, getpeername, getsockname, select, socket
getdomainname setdomainname---get/set domain name of current host

Synopsis
#include <unistd.h>

int getdomainname(char *name, int namelen);

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

Description
Getdomainname returns the standard domain name 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 name of the host machine to be name, which has length namelen. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

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

Errors
The following errors may be returned by these calls:

EFAULT

The name or namelen parameter gave an invalid address.

EPERM

The caller tried to set the hostname and was not the super-user.
See Also
gethostid, gethostname, sysctl

Bugs
Domain names are limited to MAXHOSTNAMELEN (from Ao Pa sys/param.h) characters, currently 256.
gethostbyname gethostbyname2 gethostbyaddr
gethostent sethostent endhostent herror hstrerror--
-get network host entry

Synopsis
#include <netdb.h>

extern int h_errno;

struct hostent * gethostbyname(const char *name);

struct hostent * gethostbyname2(const char *name, int af);

struct hostent * gethostbyaddr(const char *addr, int len, int type);

struct hostent * gethostent void();

void sethostent(int stayopen);

void endhostent void();

void herror(const char *string);

const char * hstrerror(int err);

Description
The gethostbyname, gethostbyname2 and gethostbyaddr functions each return a pointer to an object with the following structure describing an internet host referenced by name or by address, respectively. This structure contains either the information obtained from the name server, named(8), or broken-out fields from a line in /etc/hosts If the local name server is not running these routines do a lookup in /etc/hosts

struct hostent {
    char    *h_name;        /* official name of host */
The members of this structure are:

- **h_name**
  
  Official name of the host.

- **h_aliases**
  
  A NULL-terminated array of alternate names for the host.

- **h_addrtype**
  
  The type of address being returned; usually **AF_INET**

- **h_length**
  
  The length, in bytes, of the address.

- **h_addr_list**
  
  A NULL-terminated array of network addresses for the host. Host addresses are returned in network byte order.

- **h_addr**
  
  The first address in **h_addr_list**; this is for backward compatibility.

When using the nameserver, `gethostbyname` and `gethostbyname` will search for the named host in the current domain and its parents unless the name ends in a dot. If the name contains no dot, and if the environment variable **`HOSTALIASES`** contains the name of an alias file, the alias file will first be searched for an alias matching the input name. See `hostname(7)` for the domain search procedure and the alias file format.

The `gethostbyname2` function is an evolution of `gethostbyname` which is
intended to allow lookups in address families other than AF_INET for example AF_INET6. Currently the af argument must be specified as AF_INET else the function will return NULL after having set h_errno to NETDB_INTERNAL.

The sethostent function may be used to request the use of a connected TCP socket for queries. If the stayopen flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to gethostbyname, gethostbyname2 or gethostbyaddr. Otherwise, queries are performed using UDP datagrams.

The endhostent function closes the TCP connection.

The herror function writes a message to the diagnostic output consisting of the string parameter s, the constant string ": ", and a message corresponding to the value of h_errno.

The hstrerror function returns a string which is the message text corresponding to the value of the err parameter.

Files

/etc/hosts

/etc/host.conf

/etc/resolv.conf

Diagnostics

Error return status from gethostbyname, gethostbyname2 and gethostbyaddr is indicated by return of a null pointer. The external integer h_errno may then be checked to see whether this is a temporary failure or an invalid or unknown host.

The routine herror can be used to print an error message describing the failure. If its argument string is non-NULL it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

The variable h_errno can have the following values:

HOST_NOT_FOUND
No such host is known.

**TRY_AGAIN**

This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.

**NO_RECOVERY**

Some unexpected server failure was encountered. This is a non-recoverable error.

**NO_DATA**

The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

**See Also**

resolver, hosts(5), hostname(7), named(8)

**Caveat**

The `gethostent` function is defined, and `sethostent` and `endhostent` are redefined, when `libc` is built to use only the routines to lookup in `/etc/hosts` and not the name server.

The `gethostent` function reads the next line of `/etc/hosts` opening the file if necessary.

The `sethostent` function opens and/or rewinds the file `/etc/hosts` If the `stayopen` argument is non-zero, the file will not be closed after each call to `gethostbyname`, `gethostbyname2` or `gethostbyaddr`.

The `endhostent` function closes the file.
Bugs
These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet address format is currently understood.
getpeername---get name of connected peer

Synopsis

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

#include <sys/socket.h>

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

Description

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

Return Values

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

Errors

The call succeeds unless:

EBADF

The argument `s` is not a valid descriptor.

ENOTSOCK

The argument `s` is a file, not a socket.

ENOTCONN

The socket is not connected.

ENOBUFFERS

Insufficient resources were available in the system to perform the operation.

EFAULT
The `name` parameter points to memory not in a valid part of the process address space.

**See Also**
accept, bind, getsockname, socket
getprotoent getprotobynumber getprotobyname
setprotoent endprotoent---get protocol entry

Synopsis
#include <netdb.h>

struct protoent * getprotoent void();

struct protoent * getprotobynumber(const char *name);

struct protoent * getprotobynumber(int proto);

void setprotoent(int stayopen);

void endprotoent void();

Description
The getprotoent, getprotobyname, and getprotobynumber functions 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

```c
struct protoent {
    char    *p_name;        /* official name of protocol */
    char    **p_aliases;    /* alias list */
    int     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.
The protocol number.

The `getprotoent` function reads the next line of the file, opening the file if necessary.

The `setprotoent` function opens and rewinds the file. If the `stayopen` flag is non-zero, the net data base will not be closed after each call to `getprotobyname` or `getprotobynumber`.

The `endprotoent` function closes the file.

The `getprotobyname` function 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 Values**
Null pointer (0) returned on `EOF` or error.

**Files**

`/etc/protocols`

**See Also**
`protocols(5)`

**Bugs**
These functions use a static data space; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet protocols are currently understood.
getservent getservbyname getservbyport setservent endservent---get service entry

Synopsis
#include <netdb.h>

struct servent * getservent();

struct servent * getservbyname(const char *name, const char *proto);

struct servent * getservbyport(int port, const char *proto);

void setservent(int stayopen);

void endservent(void);

Description
The getservent, getservbyname, and getservbyport functions 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 */
    int     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.

\texttt{s\_port}

The port number at which the service resides. Port numbers are returned in network byte order.

\texttt{s\_proto}

The name of the protocol to use when contacting the service.

The \texttt{getservent} function reads the next line of the file, opening the file if necessary.

The \texttt{setservent} function opens and rewinds the file. If the \texttt{stayopen} flag is non-zero, the net data base will not be closed after each call to \texttt{getservbyname} or \texttt{getservbyport}.

The \texttt{endservent} function closes the file.

The \texttt{getservbyname} and \texttt{getservbyport} functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until \texttt{EOF} is encountered. If a protocol name is also supplied (non-\texttt{NULL}), searches must also match the protocol.

**Files**

\texttt{/etc/services}

**Diagnostics**

Null pointer (0) returned on \texttt{EOF} or error.

**See Also**

\texttt{getprotoent}, \texttt{services(5)}

**Bugs**

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Expecting port
numbers to fit in a 32 bit quantity is probably naive.
getsockname---get socket name

Synopsis
#include <sys/types.h>

#include <sys/socket.h>

int getsockname(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
The call succeeds unless:

EBADF
The argument s is not a valid descriptor.

ENOTSOCK
The argument s is a file, not a socket.

ENOBUFS
Insufficient resources were available in the system to perform the operation.

EFAULT
The name parameter points to memory not in a valid part of the process address space.
See Also
bind, getpeername, socket

Bugs
Names bound to sockets in the UNIX domain are inaccessible; getsockname returns a zero length name.
getsockopt and setsockopt---get and set options on sockets

Synopsis
#include <sys/types.h>

#include <sys/socket.h>

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

int setsockopt(int s, int level, int optname, const void *optval, int optlen);

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

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

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

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file Ao Pa sys/socket.h Ac contains definitions for socket level options, described below. Options at other
protocol levels vary in format and name; consult the appropriate entries in section 4 of the manual.

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

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

**SO_DEBUG** -- enables recording of debugging information

**SO_REUSEADDR** -- enables local address reuse

**SO_REUSEPORT** -- enables duplicate address and port bindings

**SO_KEEPALIVE** -- enables keep connections alive

**SO_DONTROUTE** -- enables routing bypass for outgoing messages

**SO_LINGER** -- linger on close if data present

**SO_BROADCAST** -- enables permission to transmit broadcast messages

**SO_OOBINLINE** -- enables reception of out-of-band data in band

**SO_SNDBUF** -- set buffer size for output

**SO_RCVBUF** -- set buffer size for input

**SO_SNDBLOWAT** -- set minimum count for output

**SO_RCVLOWAT** -- set minimum count for input

**SO_SNDTIMEO** -- set timeout value for output

**SO_RCVTIMEO** -- set timeout value for input

**SO_TYPE** -- get the type of the socket (get only)
SO_ERROR -- get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules.

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a bind call should allow reuse of local addresses. SO_REUSEPORT allows completely duplicate bindings by multiple processes if they all set SO_REUSEPORT before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port. 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 when attempting to send data.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a close 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 seconds in the setsockopt call when SO_LINGER is requested). If SO_LINGER is disabled and a close is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv or read calls without the MSG_OOB flag. Some protocols always behave as if this option is set. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute maximum on these values, which is accessible through the sysct! MIB variable “kern.maxsockbuf”

SO_SNDCOUNT is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data
to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A `select` operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for `SO_SNDLOWAT` is set to a convenient size for network efficiency, often 1024. `SO_RCVLOWAT` is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount available or the amount requested. The default value for `SO_RCVLOWAT` is 1. If `SO_RCVLOWAT` is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that which was returned.

`SO_SNDTIMEO` is an option to set a timeout value for output operations. It accepts a `struct timeval` parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or with the error `EWOULDBLOCK` if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output. `SO_RCVTIMEO` is an option to set a timeout value for input operations. It accepts a `struct timeval` parameter with the number of seconds and microseconds used to limit waits for input operations to complete. In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error `EWOULDBLOCK` if no data were received.

Finally, `SO_TYPE` and `SO_ERROR` are options used only with `getsockopt`. `SO_TYPE` returns the type of the socket, such as `SOCK_STREAM` it is useful for servers that inherit sockets on startup. `SO_ERROR` returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.
**NETWORKING**

**Return Values**
A 0 is returned if the call succeeds, -1 if it fails.

**Errors**
The call succeeds unless:

EBADF

The argument \( s \) is not a valid descriptor.

ENOTSOCK

The argument \( s \) is a file, not a socket.

ENOPROTOOPT

The option is unknown at the level indicated.

EFAULT

The address pointed to by \( optval \) is not in a valid part of the process address space. For `getsockopt`, this error may also be returned if \( optlen \) is not in a valid part of the process address space.

**See Also**
ioctl, socket, getprotoent, syscall, protocols(5), syscall(8)

**Bugs**
Several of the socket options should be handled at lower levels of the system.
htons htonl ntohl ntohs—convert values between host and network byte order

Synopsis
#include <sys/param.h>

u_long htonl(u_long hostlong);
u_short htons(u_short hostshort);

u_long ntohl(u_long netlong);
u_short ntohs(u_short netshort);

Description
These routines convert 16 and 32 bit quantities between network byte order and host byte order. On machines which have a byte order which is the same as the network order, routines are defined as null macros.

These routines are most often used in conjunction with Internet addresses and ports as returned by gethostbyname and getservent.

See Also
gethostbyname, getservent

Bugs
On the VAX bytes are handled backwards from most everyone else in the world. This is not expected to be fixed in the near future.
inet_aton 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>

int inet_aton(const char *cp, struct in_addr *pin);
unsigned long inet_addr(const char *cp);
unsigned long inet_network(const char *cp);
char *inet_ntoa(struct in_addr in);
struct in_addr inet_makeaddr(unsigned long net, unsigned long lna);
unsigned long inet_lnaof(struct in_addr in);
unsigned long inet_netof(struct in_addr in);

Description
The routines inet_aton, inet_addr and inet_network interpret character strings representing numbers expressed in the Internet standard `.' notation. The inet_aton routine interprets the specified character string as an Internet address, placing the address into the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The inet_addr and inet_network functions return 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 addresses 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. Note that when an Internet address is viewed as a 32-bit integer quantity on the VAX the bytes referred to above appear as “d.c.b.a”. That is, VAX bytes are ordered from right to left.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as “128.net.host”.

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as “net.host”.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as “parts” in a ‘.’ notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is
The\ inet\_aton\ and\ \ inet\_ntoa\ functions\ are\ semi-deprecated\ in\ favor\ of\ the\ \ addr2ascii\ family.\ However,\ since\ those\ functions\ are\ not\ yet\ widely\ implemented,\ portable\ programs\ cannot\ rely\ on\ their\ presence\ and\ will\ continue\ to\ use\ the\ \ inet\ functions\ for\ some\ time.

**Diagnostics**

The\ constant\ INADDR\_NONE\ is\ returned\ by\ \ inet\_addr\ and\ \ inet\_network\ for\ malformed\ requests.

**See Also**

\ addr2ascii,\ gethostbyname,\ getnetent,\ hosts(5),\ networks(5)

**Bugs**

The\ value\ INADDR\_NONE\ (0xffffffff)\ is\ a\ valid\ broadcast\ address,\ but\ \ inet\_addr\ cannot\ return\ that\ value\ without\ indicating\ failure.\ The\ newer\ \ inet\_aton\ function\ does\ not\ share\ this\ problem.\ The\ problem\ of\ host\ byte\ ordering\ versus\ network\ byte\ ordering\ is\ confusing.\ The\ string\ returned\ by\ \ inet\_ntoa\ resides\ in\ a\ static\ memory\ area.

\ Inet\_addr\ should\ return\ a\ struct\ in\_addr.\n
listen---listen for connections on a socket

Synopsis
#include <sys/types.h>

#include <sys/socket.h>

int listen(int s, int backlog);

Description
To accept connections, a socket is first created with socket, a willingness to accept incoming connections and a queue limit for incoming connections are specified with listen, and then the connections are accepted with accept. The listen call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The backlog parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of Er ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

The sysctl MIB variable “kern.somaxconn” specifies a hard limit on backlog; if a value greater than kern.somaxconn or less than zero is specified, backlog is silently forced to kern.somaxconn

Return Values
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
NETWORKING

The argument s is not a socket.

EOPNOTSUPP

The socket is not of a type that supports the operation `listen`.

**See Also**

`accept`, `connect`, `socket`, `sysctl`, `sysctl(8)`
**rcmd rresvport iruserok ruserok---routines for returning a stream to a remote command**

**Synopsis**

```c
#include <unistd.h>

int rcmd(char **ahost, int inport, const char *locuser, const char *remuser, const char *cmd, int *fd2p);

int rresvport(int *port);

int iruserok(u_long raddr, int superuser, const char *ruser, const char *luser);

int ruserok(const char *rhost, int superuser, const char *ruser, const char *luser);
```

**Description**

The `rcmd` function is used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. The `rresvport` function returns a descriptor to a socket with an address in the privileged port space. The `ruserok` function is used by servers to authenticate clients requesting service with `rcmd`. All three functions are present in the same file and are used by the `rshd(8)` server (among others).

The `rcmd` function looks up the host `*ahost` using `gethostbyname`, returning -1 if the host does not exist. Otherwise `*ahost` is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port `inport`.

If the connection succeeds, a socket in the Internet domain of type `SOCK_STREAM` is returned to the caller, and given to the remote command as `stdin` and `stdout`. If `fd2p` is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in `*fd2p`. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being `UNIX` signal numbers, to be forwarded...
to the process group of the command. If fd2p is 0, then the stderr (unit 2 of the remote command) will be made the same as the stdout and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in rshd(8).

The rresvport function is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by rcmd and several other functions. Privileged Internet ports are those in the range 0 to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

The iruserok and ruserok functions take a remote host’s IP address or name, as returned by the gethostbyname routines, two user names and a flag indicating whether the local user’s name is that of the super-user. Then, if the user is NOT the super-user, it checks the /etc/hosts.equiv file. If that lookup is not done, or is unsuccessful, the .rhosts in the local user’s home directory is checked to see if the request for service is allowed. If this file does not exist, is not a regular file, is owned by anyone other than the user or the super-user, or is writable by anyone other than the owner, the check automatically fails. Zero is returned if the machine name is listed in the “hosts.equiv” file, or the host and remote user name are found in the “.rhosts” file; otherwise iruserok and ruserok return -1. If the local domain (as obtained from gethostname) is the same as the remote domain, only the machine name need be specified.

The iruserok function is strongly preferred for security reasons. It requires trusting the local DNS at most, while the ruserok function requires trusting the entire DNS, which can be spoofed.

Diagnostics
The rcmd function returns a valid socket descriptor on success. It returns -1 on error and prints a diagnostic message on the standard error.

The rresvport function returns a valid, bound socket descriptor on success. It returns -1 on error with the global value errno set according to the reason for failure. The error code EAGAIN is overloaded to mean “All network ports in use.”
See Also
rlogin(1), rsh(1), intro, rexec, rexecd(8), rlogind(8), rshd(8)
recv recvfrom recvmsg---receive a message from a socket

Synopsis

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

ssize_t recv(int s, void *buf, size_t len, int flags);

ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, int *fromlen);

ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```

Description

`recvfrom` and `recvmsg` are used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

If `from` is non-nil, and the socket is not connection-oriented, 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 `recv` call is normally used only on a connected socket (see `connect`) and is identical to `recvfrom` with a nil `from` parameter. As it is redundant, it may not be supported in future releases.

All three routines return the length of the message on successful completion. 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`).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see `fcntl`) in which case the value -1 is returned and the external variable `errno` set to `EAGAIN`. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the...
socket-level options **SO_RCVLOWAT** and **SO_RCVTIMEO** described in `getsockopt`.

The `select` call may be used to determine when more data arrive.

The `flags` argument to a `recv` call is formed by *or* ing one or more of the values:

- **MSG_OOB** -- process out-of-band data
- **MSG_PEEK** -- peek at incoming message
- **MSG_WAITALL** -- wait for full request or error

The **MSG_OOB** flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The **MSG_PEEK** flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The **MSG_WAITALL** flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned.

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 */
    u_int msg_name_len;     /* size of address */
    struct iovec *msg_iov;  /* scatter/gather array */
    u_int msg_iov_len;      /* # elements in msg_iov */
    caddr_t msg_control;    /* ancillary data, see below */
    u_int msg_control_len;  /* ancillary data buffer len */
    int msg_flags;          /* flags on received message */
};
```

Here `msg_name` and `msg_name_len` 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. `Msg_iov` and `msg_iov_len` describe scatter gather locations, as discussed
in read. *Msg_control*, which has length *msg_controllen*, points to a buffer for other protocol control related messages or other miscellaneous ancillary data. The messages are of the form:

```c
struct cmsghdr {
    u_int   cmsg_len;       /* data byte count, including hdr */
    int     cmsg_level;     /* originating protocol */
    int     cmsg_type;      /* protocol-specific type */
    u_char  cmsg_data[];    /* followed by */
};
```

As an example, one could use this to learn of changes in the data-stream in XNS/SPP, or in ISO, to obtain user-connection-request data by requesting a recvmsg with no data buffer provided immediately after an `accept` call.

Open file descriptors are now passed as ancillary data for `AF_UNIX` domain sockets, with `cmsg_level` set to `SOL_SOCKET` and `cmsg_type` set to `SCM_RIGHTS`.

Process credentials can also be passed as ancillary data for `AF_UNIX` domain sockets using a `cmsg_type` of `SCM_CREDS`. In this case, `cmsg_data` should be a structure of type `cmsgcred`, which is defined in Ao Pa `sys/socket.h` Ac as follows:

```c
struct cmsgcred {
    pid_t   cmcred_pid;             /* PID of sending process */
    uid_t   cmcred_uid;             /* real UID of sending process */
    uid_t   cmcred_euid;            /* effective UID of sending process */
    gid_t   cmcred_gid;             /* real GID of sending process */
    short   cmcred_ngroups;         /* number or groups */
    gid_t   cmcred_groups[CMGROUP_MAX];     /* groups */
};
```

The kernel will fill in the credential information of the sending process and deliver it to the receiver.

The `msg_flags` field is set on return according to the message received. *MSG_EOR* indicates end-of-record; the data returned completed a record
MSG_TRUNC indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied. MSG_CTRUNC indicates that some control data were discarded due to lack of space in the buffer for ancillary data. MSG_OOB is returned to indicate that expedited or out-of-band data were received.

**Return Values**
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.

**ENOTCONN**

The socket is associated with a connection-oriented protocol and has not been connected (see connect and accept).

**ENOTSOCK**

The argument s does not refer to a socket.

**EAGAIN**

The socket is marked non-blocking, and the receive operation would block, or a receive timeout had been set, and the timeout expired before data were received.

**EINVAL**

The receive was interrupted by delivery of a signal before any data were available.

**EFAULT**

The receive buffer pointer(s) point outside the process’s address space.
See Also
fcntl, getsockopt, read, select, socket
rexec---return stream to a remote command

Synopsis
int rexec(char **ahost, int inport, char *user, char *passwd, char *cmd, int *fd2p);

Description
The rexec function looks up the host *ahost using gethostbyname, returning -1 if
the host does not exist. Otherwise *ahost is set to the standard name of the host. If
a username and password are both specified, then these are used to authenticate to
the foreign host; otherwise the environment and then the user’s .netrc file in his
home directory are searched for appropriate information. If all this fails, the user
is prompted for the information.

The port inport specifies which well-known DARPA Internet port to use for the
connection; the call ‘getservbyname(\*qexec\*q,’\*qtcp\*q) (see getservent) will
return a pointer to a structure, which contains the necessary port. The protocol for
connection is described in detail in reexecd(8).

If the connection succeeds, a socket in the Internet domain of type
SOCK_STREAM is returned to the caller, and given to the remote command as
stdin and stdout If fd2p is non-zero, then an auxiliary channel to a control process
will be setup, and a descriptor for it will be placed in *fd2p. The control process
will return diagnostic output from the command (unit 2) on this channel, and will
also accept bytes on this channel as being UNIX signal numbers, to be forwarded
to the process group of the command. The diagnostic information returned does
not include remote authorization failure, as the secondary connection is set up
after authorization has been verified. If fd2p is 0, then the stderr (unit 2 of the
remote command) will be made the same as the stdout and no provision is made
for sending arbitrary signals to the remote process, although you may be able to
get its attention by using out-of-band data.

See Also
rcmd, reexecd(8)

Bugs
The `rexec` function sends the unencrypted password across the network.

The underlying service is considered a big security hole and therefore not enabled on many sites, see `rexecd(8)` for explanations.
send, sendto, sendmsg---send a message from a socket

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

ssize_t send(int s, const void *msg, size_t len, int flags);
ssize_t sendto(int s, const void *msg, size_t len, int flags,
const struct sockaddr *to, int tolen);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);

Description
Send, sendto, and sendmsg are used to transmit a message to another socket. Send may be used only when the socket is in a connected state, while sendto and sendmsg may be used at any time.

The address of the target is given by to with tolen specifying its size. The length of the message is given by len. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a send. Locally detected errors are indicated by a return value of -1.

If no messages space is available at the socket to hold the message to be transmitted, then send normally blocks, unless the socket has been placed in non-blocking I/O mode. The select call may be used to determine when it is possible to send more data.

The flags parameter may include one or more of the following:

#define MSG_OOB 0x1 /* process out-of-band data */
#define MSG_PEEK 0x2 /* peek at incoming message */
#define MSG_DONTROUTE   0x4   /* bypass routing, use direct interface */
#define MSG_EOR         0x8   /* data completes record */
#define MSG_EOF         0x100 /* data completes transaction */

The flag MSG_OOB is used to send ‘‘out-of-band’’ data on sockets that support this notion (e.g. SOCK_STREAM) the underlying protocol must also support ‘‘out-of-band’’ data. MSG_EOR is used to indicate a record mark for protocols which support the concept. MSG_EOF requests that the sender side of a socket be shut down, and that an appropriate indication be sent at the end of the specified data; this flag is only implemented for SOCK_STREAM sockets in the PF_INET protocol family, and is used to implement Transaction TCP (see ttcp(4)). MSG_DONTROUTE is usually used only by diagnostic or routing programs.

See recv for a description of the msghdr structure.

**Return Values**
The call returns the number of characters sent, or -1 if an error occurred.

**Errors**
Send, sendto, and sendmsg fail if:

**EBADF**
An invalid descriptor was specified.

**EACCES**
The destination address is a broadcast address, and SO_BROADCAST has not been set on the socket.

**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.

EAGAIN

The socket is marked non-blocking and the requested operation would block.

ENOBUFFS

The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

ENOBUFFS

The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

EHOSTUNREACH

The remote host was unreachable.

Bugs
Because sendmsg doesn’t necessarily block until the data has been transferred, it is possible to transfer an open file descriptor across an AF_UNIX domain socket (see recv) then close it before it has actually been sent, the result being that the receiver gets a closed file descriptor. It is left to the application to implement an acknowledgment mechanism to prevent this from happening.

See Also
fcntl, getsockopt, recv, select, socket, write
**shutdown---shut down part of a full-duplex connection**

**Synopsis**

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

#include <sys/socket.h>

int shutdown(int s, int 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, further receives will be disallowed. If `how` is 1, further sends will be disallowed. If `how` is 2, further sends and receives will be disallowed.

**Return Values**

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**
socket---create an endpoint for communication

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

int socket(int domain, int type, int protocol);

Description
Socket creates an endpoint for communication and returns a descriptor.

The domain parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in the include file <sys/socket.h>.

The currently understood formats are:

- PF_LOCAL (Host-internal protocols, formerly called PF_UNIX),
- PF_INET (ARPA Internet protocols),
- PF_ISO (ISO protocols),
- PF_CCITT (ITU-T protocols, like X.25),
- PF_NS (Xerox Network Systems protocols), and

The socket has the indicated type, which specifies the semantics of communication. Currently defined types are:

- SOCK_STREAM
- SOCK_DGRAM
- SOCK_RAW
- SOCK_SEQPACKET
- SOCK_RDM

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A
SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently implemented only for PF_NS SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW which is available only to the super-user, and SOCK_RDM which is planned, but not yet implemented, are not described here.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the “communication domain” in which communication is to take place; see protocols(5).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect call. Once connected, data may be transferred using read and write calls or some variant of the send and recv calls. (Some protocol families, such as the Internet family, support the notion of an “implied connect,” which permits data to be sent piggybacked onto a connect operation by using the sendto call.) When a session has been completed a close may be performed. Out-of-band data may also be transmitted as described in send and received as described in recv.

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets “warm” by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read calls will return only the amount of data requested, and any remaining in the arriving packet will be
discarded.

**SOCK_DGRAM** and **SOCK_RAW** sockets allow sending of datagrams to correspondents named in `send` calls. Datagrams are generally received with `recvfrom`, which returns the next datagram with its return address.

An `fcntl` call can be used to specify a process group to receive a **SIGURG** signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events via **SIGIO**.

The operation of sockets is controlled by socket level `options`. These options are defined in the file `sys/socket.h`. `setsockopt` and `getsockopt` are used to set and get options, respectively.

**Return Values**
A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

**Errors**
The `socket` call fails if:

**EPROTONOSUPPORT**

The protocol type or the specified protocol is not supported within this domain.

**EMFILE**

The per-process descriptor table is full.

**ENFILE**

The system file table is full.

**EACCES**

Permission to create a socket of the specified type and/or protocol is denied.

**ENOBUFS**

Insufficient buffer space is available. The socket cannot be created until
sufficient resources are freed.

See Also
accept, bind, connect, getpeername, getsockname, getsockopt, ioctl, listen, read, recv, select, send, shutdown, socketpair, write, getprotoent, protocols(5)

"An Introductory 4.3 BSD Interprocess Communication Tutorial" PS1 7

"BSD Interprocess Communication Tutorial" PS1 8
socketpair---create a pair of connected sockets

Synopsis
#include <sys/types.h>

#include <sys/socket.h>

int socketpair(int d, int type, int protocol, int *sv);

Description
The socketpair call creates an unnamed pair of connected sockets in the specified domain d, of the specified type, and using the optionally specified protocol. The descriptors used in referencing the new sockets are returned in sv Ns [0] and sv Ns [1]. The two sockets are indistinguishable.

Return Values
A 0 is returned if the call succeeds, -1 if it fails.

Errors
The call succeeds unless:

EMFILE
Too many descriptors are in use by this process.

EAFNOSUPPORT
The specified address family is not supported on this machine.

EPROTONOSUPPORT
The specified protocol is not supported on this machine.

EOPNOSUPPORT
The specified protocol does not support creation of socket pairs.

EFAULT
The address sv does not specify a valid part of the process address space.

See Also
pipe, read, write

Bugs
This call is currently implemented only for the UNIX domain.
Chapter 3 C Library

Standard Utility Functions

This chapter groups utility functions useful in a variety of programs. The corresponding declarations are in the header file `stdlib.h`

- **abort**: Abnormal termination of a program
- **abs**: Integer absolute value (magnitude)
- **assert**: Macro for Debugging Diagnostics
- **atexit**: Request execution of functions at program exit
- **atof**: String to double or float
- **atoi**: String to integer
- **bsearch**: Binary search
- **calloc**: Allocate space for arrays
- **div**: Divide two integers
- **ecvtbuf**: Double or float to string of digits
- **ecvt**: Double or float to string of digits (malloc result)
- **gvcvt**: Format double or float as string
- **exit**: End program execution
- **getenv**: Look up environment variable
- **labs**: Long integer absolute value (magnitude)
- **ldiv**: Divide two long integers
C LIBRARY

- **malloc**: Allocate and manage memory (malloc, realloc, free)
- **mallinfo**: Get information about allocated memory
- **_malloc_lock**: Lock memory pool for malloc and free
- **mbtowc**: Minimal multibyte to wide character converter
- **qsort**: Sort an array
- **rand**: Pseudo-random numbers
- **strtof**: String to double or float
- **strtol**: String to long
- **strtoul**: String to unsigned long
- **system**: Execute command string
- **wctomb**: Minimal wide character to multibyte converter
abort---abnormal termination of a program

Synopsis

#include <stdlib.h>
void abort(void);

Description
Use `abort` to signal that your program has detected a condition it cannot deal with. Normally, `abort` ends your program’s execution.

Before terminating your program, `abort` raises the exception SIGABRT (using ‘raise(SIGABRT)’). If you have used `signal` to register an exception handler for this condition, that handler has the opportunity to retain control, thereby avoiding program termination.

In this implementation, `abort` does not perform any stream- or file-related cleanup (the host environment may do so; if not, you can arrange for your program to do its own cleanup with a SIGABRT exception handler).

Returns
`abort` does not return to its caller.

Portability
ANSI C requires `abort`.
abs---integer absolute value (magnitude)

Synopsis

```c
#include <stdlib.h>
int abs(int i);
```

Description

`abs` returns the absolute value of `i` (also called the magnitude of `i`). That is, if `i` is negative, the result is the opposite of `i`, but if `i` is nonnegative the result is `i`.

The similar function `labs` uses and returns `long` rather than `int` values.

Returns

The result is a nonnegative integer.

Portability

`abs` is ANSI.
assert---Macro for Debugging Diagnostics

Synopsis

#include <assert.h>
void assert(int expression);

Description
Use this macro to embed debugging diagnostic statements in your programs. The argument expression should be an expression which evaluates to true (nonzero) when your program is working as you intended.

When expression evaluates to false (zero), assert calls abort, after first printing a message showing what failed and where:

  Assertion failed: expression, file filename, line lineno

The macro is defined to permit you to turn off all uses of assert at compile time by defining NDEBUG as a preprocessor variable. If you do this, the assert macro expands to

    (void(0))

Returns
assert does not return a value.

Portability
The assert macro is required by ANSI, as is the behavior when NDEBUG is defined.
atexit---request execution of functions at program exit

Synopsis

#include <stdlib.h>
int atexit(void (*function)(void);)

Description
You can use `atexit` to enroll functions in a list of functions that will be called when your program terminates normally. The argument is a pointer to a user-defined function (which must not require arguments and must not return a result).

The functions are kept in a LIFO stack; that is, the last function enrolled by `atexit` will be the first to execute when your program exits.

There is no built-in limit to the number of functions you can enroll in this list; however, after every group of 32 functions is enrolled, `atexit` will call `malloc` to get space for the next part of the list. The initial list of 32 functions is statically allocated, so you can always count on at least that many slots available.

Returns
`atexit` returns 0 if it succeeds in enrolling your function, −1 if it fails (possible only if no space was available for `malloc` to extend the list of functions).

Portability
`atexit` is required by the ANSI standard, which also specifies that implementations must support enrolling at least 32 functions.
atof, atoff---string to double or float

Synopsis

#include <stdlib.h>
double atof(const char * s);
float atoff(const char * s);

Description

atof converts the initial portion of a string to a double. atoff converts the initial portion of a string to a float.

The functions parse the character string s, locating a substring which can be converted to a floating point value. The substring must match the format:

    [+|]digits[.]?[digits] [[e|E][+|]digits]

The substring converted is the longest initial fragment of s that has the expected format, beginning with the first non-whitespace character. The substring is empty if str is empty, consists entirely of whitespace, or if the first non-whitespace character is something other than +, -, ., or a digit.

atof(s) is implemented as strtod(s, NULL). atoff(s) is implemented as strtodf(s, NULL).

Returns

atof returns the converted substring value, if any, as a double; or 0.0, if no conversion could be performed. If the correct value is out of the range of representable values, plus or minus HUGE_VAL is returned, and ERANGE is stored in errno. If the correct value would cause underflow, 0.0 is returned and ERANGE is stored in errno.

atoff obeys the same rules as atof, except that it returns a float.

Portability

atof is ANSI C. atof, atoi, and atol are subsumed by strod and strof, but are used extensively in existing code. These functions are less reliable, but may be
faster if the argument is verified to be in a valid range.
atoi, atol---string to integer

Synopsis

```c
#include <stdlib.h>
int atoi(const char * s);
long atol(const char * s);
```

Description

`atoi` converts the initial portion of a string to an `int`. `atol` converts the initial portion of a string to a `long`.

`atoi(s)` is implemented as `(int)strtol(s, NULL, 10)`. `atol(s)` is implemented as `strtol(s, NULL, 10)`.

Returns

The functions return the converted value, if any. If no conversion was made, 0 is returned.

Portability

`atoi` is ANSI.
bsearch---binary search

Synopsis

#include <stdlib.h>
void *bsearch(const void * key, const void * base,
    size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));

Description

bsearch searches an array beginning at base for any element that matches key, using binary search. nmemb is the element count of the array; size is the size of each element.

The array must be sorted in ascending order with respect to the comparison function compar (which you supply as the last argument of bsearch).

You must define the comparison function (*compar) to have two arguments; its result must be negative if the first argument is less than the second, zero if the two arguments match, and positive if the first argument is greater than the second (where "less than" and "greater than" refer to whatever arbitrary ordering is appropriate).

Returns

Returns a pointer to an element of array that matches key. If more than one matching element is available, the result may point to any of them.

Portability

bsearch is ANSI.
calloc---allocate space for arrays

Synopsis

```c
#include <stdlib.h>
void *calloc(size_t n, size_t s);
void *calloc_r(void *reent, size_t n, size_t s);
```

Description

Use `calloc` to request a block of memory sufficient to hold an array of `n` elements, each of which has size `s`.

The memory allocated by `calloc` comes out of the same memory pool used by `malloc`, but the memory block is initialized to all zero bytes. (To avoid the overhead of initializing the space, use `malloc` instead.)

The alternate function `_calloc_r` is reentrant. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

If successful, a pointer to the newly allocated space.

If unsuccessful, `NULL`.

Portability

calloc is ANSI.
**div---divide two integers**

**Synopsis**

```c
#include <stdlib.h>
div_t div(int n, int d);
```

**Description**

Divide returning quotient and remainder as two integers in a structure `div_t`.

**Returns**

The result is represented with the structure

```c
typedef struct
{
    int quot;
    int rem;
} div_t;
```

where the `quot` field represents the quotient, and `rem` the remainder. For nonzero `d`, if `r = div(n, d)`; then `n` equals `r.rem + d*r.quot`.

To divide `long` rather than `int` values, use the similar function `ldiv`.

**Portability**

`div` is ANSI.
ecvt,ecvtf,fcvt,fcvtf---double or float to string

Synopsis

```c
#include <stdlib.h>

char *ecvt(double val, int chars, int *decpt, int *sgn);
char *ecvtf(float val, int chars, int *decpt, int *sgn);

char *fcvt(double val, int decimals, int *decpt, int *sgn);
char *fcvtf(float val, int decimals, int *decpt, int *sgn);
```

Description

`ecvt` and `fcvt` produce (null-terminated) strings of digits representing the double number `val`. `ecvtf` and `fcvtf` produce the corresponding character representations of float numbers.

(The `stdlib` functions `ecvtbuf` and `fcvtbuf` are reentrant versions of `ecvt` and `fcvt`.)

The only difference between `ecvt` and `fcvt` is the interpretation of the second argument (`chars` or `decimals`). For `ecvt`, the second argument `chars` specifies the total number of characters to write (which is also the number of significant digits in the formatted string, since these two functions write only digits). For `fcvt`, the second argument `decimals` specifies the number of characters to write after the decimal point; all digits for the integer part of `val` are always included.

Since `ecvt` and `fcvt` write only digits in the output string, they record the location of the decimal point in `*decpt`, and the sign of the number in `*sgn`. After formatting a number, `*decpt` contains the number of digits to the left of the decimal point. `*sgn` contains 0 if the number is positive, and 1 if it is negative.

Returns

All four functions return a pointer to the new string containing a character representation of `val`. 
Portability
None of these functions are ANSI C.
gcvt, gcvtf---format double or float as string

Synopsis

#include <stdlib.h>

char *gcvt(double val, int precision, char *buf);
char *gcvtf(float val, int precision, char *buf);

Description
gcvt writes a fully formatted number as a null-terminated string in the buffer *buf. gcvtf produces corresponding character representations of float numbers.

gcvt uses the same rules as the printf format ‘%.precisiong’---only negative values are signed (with ‘-’), and either exponential or ordinary decimal-fraction format is chosen depending on the number of significant digits (specified by precision).

Returns
The result is a pointer to the formatted representation of val (the same as the argument buf).

Portability
Neither function is ANSI C.
ecvtbuf, fcvtbuf---double or float to string

Synopsis

#include <stdio.h>

char *ecvtbuf(double val, int chars, int *decpt, int *sgn, char *buf);

char *fcvtbuf(double val, int decimals, int *decpt, int *sgn, char *buf);

Description

ecvtbuf and fcvtbuf produce (null-terminated) strings of digits representing the double number val.

The only difference between ecvtbuf and fcvtbuf is the interpretation of the second argument (chars or decimals). For ecvtbuf, the second argument chars specifies the total number of characters to write (which is also the number of significant digits in the formatted string, since these two functions write only digits). For fcvtbuf, the second argument decimals specifies the number of characters to write after the decimal point; all digits for the integer part of val are always included.

Since ecvtbuf and fcvtbuf write only digits in the output string, they record the location of the decimal point in *decpt, and the sign of the number in *sgn. After formatting a number, *decpt contains the number of digits to the left of the decimal point. *sgn contains 0 if the number is positive, and 1 if it is negative. For both functions, you supply a pointer buf to an area of memory to hold the converted string.

Returns

Both functions return a pointer to buf, the string containing a character representation of val.

Portability

Neither function is ANSI C.
exit---end program execution

Synopsis

```c
#include <stdlib.h>
void exit(int code);
```

Description

Use `exit` to return control from a program to the host operating environment. Use the argument `code` to pass an exit status to the operating environment: two particular values, `EXIT_SUCCESS` and `EXIT_FAILURE`, are defined in `stdlib.h` to indicate success or failure in a portable fashion.

`exit` does two kinds of cleanup before ending execution of your program. First, it calls all application-defined cleanup functions you have enrolled with `atexit`. Second, files and streams are cleaned up: any pending output is delivered to the host system, each open file or stream is closed, and files created by `tmpfile` are deleted.

Returns

`exit` does not return to its caller.

Portability

ANSI C requires `exit`, and specifies that `EXIT_SUCCESS` and `EXIT_FAILURE` must be defined.
getenv---look up environment variable

Synopsis

#include <stdlib.h>
char *getenv(const char *name);

Description

getenv searches the list of environment variable names and values (using the
global pointer `char **environ`) for a variable whose name matches the string at
name. If a variable name matches, getenv returns a pointer to the associated
value.

Returns

A pointer to the (string) value of the environment variable, or NULL if there is no
such environment variable.

Portability

getenv is ANSI, but the rules for properly forming names of environment
variables vary from one system to another.

getenv requires a global pointer environ.
labs---long integer absolute value

Synopsis

#include <stdlib.h>
long labs(long i);

Description

labs returns the absolute value of i (also called the magnitude of i). That is, if i is negative, the result is the opposite of i, but if i is nonnegative the result is i.

The similar function abs uses and returns int rather than long values.

Returns

The result is a nonnegative long integer.

Portability

labs is ANSI.
ldiv---divide two long integers

Synopsis

#include <stdlib.h>
ldiv_t ldiv(long n, long d);

Description
Divide returning quotient and remainder as two long integers in a structure ldiv_t.

Returns
The result is represented with the structure

typedef struct
{
    long quot;
    long rem;
} ldiv_t;

where the quot field represents the quotient, and rem the remainder. For nonzero d, if ‘r = ldiv(n, d);’ then n equals ‘r.rem + d*r.quot’.

To divide int rather than long values, use the similar function div.

Portability
ldiv is ANSI.
malloc, realloc, free---manage memory

Synopsis

#include <stdlib.h>
void *malloc(size_t nbytes);
void *realloc(void *aptr, size_t nbytes);
void free(void *aptr);

void *memalign(size_t align, size_t nbytes);

size_t malloc_usable_size(void *aptr);

void *__malloc_r(void *reent, size_t nbytes);
void *__realloc_r(void *reent,
    void *aptr, size_t nbytes);
void __free_r(void *reent, void *aptr);

void *__memalign_r(void *reent,
    size_t align, size_t nbytes);

size_t __malloc_usable_size_r(void *reent, void *aptr);

Description
These functions manage a pool of system memory.

Use malloc to request allocation of an object with at least nbytes bytes of storage available. If the space is available, malloc returns a pointer to a newly allocated block as its result.

If you already have a block of storage allocated by malloc, but you no longer need all the space allocated to it, you can make it smaller by calling realloc with both the object pointer and the new desired size as arguments. realloc guarantees that the contents of the smaller object match the beginning of the original object.

Similarly, if you need more space for an object, use realloc to request the larger size; again, realloc guarantees that the beginning of the new, larger object matches the contents of the original object.

When you no longer need an object originally allocated by malloc or realloc (or
the related function `calloc`), return it to the memory storage pool by calling `free` with the address of the object as the argument. You can also use `realloc` for this purpose by calling it with 0 as the `nbytes` argument.

The `memalign` function returns a block of size `nbytes` aligned to a `align` boundary. The `align` argument must be a power of two.

The `malloc_usable_size` function takes a pointer to a block allocated by `malloc`. It returns the amount of space that is available in the block. This may or may not be more than the size requested from `malloc`, due to alignment or minimum size constraints.

The alternate functions `_malloc_r`, `_realloc_r`, `_free_r`, `_memalign_r`, and `_malloc_usable_size_r` are reentrant versions. The extra argument `reent` is a pointer to a reentrancy structure.

If you have multiple threads of execution which may call any of these routines, or if any of these routines may be called reentrantly, then you must provide implementations of the `__malloc_lock` and `__malloc_unlock` functions for your system. See the documentation for those functions.

These functions operate by calling the function `_sbrk_r` or `sbrk`, which allocates space. You may need to provide one of these functions for your system. `_sbrk_r` is called with a positive value to allocate more space, and with a negative value to release previously allocated space if it is no longer required. See section Definitions for OS interface.

**Returns**

`malloc` returns a pointer to the newly allocated space, if successful; otherwise it returns `NULL`. If your application needs to generate empty objects, you may use `malloc(0)` for this purpose.

`realloc` returns a pointer to the new block of memory, or `NULL` if a new block could not be allocated. `NULL` is also the result when you use `realloc(aptr,0)` (which has the same effect as `free(aptr)`). You should always check the result of `realloc`; successful reallocation is not guaranteed even when you request a smaller object.

`free` does not return a result.
memalign returns a pointer to the newly allocated space.

malloc_usable_size returns the usable size.

**Portability**

malloc, realloc, and free are specified by the ANSI C standard, but other conforming implementations of malloc may behave differently when nbytes is zero.

memalign is part of SVR4.

malloc_usable_size is not portable.
mbtowc---minimal multibyte to wide char converter

Synopsis

#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);

Description
When MB_CAPABLE is not defined, this is a minimal ANSI-conforming implementation of mbtowc. In this case, only "multi-byte character sequences" recognized are single bytes, and they are "converted" to themselves. Each call to mbtowc copies one character from *s to *pwc, unless s is a null pointer. The argument n is ignored.

When MB_CAPABLE is defined, this routine calls _mbtowc_r to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Returns
This implementation of mbtowc returns 0 if s is NULL or is the empty string; it returns 1 if not MB_CAPABLE or the character is a single-byte character; it returns -1 if n is 0 or the multi-byte character is invalid; otherwise it returns the number of bytes in the multibyte character. If the return value is -1, no changes are made to the pwc output string. If the input is the empty string, a wchar_t nul is placed in the output string and 0 is returned. If the input has a length of 0, no changes are made to the pwc output string.

Portability
mbtowc is required in the ANSI C standard. However, the precise effects vary with the locale.
qsort---sort an array

Synopsis

```c
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size, 
     int (*compar)(const void *, const void *) );
```

Description

qsort sorts an array (beginning at base) of nmemb objects. size describes the size of each element of the array.

You must supply a pointer to a comparison function, using the argument shown as compar. (This permits sorting objects of unknown properties.) Define the comparison function to accept two arguments, each a pointer to an element of the array starting at base. The result of (*compar) must be negative if the first argument is less than the second, zero if the two arguments match, and positive if the first argument is greater than the second (where "less than" and "greater than" refer to whatever arbitrary ordering is appropriate).

The array is sorted in place; that is, when qsort returns, the array elements beginning at base have been reordered.

Returns

qsort does not return a result.

Portability

qsort is required by ANSI (without specifying the sorting algorithm).
**Synopsis**

```c
#include <stdlib.h>
int rand(void);
void srand(unsigned int seed);
int rand_r(unsigned int *seed);
```

**Description**

`rand` returns a different integer each time it is called; each integer is chosen by an algorithm designed to be unpredictable, so that you can use `rand` when you require a random number. The algorithm depends on a static variable called the "random seed"; starting with a given value of the random seed always produces the same sequence of numbers in successive calls to `rand`.

You can set the random seed using `srand`; it does nothing beyond storing its argument in the static variable used by `rand`. You can exploit this to make the pseudo-random sequence less predictable, if you wish, by using some other unpredictable value (often the least significant parts of a time-varying value) as the random seed before beginning a sequence of calls to `rand`; or, if you wish to ensure (for example, while debugging) that successive runs of your program use the same "random" numbers, you can use `srand` to set the same random seed at the outset.

**Returns**

`rand` returns the next pseudo-random integer in sequence; it is a number between 0 and `RAND_MAX` (inclusive).

`srand` does not return a result.

**Portability**

`rand` is required by ANSI, but the algorithm for pseudo-random number generation is not specified; therefore, even if you use the same random seed, you cannot expect the same sequence of results on two different systems.
**strtod, strtodf---string to double or float**

**Synopsis**

```c
#include <stdlib.h>

double strtod(const char *str, char **tail);
float strtodf(const char *str, char **tail);

double _strtod_r(void *reent,
                 const char *str, char **tail);
```

**Description**

The function `strtod` parses the character string `str`, producing a substring which can be converted to a double value. The substring converted is the longest initial subsequence of `str`, beginning with the first non-whitespace character, that has the format:

```
[+-]digits[.]digits[([eE][+-]digits]
```

The substring contains no characters if `str` is empty, consists entirely of whitespace, or if the first non-whitespace character is something other than `+`, `-`, `.`, or a digit. If the substring is empty, no conversion is done, and the value of `str` is stored in `*tail`. Otherwise, the substring is converted, and a pointer to the final string (which will contain at least the terminating null character of `str`) is stored in `*tail`. If you want no assignment to `*tail`, pass a null pointer as `tail`. `strtodf` is identical to `strtod` except for its return type.

This implementation returns the nearest machine number to the input decimal string. Ties are broken by using the IEEE round-even rule.

The alternate function `_strtod_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

**Returns**

`strtod` returns the converted substring value, if any. If no conversion could be performed, 0 is returned. If the correct value is out of the range of representable values, plus or minus `HUGE_VAL` is returned, and `ERANGE` is stored in `errno`. If the correct value would cause underflow, 0 is returned and `ERANGE` is stored in `errno`. 
C LIBRARY

**strtol---string to long**

**Synopsis**

```c
#include <stdlib.h>
long strtol(const char * s, char ** ptr, int base);

long _strtol_r(void * reent, 
    const char * s, char ** ptr, int base);
```

**Description**

The function *strtol* converts the string *s* to a long. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of characters resembling an integer in the radix specified by *base*; and a trailing portion consisting of zero or more unparsable characters, and always including the terminating null character. Then, it attempts to convert the subject string into a long and returns the result.

If the value of *base* is 0, the subject string is expected to look like a normal C integer constant: an optional sign, a possible ‘0x’ indicating a hexadecimal base, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer in the radix specified by *base*, with an optional plus or minus sign. The letters a--z (or, equivalently, A--Z) are used to signify values from 10 to 35; only letters whose ascribed values are less than *base* are permitted. If *base* is 16, a leading 0x is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible letter or digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, *strtol* attempts to determine the radix from the input string. A string with a leading 0x is treated as a hexadecimal value; a string with a leading 0 and no x is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. If the subject string begins with a minus sign, the value is negated. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not NULL.
If the subject string is empty (or not in acceptable form), no conversion is performed and the value of \( s \) is stored in \( ptr \) (if \( ptr \) is not NULL).

The alternate function \_strtol\_r is a reentrant version. The extra argument \( reent \) is a pointer to a reentrancy structure.

**Returns**

\texttt{strtol} returns the converted value, if any. If no conversion was made, 0 is returned.

\texttt{strtol} returns \texttt{LONG\_MAX} or \texttt{LONG\_MIN} if the magnitude of the converted value is too large, and sets \texttt{errno} to \texttt{ERANGE}.

**Portability**

\texttt{strtol} is ANSI.
C LIBRARY

strtol---string to unsigned long

Synopsis

```c
#include <stdlib.h>
unsigned long strtol(const char *s, char **ptr, int base);
```

Description
The function `strtol` converts the string `s` to an unsigned long. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of the digits meaningful in the radix specified by `base` (for example, 0 through 7 if the value of `base` is 8); and a trailing portion consisting of one or more unparsable characters, which always includes the terminating null character. Then, it attempts to convert the subject string into an unsigned long integer, and returns the result.

If the value of `base` is zero, the subject string is expected to look like a normal C integer constant (save that no optional sign is permitted): a possible `0x` indicating hexadecimal radix, and a number. If `base` is between 2 and 36, the expected form of the subject is a sequence of digits (which may include letters, depending on the base) representing an integer in the radix specified by `base`. The letters a--z (or A--Z) are used as digits valued from 10 to 35. If `base` is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible digit, the subject string is empty.

If the subject string is acceptable, and the value of `base` is zero, `strtol` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading `0` and no `x` is treated as octal; all other strings are treated as decimal. If `base` is between 2 and 36, it is used as the conversion radix, as described above. Finally, a pointer to the first character past the converted subject string is stored in `ptr`, if `ptr` is not NULL.
If the subject string is empty (that is, if *s does not start with a substring in acceptable form), no conversion is performed and the value of *s is stored in *ptr (if *ptr is not NULL).

The alternate function _strtoul_r is a reentrant version. The extra argument *reent is a pointer to a reentrancy structure.

**Returns**

`strtoul` returns the converted value, if any. If no conversion was made, 0 is returned.

`strtoul` returns ULONG_MAX if the magnitude of the converted value is too large, and sets errno to ERANGE.

**Portability**

`strtoul` is ANSI.
system---execute command string

Synopsis

```c
#include <stdlib.h>
int system(char *s);

int _system_r(void *reent, char *s);
```

Description

Use `system` to pass a command string *s* to `/bin/sh` on your system, and wait for it to finish executing.

Use `system(NULL)` to test whether your system has `/bin/sh` available.

The alternate function `_system_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`system(NULL)` returns a non-zero value if `/bin/sh` is available, and 0 if it is not.

With a command argument, the result of `system` is the exit status returned by `/bin/sh`.

Portability

ANSI C requires `system`, but leaves the nature and effects of a command processor undefined. ANSI C does, however, specify that `system(NULL)` return zero or nonzero to report on the existence of a command processor.

POSIX.2 requires `system`, and requires that it invoke a `sh`. Where `sh` is found is left unspecified.
wctomb---minimal wide char to multibyte converter

Synopsis

```c
#include <stdlib.h>
int wctomb(char * s, wchar_t wchar);
```

Description

When MB_CAPABLE is not defined, this is a minimal ANSI-conforming implementation of `wctomb`. The only "wide characters" recognized are single bytes, and they are "converted" to themselves.

When MB_CAPABLE is defined, this routine calls `_wctomb_r` to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Each call to `wctomb` modifies `*s` unless `s` is a null pointer or MB_CAPABLE is defined and `wchar` is invalid.

Returns

This implementation of `wctomb` returns 0 if `s` is NULL; it returns −1 if MB_CAPABLE is enabled and the wchar is not a valid multi-byte character, it returns 1 if MB_CAPABLE is not defined or the wchar is in reality a single byte character, otherwise it returns the number of bytes in the multi-byte character.

Portability

`wctomb` is required in the ANSI C standard. However, the precise effects vary with the locale.
Character Type Macros and Functions

This chapter groups macros (which are also available as subroutines) to classify characters into several categories (alphabetic, numeric, control characters, whitespace, and so on), or to perform simple character mappings.

The header file `ctype.h' defines the macros.

- **isalnum**: Alphanumeric character predicate
- **isalpha**: Alphabetic character predicate
- **isascii**: ASCII character predicate
- **iscntrl**: Control character predicate
- **isdigit**: Decimal digit predicate
- **islower**: Lower-case character predicate
- **isprint**: Printable character predicates (isprint, isgraph)
- **ispunct**: Punctuation character predicate
- **isspace**: Whitespace character predicate
- **isupper**: Uppercase character predicate
- **isxdigit**: Hexadecimal digit predicate
- **toascii**: Force integers to ASCII range
- **tolower**: Translate characters to lower case
- **toupper**: Translate characters to upper case
isalnum---alphanumeric character predicate

Synopsis

```c
#include <ctype.h>
int isalnum(int c);
```

Description

`isalnum` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for alphabetic or numeric ASCII characters, and 0 for other arguments. It is defined for all integer values.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isalnum`.

Returns

`isalnum` returns non-zero if `c` is a letter (a--z or A--Z) or a digit (0--9).

Portability

`isalnum` is ANSI C.

No OS subroutines are required.
isalpha---alphabetic character predicate

Synopsis

```c
#include <ctype.h>
int isalpha(int c);
```

Description

`isalpha` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero when `c` represents an alphabetic ASCII character, and 0 otherwise. It is defined only when `isascii(c)` is true or `c` is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isalpha`.

Returns

`isalpha` returns non-zero if `c` is a letter (A--Z or a--z).

Portability

`isalpha` is ANSI C.
isascii---ASCII character predicate

Synopsis

#include <ctype.h>
int isascii(int c);

Description

isascii is a macro which returns non-zero when \( c \) is an ASCII character, and 0 otherwise. It is defined for all integer values.

You can use a compiled subroutine instead of the macro definition by undefining the macro using ‘#undef isascii’.

Returns

isascii returns non-zero if the low order byte of \( c \) is in the range 0 to 127 (0x00-0x7F).

Portability

isascii is ANSI C.
iscntrl---control character predicate

Synopsis

#include <ctype.h>
int iscntrl(int c);

Description
iscntrl is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for control characters, and 0 for other characters. It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '#undef iscntrl'.

Returns
iscntrl returns non-zero if c is a delete character or ordinary control character (0x7F or 0x00--0x1F).

Portability
iscntrl is ANSI C.
isdigit---decimal digit predicate

Synopsis

#include <ctype.h>
int isdigit(int c);

Description

isdigit is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for decimal digits, and 0 for other characters. It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using "#undef isdigit".

Returns

isdigit returns non-zero if c is a decimal digit (0--9).

Portability

isdigit is ANSI C.
C LIBRARY

islower---lower-case character predicate

Synopsis

```
#include <ctype.h>
int islower(int c);
```

Description

islower is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for minuscules (lower-case alphabetic characters), and 0 for other characters. It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using "#undef islower".

Returns

islower returns non-zero if c is a lower case letter (a--z).

Portability

islower is ANSI C.
isprint, isgraph---printable character predicates

Synopsis

#include <ctype.h>
int isprint(int c);
int isgraph(int c);

Description

isprint is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for printable characters, and 0 for other character arguments. It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining either macro using ‘#undef isprint’ or ‘#undef isgraph’.

Returns

isprint returns non-zero if c is a printing character, (0x20--0x7E). isgraph behaves identically to isprint, except that the space character (0x20) is excluded.

Portability

isprint and isgraph are ANSI C.
ispunct---punctuation character predicate

Synopsis

```c
#include <ctype.h>
int ispunct(int c);
```

Description

`ispunct` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for printable punctuation characters, and 0 for other characters. It is defined only when `isascii(c)` is true or `c` is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef ispunct`.

Returns

`ispunct` returns non-zero if `c` is a printable punctuation character (`isgraph(c) && !isalnum(c)`).

Portability

`ispunct` is ANSI C.
isspace---whitespace character predicate

Synopsis

#include <ctype.h>
int isspace(int c);

Description

isspace is a macro which classifies ASCII integer values by table lookup. It is a
predicate returning non-zero for whitespace characters, and 0 for other characters.
It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining
the macro using '#undef isspace'.

Returns

isspace returns non-zero if c is a space, tab, carriage return, new line, vertical
tab, or formfeed (0x09--0x0D, 0x20).

Portability

isspace is ANSI C.
isupper---uppercase character predicate

Synopsis

#include <ctype.h>
int isupper(int c);

Description

isupper is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for upper-case letters (A--Z), and 0 for other characters. It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '#undef isupper'.

Returns

isupper returns non-zero if c is a upper case letter (A-Z).

Portability

isupper is ANSI C.
isxdigit---hexadecimal digit predicate

Synopsis

#include <ctype.h>
int isxdigit(int c);

Description

isxdigit is a macro which classifies ASCII integer values by table lookup. It is a
predicate returning non-zero for hexadecimal digits, and 0 for other characters. It
is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining
the macro using ‘#undef isxdigit’.

Returns

isxdigit returns non-zero if c is a hexadecimal digit (0--9, a--f, or A--F).

Portability

isxdigit is ANSI C.
toascii---force integers to ASCII range

Synopsis

```c
#include <ctype.h>
int toascii(int c);
```

Description

`toascii` is a macro which coerces integers to the ASCII range (0--127) by zeroing any higher-order bits.

You can use a compiled subroutine instead of the macro definition by undefining this macro using `#undef toascii`.

Returns

`toascii` returns integers between 0 and 127.

Portability

`toascii` is not ANSI C.
tolower---translate characters to lower case

Synopsis

```c
#include <ctype.h>
int tolower(int c);
int _tolower(int c);
```

Description
tolower is a macro which converts upper-case characters to lower case, leaving all other characters unchanged. It is only defined when \( c \) is an integer in the range EOF to 255.

You can use a compiled subroutine instead of the macro definition by undefining this macro using `#undef tolower`.

(tolower) performs the same conversion as tolower, but should only be used when \( c \) is known to be an uppercase character (A--Z).

Returns
tolower returns the lower-case equivalent of \( c \) when it is a character between A and Z, and \( c \) otherwise.

(tolower) returns the lower-case equivalent of \( c \) when it is a character between A and Z. If \( c \) is not one of these characters, the behaviour of _tolower is undefined.

Portability
tolower is ANSI C. _tolower is not recommended for portable programs.
C LIBRARY

toupper---translate characters to upper case

Synopsis

#include <ctype.h>
int toupper(int c);
int _toupper(int c);

Description
toupper is a macro which converts lower-case characters to upper case, leaving all other characters unchanged. It is only defined when \textit{c} is an integer in the range EOF to 255.

You can use a compiled subroutine instead of the macro definition by undefining this macro using `#undef toupper'.

\_toupper performs the same conversion as toupper, but should only be used when \textit{c} is known to be a lowercase character (a--z).

Returns
toupper returns the upper-case equivalent of \textit{c} when it is a character between a and z, and \textit{c} otherwise.

\_toupper returns the upper-case equivalent of \textit{c} when it is a character between a and z. If \textit{c} is not one of these characters, the behaviour of \_toupper is undefined.

Portability
toupper is ANSI C. \_toupper is not recommended for portable programs.
Input and Output

This chapter comprises functions to manage files or other input/output streams. Among these functions are subroutines to generate or scan strings according to specifications from a format string.

The underlying facilities for input and output depend on the host system, but these functions provide a uniform interface.

The corresponding declarations are in `stdio.h`.

The reentrant versions of these functions use macros

```c
__stdin_r(reent)
__stdout_r(reent)
__stderr_r(reent)
```

instead of the globals `stdin`, `stdout`, and `stderr`. The argument `<reent>` is a pointer to a reentrancy structure.

- `clearerr`: Clear file or stream error indicator
- `fclose`: Close a file
- `feof`: Test for end of file
- `ferror`: Test whether read/write error has occurred
- `fflush`: Flush buffered file output
- `fgetc`: Get a character from a file or stream
- `fgetpos`: Record position in a stream or file
- `fgets`: Get character string from a file or stream
- `fiprintf`: Write formatted output to file (integer only)
- `fopen`: Open a file
- `fdopen`: Turn an open file into a stream
C LIBRARY

- **fputc**: Write a character on a stream or file
- **fputs**: Write a character string in a file or stream
- **fread**: Read array elements from a file
- **freopen**: Open a file using an existing file descriptor
- **fseek**: Set file position
- **fsetpos**: Restore position of a stream or file
- **ftell**: Return position in a stream or file
- **fwrite**: Write array elements from memory to a file or stream
- **getc**: Get a character from a file or stream (macro)
- **getchar**: Get a character from standard input (macro)
- **gets**: Get character string from standard input (obsolete)
- **iprintf**: Write formatted output (integer only)
- **mktemp**: Generate unused file name
- **perror**: Print an error message on standard error
- **putc**: Write a character on a stream or file (macro)
- **putchar**: Write a character on standard output (macro)
- **puts**: Write a character string on standard output
- **remove**: Delete a file’s name
- **rename**: Rename a file
- **rewind**: Reinitialize a file or stream
- **setbuf**: Specify full buffering for a file or stream
- **setvbuf**: Specify buffering for a file or stream
- **siprintf**: Write formatted output (integer only)
- **printf**: Write formatted output
- **scanf**: Scan and format input
- **tmpfile**: Create a temporary file
- **tmpnam**: Generate name for a temporary file
- **vprintf**: Format variable argument list
clearerr---clear file or stream error indicator

Synopsis

#include <stdio.h>
void clearerr(FILE *fp);

Description
The stdio functions maintain an error indicator with each file pointer fp, to record whether any read or write errors have occurred on the associated file or stream. Similarly, it maintains an end-of-file indicator to record whether there is no more data in the file.

Use clearerr to reset both of these indicators.

See ferror andfeof to query the two indicators.

Returns
clearerr does not return a result.

Portability
ANSI C requires clearerr.
fclose---close a file

Synopsis

#include <stdio.h>
int fclose(FILE * fp);

Description
If the file or stream identified by fp is open, fclose closes it, after first ensuring that any pending data is written (by calling fflush(fp)).

Returns
fclose returns 0 if successful (including when fp is NULL or not an open file); otherwise, it returns EOF.

Portability
fclose is required by ANSI C.

Required OS subroutines: close, fstat, isatty, lseek, read, sbrk, write.
feof---test for end of file

Synopsis

#include <stdio.h>
int feof(FILE *fp);

Description
feof tests whether or not the end of the file identified by fp has been reached.

Returns
feof returns 0 if the end of file has not yet been reached; if at end of file, the result is nonzero.

Portability
feof is required by ANSI C.
ferror---test whether read/write error has occurred

Synopsis

#include <stdio.h>
int ferror(FILE *fp);

Description
The stdio functions maintain an error indicator with each file pointer fp, to record whether any read or write errors have occurred on the associated file or stream. Use ferror to query this indicator.

See clearerr to reset the error indicator.

Returns
ferror returns 0 if no errors have occurred; it returns a nonzero value otherwise.

Portability
ANSI C requires ferror.
fflush---flush buffered file output

Synopsis

#include <stdio.h>
int fflush(FILE *fp);

Description

The stdin output functions can buffer output before delivering it to the host system, in order to minimize the overhead of system calls.

Use fflush to deliver any such pending output (for the file or stream identified by fp) to the host system.

If fp is NULL, fflush delivers pending output from all open files.

Returns

fflush returns 0 unless it encounters a write error; in that situation, it returns EOF.

Portability

ANSI C requires fflush.
fgetc---get a character from a file or stream

Synopsis

#include <stdio.h>
int fgetc(FILE *fp);

Description
Use fgetc to get the next single character from the file or stream identified by fp. As a side effect, fgetc advances the file’s current position indicator.

For a macro version of this function, see getc.

Returns
The next character (read as an unsigned char, and cast to int), unless there is no more data, or the host system reports a read error; in either of these situations, fgetc returns EOF.

You can distinguish the two situations that cause an EOF result by using the ferror and feof functions.

Portability
ANSI C requires fgetc.
fgetpos---record position in a stream or file

Synopsis

#include <stdio.h>
int fgetpos(FILE *fp, fpos_t *pos);

Description

Objects of type FILE can have a "position" that records how much of the file your program has already read. Many of the stdio functions depend on this position, and many change it as a side effect.

You can use fgetpos to report on the current position for a file identified by fp; fgetpos will write a value representing that position at *pos. Later, you can use this value with fsetpos to return the file to this position.

In the current implementation, fgetpos simply uses a character count to represent the file position; this is the same number that would be returned by ftell.

Returns

fgetpos returns 0 when successful. If fgetpos fails, the result is 1. Failure occurs on streams that do not support positioning; the global errno indicates this condition with the value ESPIPE.

Portability

fgetpos is required by the ANSI C standard, but the meaning of the value it records is not specified beyond requiring that it be acceptable as an argument to fsetpos. In particular, other conforming C implementations may return a different result from ftell than what fgetpos writes at *pos.
fgets---get character string from a file or stream

Synopsis

#include <stdio.h>
char *fgets(char *buf, int n, FILE *fp);

Description
Reads at most \( n-1 \) characters from \( fp \) until a newline is found. The characters including to the newline are stored in \( buf \). The buffer is terminated with a 0.

Returns
\texttt{fgets} returns the buffer passed to it, with the data filled in. If end of file occurs with some data already accumulated, the data is returned with no other indication. If no data are read, NULL is returned instead.

Portability
\texttt{fgets} should replace all uses of \texttt{gets}. Note however that \texttt{fgets} returns all of the data, while \texttt{gets} removes the trailing newline (with no indication that it has done so.)
fopen---open a file

Synopsis

```c
#include <stdio.h>
FILE *fopen(const char * file, const char * mode);
FILE *_fopen_r(void * reent,
               const char * file, const char * mode);
```

Description

`fopen` initializes the data structures needed to read or write a file. Specify the file's name as the string at `file`, and the kind of access you need to the file with the string at `mode`.

The alternate function `_fopen_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Three fundamental kinds of access are available: read, write, and append. `mode` must begin with one of the three characters ‘r’, ‘w’, or ‘a’, to select one of these:

`r`

Open the file for reading; the operation will fail if the file does not exist, or if the host system does not permit you to read it.

`w`

Open the file for writing from the beginning of the file: effectively, this always creates a new file. If the file whose name you specified already existed, its old contents are discarded.

`a`

Open the file for appending data, that is writing from the end of file. When you open a file this way, all data always goes to the current end of file; you cannot change this using `fseek`.

Some host systems distinguish between "binary" and "text" files. Such systems may perform data transformations on data written to, or read from, files opened as "text". If your system is one of these, then you can append a ‘b’ to any of the three
modes above, to specify that you are opening the file as a binary file (the default is to open the file as a text file).

'rb', then, means "read binary"; 'wb', "write binary"; and 'ab', "append binary".

To make C programs more portable, the 'b' is accepted on all systems, whether or not it makes a difference.

Finally, you might need to both read and write from the same file. You can also append a '+' to any of the three modes, to permit this. (If you want to append both 'b' and '+', you can do it in either order: for example, "rb+" means the same thing as "r+b" when used as a mode string.)

Use "r+" (or "rb+") to permit reading and writing anywhere in an existing file, without discarding any data; "w+" (or "wb+") to create a new file (or begin by discarding all data from an old one) that permits reading and writing anywhere in it; and "a+" (or "ab+") to permit reading anywhere in an existing file, but writing only at the end.

Returns

樾‘ fopen 梀 returns a file pointer which you can use for other file operations, unless the file you requested could not be opened; in that situation, the result is NULL. If the reason for failure was an invalid string at mode, errno is set to EINVAL.

Portability

樾‘ fopen 梀 is required by ANSI C.
fdopen---turn open file into a stream

Synopsis

```c
#include <stdio.h>
FILE *fdopen(int fd, const char *mode);
FILE *__fdopen_r(void *reent,
    int fd, const char *mode);
```

Description

`fdopen` produces a file descriptor of type `FILE *`, from a descriptor for an already-open file (returned, for example, by the system subroutine `open` rather than by `fopen`). The `mode` argument has the same meanings as in `fopen.`

Returns

File pointer or `NULL`, as for `fopen`.

Portability

`fdopen` is ANSI.
fputc---write a character on a stream or file

Synopsis

```c
#include <stdio.h>
int fputc(int ch, FILE *fp);
```

Description

_fputc_ converts the argument _ch_ from an _int_ to an _unsigned char_, then writes it to the file or stream identified by _fp_.

If the file was opened with append mode (or if the stream cannot support positioning), then the new character goes at the end of the file or stream. Otherwise, the new character is written at the current value of the position indicator, and the position indicator oadvances by one.

For a macro version of this function, see _putc_.

Returns

If successful, _fputc_ returns its argument _ch_. If an error intervenes, the result is _EOF_. You can use ‘_ferror(fp)’ to query for errors.

Portability

_fputc_ is required by ANSI C.
**Synopsis**

```c
#include <stdio.h>
int fputs(const char *s, FILE *fp);
```

**Description**

`fputs` writes the string at `s` (but without the trailing null) to the file or stream identified by `fp`.

**Returns**

If successful, the result is 0; otherwise, the result is `EOF`.

**Portability**

ANSI C requires `fputs`, but does not specify that the result on success must be 0; any non-negative value is permitted.
fread---read array elements from a file

Synopsis

```c
#include <stdio.h>
size_t fread(void *buf, size_t size, size_t count,
              FILE *fp);
```

Description

`fread` attempts to copy, from the file or stream identified by `fp`, `count` elements (each of size `size`) into memory, starting at `buf`. `fread` may copy fewer elements than `count` if an error, or end of file, intervenes.

`fread` also advances the file position indicator (if any) for `fp` by the number of `characters` actually read.

Returns

The result of `fread` is the number of elements it succeeded in reading.

Portability

ANSI C requires `fread`. 
freopen---open a file using an existing file descriptor

Synopsis

```
#include <stdio.h>
FILE *freopen(const char * file, const char * mode, FILE * fp);
```

Description

Use this variant of `fopen` if you wish to specify a particular file descriptor `fp` (notably `stdin`, `stdout`, or `stderr`) for the file.

If `fp` was associated with another file or stream, `freopen` closes that other file or stream (but ignores any errors while closing it).

`file` and `mode` are used just as in `fopen`.

Returns

If successful, the result is the same as the argument `fp`. If the file cannot be opened as specified, the result is `NULL`.

Portability

ANSI C requires `freopen`.
fseek---set file position

Synopsis

```c
#include <stdio.h>
int fseek(FILE *fp, long offset, int whence)
```

Description

Objects of type `FILE` can have a "position" that records how much of the file your program has already read. Many of the `stdio` functions depend on this position, and many change it as a side effect.

You can use `fseek` to set the position for the file identified by `fp`. The value of `offset` determines the new position, in one of three ways selected by the value of `whence` (defined as macros in `<stdio.h>`):

- **SEEK_SET---** `offset` is the absolute file position (an offset from the beginning of the file) desired. `offset` must be positive.

- **SEEK_CUR---** `offset` is relative to the current file position. `offset` can meaningfully be either positive or negative.

- **SEEK_END---** `offset` is relative to the current end of file. `offset` can meaningfully be either positive (to increase the size of the file) or negative.

See `ftell` to determine the current file position.

Returns

`fseek` returns 0 when successful. If `fseek` fails, the result is `EOF`. The reason for failure is indicated in `errno`: either `ESPIPE` (the stream identified by `fp` doesn't support repositioning) or `EINVAL` (invalid file position).

Portability

ANSI C requires `fseek`. 
fsetpos---restore position of a stream or file

Synopsis

#include <stdio.h>
int fsetpos(FILE *fp, const fpos_t *pos);

Description

Objects of type FILE can have a "position" that records how much of the file your program has already read. Many of the stdio functions depend on this position, and many change it as a side effect.

You can use fsetpos to return the file identified by fp to a previous position *pos (after first recording it with fgetpos).

See fseek for a similar facility.

Returns

fgetpos returns 0 when successful. If fgetpos fails, the result is 1. The reason for failure is indicated in errno: either EPIPE (the stream identified by fp doesn’t support repositioning) or EINVAL (invalid file position).

Portability

ANSI C requires fsetpos, but does not specify the nature of *pos beyond identifying it as written by fgetpos.
ftell---return position in a stream or file

Synopsis

#include <stdio.h>
long ftell(FILE *fp);

Description
Objects of type FILE can have a "position" that records how much of the file your program has already read. Many of the stdio functions depend on this position, and many change it as a side effect.

The result of ftell is the current position for a file identified by fp. If you record this result, you can later use it with fseek to return the file to this position.

In the current implementation, ftell simply uses a character count to represent the file position; this is the same number that would be recorded by fgetpos.

Returns
ftell returns the file position, if possible. If it cannot do this, it returns -1L. Failure occurs on streams that do not support positioning; the global errno indicates this condition with the value ESPIPE.

Portability
ftell is required by the ANSI C standard, but the meaning of its result (when successful) is not specified beyond requiring that it be acceptable as an argument to fseek. In particular, other conforming C implementations may return a different result from ftell than what fgetpos records.
fwrite---write array elements

Synopsis

#include <stdio.h>
size_t fwrite(const void * buf, size_t size, 
  size_t count, FILE * fp);

Description

fwrite attempts to copy, starting from the memory location buf, count elements 
(each of size size) into the file or stream identified by fp. fwrite may copy fewer 
elements than count if an error intervenes.

fwrite also advances the file position indicator (if any) for fp by the number of 
characters actually written.

Returns

If fwrite succeeds in writing all the elements you specify, the result is the same 
as the argument count. In any event, the result is the number of complete elements 
that fwrite copied to the file.

Portability

ANSI C requires fwrite.
getc---read a character (macro)

Synopsis

```c
#include <stdio.h>
int getc(FILE *fp);
```

Description

getc is a macro, defined in stdio.h. You can use getc to get the next single character from the file or stream identified by fp. As a side effect, getc advances the file’s current position indicator.

For a subroutine version of this macro, see fgetc.

Returns

The next character (read as an unsigned char, and cast to int), unless there is no more data, or the host system reports a read error; in either of these situations, getc returns EOF.

You can distinguish the two situations that cause an EOF result by using the ferror and feof functions.

Portability

ANSI C requires getc; it suggests, but does not require, that getc be implemented as a macro. The standard explicitly permits macro implementations of getc to use the argument more than once; therefore, in a portable program, you should not use an expression with side effects as the getc argument.
getchar---read a character (macro)

Synopsis

```
#include <stdio.h>
int getchar(void);

int _getchar_r(void *reent);
```

Description

getchar is a macro, defined in stdio.h. You can use getchar to get the next single character from the standard input stream. As a side effect, getchar advances the standard input’s current position indicator.

The alternate function _getchar_r is a reentrant version. The extra argument reent is a pointer to a reentrancy structure.

Returns

The next character (read as an unsigned char, and cast to int), unless there is no more data, or the host system reports a read error; in either of these situations, getchar returns EOF.

You can distinguish the two situations that cause an EOF result by using ‘ferror(stdin)’ and ‘feof(stdin)’.

Portability

ANSI C requires getchar; it suggests, but does not require, that getchar be implemented as a macro.
gets---get character string (obsolete, use fgets instead)

Synopsis

```c
#include <stdio.h>

char *gets(char *buf);

char *__gets_r(void *reent, char *buf);
```

Description

Reads characters from standard input until a newline is found. The characters up to the newline are stored in `buf`. The newline is discarded, and the buffer is terminated with a 0.

This is a dangerous function, as it has no way of checking the amount of space available in `buf`. One of the attacks used by the Internet Worm of 1988 used this to overrun a buffer allocated on the stack of the finger daemon and overwrite the return address, causing the daemon to execute code downloaded into it over the connection.

The alternate function `_gets_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`gets` returns the buffer passed to it, with the data filled in. If end of file occurs with some data already accumulated, the data is returned with no other indication. If end of file occurs with no data in the buffer, NULL is returned.
iprintf---write formatted output (integer only)

Synopsis

#include <stdio.h>

int iprintf(const char * format, ...);

Description

iprintf is a restricted version of printf: it has the same arguments and behavior, save that it cannot perform any floating-point formatting: the f, g, G, e, and F type specifiers are not recognized.

Returns

iprintf returns the number of bytes in the output string, save that the concluding NULL is not counted. iprintf returns when the end of the format string is encountered. If an error occurs, iprintf returns EOF.

Portability

iprintf is not required by ANSI C.
mktemp, mkstemp---generate unused file name

Synopsis

```c
#include <stdio.h>
char *mktemp(char *path);
int mkstemp(char *path);

char *__mktemp_r(void *reent, char *path);
int *__mkstemp_r(void *reent, char *path);
```

Description

`mktemp` and `mkstemp` attempt to generate a file name that is not yet in use for any existing file. `mkstemp` creates the file and opens it for reading and writing; `mktemp` simply generates the file name.

You supply a simple pattern for the generated file name, as the string at `path`. The pattern should be a valid filename (including path information if you wish) ending with some number of ‘x’ characters. The generated filename will match the leading part of the name you supply, with the trailing ‘x’ characters replaced by some combination of digits and letters.

The alternate functions `__mktemp_r` and `__mkstemp_r` are reentrant versions. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`mktemp` returns the pointer `path` to the modified string representing an unused filename, unless it could not generate one, or the pattern you provided is not suitable for a filename; in that case, it returns `NULL`.

`mkstemp` returns a file descriptor to the newly created file, unless it could not generate an unused filename, or the pattern you provided is not suitable for a filename; in that case, it returns `-1`.

Portability

ANSI C does not require either `mktemp` or `mkstemp`; the System V Interface Definition requires `mktemp` as of Issue 2.
perror---print an error message on standard error

Synopsis

```
#include <stdio.h>
void perror(char *prefix);

void _perror_r(void *reent, char *prefix);
```

Description

Use `perror` to print (on standard error) an error message corresponding to the current value of the global variable `errno`. Unless you use `NULL` as the value of the argument `prefix`, the error message will begin with the string at `prefix`, followed by a colon and a space (`:`). The remainder of the error message is one of the strings described for `strerror`.

The alternate function `_perror_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`perror` returns no result.

Portability

ANSI C requires `perror`, but the strings issued vary from one implementation to another.
putc---write a character (macro)

Synopsis

#include <stdio.h>
int putc(int ch, FILE *fp);

Description

putc is a macro, defined in stdio.h. putc writes the argument ch to the file or stream identified by fp, after converting it from an int to an unsigned char.

If the file was opened with append mode (or if the stream cannot support positioning), then the new character goes at the end of the file or stream. Otherwise, the new character is written at the current value of the position indicator, and the position indicator advances by one.

For a subroutine version of this macro, see fputc.

Returns

If successful, putc returns its argument ch. If an error intervenes, the result is EOF. You can use ‘ferror(fp)’ to query for errors.

Portability

ANSI C requires putc; it suggests, but does not require, that putc be implemented as a macro. The standard explicitly permits macro implementations of putc to use the fp argument more than once; therefore, in a portable program, you should not use an expression with side effects as this argument.
putchar---write a character (macro)

Synopsis

#include <stdio.h>
int putchar(int ch);

int _putchar_r(void * reent, int ch);

Description

putchar is a macro, defined in stdio.h. putchar writes its argument to the standard output stream, after converting it from an int to an unsigned char. The alternate function _putchar_r is a reentrant version. The extra argument reent is a pointer to a reentrancy structure.

Returns

If successful, putchar returns its argument ch. If an error intervenes, the result is EOF. You can use `ferror(stdin)` to query for errors.

Portability

ANSI C requires putchar; it suggests, but does not require, that putchar be implemented as a macro.
puts---write a character string

Synopsis

```c
#include <stdio.h>
int puts(const char * s);

int _puts_r(void * reent, const char * s);
```

Description

`puts` writes the string at `s` (followed by a newline, instead of the trailing null) to the standard output stream.

The alternate function `_puts_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

If successful, the result is a nonnegative integer; otherwise, the result is `EOF`.

Portability

ANSI C requires `puts`, but does not specify that the result on success must be 0; any non-negative value is permitted.
C LIBRARY

remove---delete a file's name

Synopsis

#include <stdio.h>
int remove(char *filename);

int _remove_r(void *reent, char *filename);

Description

Use remove to dissolve the association between a particular filename (the string at filename) and the file it represents. After calling remove with a particular filename, you will no longer be able to open the file by that name.

In this implementation, you may use remove on an open file without error; existing file descriptors for the file will continue to access the file's data until the program using them closes the file.

The alternate function _remove_r is a reentrant version. The extra argument reent is a pointer to a reentrancy structure.

Returns

remove returns 0 if it succeeds, -1 if it fails.

Portability

ANSI C requires remove, but only specifies that the result on failure be nonzero. The behavior of remove when you call it on an open file may vary among implementations.
rename---rename a file

Synopsis

```
#include <stdio.h>
int rename(const char * old, const char * new);

int _rename_r(void * reent,  
       const char * old, const char * new);
```

Description

Use `rename` to establish a new name (the string at `new`) for a file now known by
the string at `old`. After a successful `rename`, the file is no longer accessible by the
string at `old`.

If `rename` fails, the file named `*old` is unaffected. The conditions for failure
depend on the host operating system.

The alternate function `_rename_r` is a reentrant version. The extra argument `reent`
is a pointer to a reentrancy structure.

Returns

The result is either 0 (when successful) or -1 (when the file could not be
renamed).

Portability

ANSI C requires `rename`, but only specifies that the result on failure be nonzero.
The effects of using the name of an existing file as `*new` may vary from one
implementation to another.
rewind---reinitialize a file or stream

Synopsis

```c
#include <stdio.h>
void rewind(FILE *fp);
```

Description

`rewind` returns the file position indicator (if any) for the file or stream identified by `fp` to the beginning of the file. It also clears any error indicator and flushes any pending output.

Returns

`rewind` does not return a result.

Portability

ANSI C requires `rewind`. 
setbuf---specify full buffering for a file or stream

Synopsis

#include <stdio.h>
void setbuf(FILE *fp, char *buf);

Description

setbuf specifies that output to the file or stream identified by fp should be fully buffered. All output for this file will go to a buffer (of size BUFSIZ, specified in ‘stdio.h’). Output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

You may, if you wish, supply your own buffer by passing a pointer to it as the argument buf. It must have size BUFSIZ. You can also use NULL as the value of buf, to signal that the setbuf function is to allocate the buffer.

Warnings

You may only use setbuf before performing any file operation other than opening the file.

If you supply a non-null buf, you must ensure that the associated storage continues to be available until you close the stream identified by fp.

Returns

setbuf does not return a result.

Portability

Both ANSI C and the System V Interface Definition (Issue 2) require setbuf. However, they differ on the meaning of a NULL buffer pointer: the SVID issue 2 specification says that a NULL buffer pointer requests unbuffered output. For maximum portability, avoid NULL buffer pointers.
setvbuf---specify file or stream buffering

Synopsis

```c
#include <stdio.h>
int setvbuf(FILE *fp, char *buf,
    int mode, size_t size);
```

Description

Use `setvbuf` to specify what kind of buffering you want for the file or stream identified by `fp`, by using one of the following values (from `stdio.h`) as the `mode` argument:

- `_IONBF`
  Do not use a buffer: send output directly to the host system for the file or stream identified by `fp`.

- `_IOFBF`
  Use full output buffering: output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

- `_IOLBF`
  Use line buffering: pass on output to the host system at every newline, as well as when the buffer is full, or when an input operation intervenes.

Use the `size` argument to specify how large a buffer you wish. You can supply the buffer itself, if you wish, by passing a pointer to a suitable area of memory as `buf`. Otherwise, you may pass `NULL` as the `buf` argument, and `setvbuf` will allocate the buffer.

Warnings

You may only use `setvbuf` before performing any file operation other than opening the file.

If you supply a non-null `buf`, you must ensure that the associated storage continues to be available until you close the stream identified by `fp`. 
**Returns**
A 0 result indicates success, EOF failure (invalid mode or size can cause failure).

**Portability**
Both ANSI C and the System V Interface Definition (Issue 2) require `setvbuf`. However, they differ on the meaning of a `NULL` buffer pointer: the SVID issue 2 specification says that a `NULL` buffer pointer requests unbuffered output. For maximum portability, avoid `NULL` buffer pointers.

Both specifications describe the result on failure only as a nonzero value.
printf, fprintf, sprintf---format output

Synopsis

#include <stdio.h>

int printf(const char * format [, arg, ...]);
int fprintf(FILE * fd, const char * format [, arg, ...]);
int sprintf(char * str, const char * format [, arg, ...]);

Description

printf accepts a series of arguments, applies to each a format specifier from
*format, and writes the formatted data to stdout, terminated with a null character. The behavior of printf is undefined if there are not enough arguments
for the format. printf returns when it reaches the end of the format string. If
there are more arguments than the format requires, excess arguments are ignored.

fprintf and sprintf are identical to printf, other than the destination of the
formatted output: fprintf sends the output to a specified file fd, while sprintf
stores the output in the specified char array str. For sprintf, the behavior is also
undefined if the output *str overlaps with one of the arguments. format is a
pointer to a character string containing two types of objects: ordinary characters
(other than %), which are copied unchanged to the output, and conversion
specifications, each of which is introduced by %. (To include % in the output, use
%% in the format string.) A conversion specification has the following form:

%[flags][width][. prec][size][type]

The fields of the conversion specification have the following meanings:

- flags an optional sequence of characters which control output justification,
  numeric signs, decimal points, trailing zeroes, and octal and hex prefixes.
The flag characters are minus (-), plus (+), space ( ), zero (0), and sharp (#). They can appear in any combination.

  The result of the conversion is left justified, and the right is padded with blanks. If you do not use this flag, the result is right justified, and
padded on the left.

+ 

The result of a signed conversion (as determined by type) will always begin with a plus or minus sign. (If you do not use this flag, positive values do not begin with a plus sign.)

" " (space)

If the first character of a signed conversion specification is not a sign, or if a signed conversion results in no characters, the result will begin with a space. If the space ( ) flag and the plus (+) flag both appear, the space flag is ignored.

0

If the type character is d, i, o, u, x, X, e, E, f, g, or G: leading zeroes, are used to pad the field width (following any indication of sign or base); no spaces are used for padding. If the zero (0) and minus (−) flags both appear, the zero (0) flag will be ignored. For d, i, o, u, x, and x conversions, if a precision prec is specified, the zero (0) flag is ignored. Note that 0 is interpreted as a flag, not as the beginning of a field width.

#

The result is to be converted to an alternative form, according to the next character:

0

increases precision to force the first digit of the result to be a zero.

x

a non-zero result will have a 0x prefix.

X

a non-zero result will have a 0X prefix.

e, E or f
The result will always contain a decimal point even if no digits follow the point. (Normally, a decimal point appears only if a digit follows it.) Trailing zeroes are removed.

\texttt{g} or \texttt{G}

same as \texttt{e} or \texttt{E}, but trailing zeroes are not removed.

\texttt{all others}

undefined.

- \texttt{width width} is an optional minimum field width. You can either specify it directly as a decimal integer, or indirectly by using instead an asterisk (\texttt{*}), in which case an \texttt{int} argument is used as the field width. Negative field widths are not supported; if you attempt to specify a negative field width, it is interpreted as a minus (\texttt{-}) flag followed by a positive field width.

- \texttt{prec} an optional field; if present, it is introduced with \texttt{‘.‘} (a period). This field gives the maximum number of characters to print in a conversion; the minimum number of digits of an integer to print, for conversions with \texttt{type} \texttt{d}, \texttt{i}, \texttt{o}, \texttt{u}, \texttt{x}, and \texttt{X}; the maximum number of significant digits, for the \texttt{g} and \texttt{G} conversions; or the number of digits to print after the decimal point, for \texttt{e}, \texttt{E}, and \texttt{f} conversions. You can specify the precision either directly as a decimal integer or indirectly by using an asterisk (\texttt{*}), in which case an \texttt{int} argument is used as the precision. Supplying a negative precision is equivalent to omitting the precision. If only a period is specified the precision is zero. If a precision appears with any other conversion \texttt{type} than those listed here, the behavior is undefined.

- \texttt{size} \texttt{h}, \texttt{l}, and \texttt{L} are optional size characters which override the default way that \texttt{printf} interprets the data type of the corresponding argument. \texttt{h} forces the following \texttt{d}, \texttt{i}, \texttt{o}, \texttt{u}, \texttt{x} or \texttt{X} conversion \texttt{type} to apply to a \texttt{short} or \texttt{unsigned short}. \texttt{h} also forces a following \texttt{n} \texttt{type} to apply to a pointer to a \texttt{short}. Similarly, an \texttt{l} forces the following \texttt{d}, \texttt{i}, \texttt{o}, \texttt{u}, \texttt{x} or \texttt{X} conversion \texttt{type} to apply to a \texttt{long} or \texttt{unsigned long}. \texttt{l} also forces a following \texttt{n} \texttt{type} to apply to a pointer to a \texttt{long}. If an \texttt{h} or an \texttt{l} appears with another conversion specifier, the behavior is undefined. \texttt{L} forces a following \texttt{e}, \texttt{E}, \texttt{f}, \texttt{g} or \texttt{G} conversion \texttt{type} to apply to a \texttt{long double} argument. If \texttt{L} appears with any other conversion \texttt{type}, the behavior is
undefined.

- *type* specifies what kind of conversion `printf` performs. Here is a table of these:

  %
  
  prints the percent character (%)

  c
  
  prints *arg* as single character

  s
  
  prints characters until precision is reached or a null terminator is encountered; takes a string pointer

  d
  
  prints a signed decimal integer; takes an `int` (same as i)

  i
  
  prints a signed decimal integer; takes an `int` (same as d)

  o
  
  prints a signed octal integer; takes an `int`

  u
  
  prints an unsigned decimal integer; takes an `int`

  x
  
  prints an unsigned hexadecimal integer (using `abcdef` as digits beyond 9); takes an `int`

  X
  
  prints an unsigned hexadecimal integer (using `ABCDEF` as digits beyond 9); takes an `int`
prints a signed value of the form \([-\]9999.9999; takes a floating point number

\texttt{e}

prints a signed value of the form \([-\]9.9999e[+|−]999; takes a floating point number

\texttt{E}

prints the same way as \texttt{e}, but using \texttt{E} to introduce the exponent; takes a floating point number

\texttt{g}

prints a signed value in either \texttt{f} or \texttt{e} form, based on given value and precision--trailing zeros and the decimal point are printed only if necessary; takes a floating point number

\texttt{G}

prints the same way as \texttt{g}, but using \texttt{E} for the exponent if an exponent is needed; takes a floating point number

\texttt{n}

stores (in the same object) a count of the characters written; takes a pointer to \texttt{int}

\texttt{p}

prints a pointer in an implementation-defined format. This implementation treats the pointer as an \texttt{unsigned long} (same as \texttt{Lu}).

**Returns**

\texttt{sprintf} returns the number of bytes in the output string, save that the concluding \texttt{NULL} is not counted. \texttt{printf} and \texttt{fprintf} return the number of characters transmitted. If an error occurs, \texttt{printf} and \texttt{fprintf} return \texttt{EOF}. No error returns occur for \texttt{sprintf}. 

Portability
The ANSI C standard specifies that implementations must support at least formatted output of up to 509 characters.
**C LIBRARY**

**scanf, fscanf, sscanf---scan and format input**

**Synopsis**

```c
#include <stdio.h>

int scanf(const char *format [, arg, ...]);
int fscanf(FILE *fd, const char *format [, arg, ...]);
int sscanf(const char *str, const char *format [, arg, ...]);
```

**Description**

`scanf` scans a series of input fields from standard input, one character at a time. Each field is interpreted according to a format specifier passed to `scanf` in the format string at `*format`. `scanf` stores the interpreted input from each field at the address passed to it as the corresponding argument following `format`. You must supply the same number of format specifiers and address arguments as there are input fields.

There must be sufficient address arguments for the given format specifiers; if not the results are unpredictable and likely disastrous. Excess address arguments are merely ignored.

`scanf` often produces unexpected results if the input diverges from an expected pattern. Since the combination of `gets` or `fgets` followed by `sscanf` is safe and easy, that is the preferred way to be certain that a program is synchronized with input at the end of a line.

`fscanf` and `sscanf` are identical to `scanf`, other than the source of input: `fscanf` reads from a file, and `sscanf` from a string.

The string at `*format` is a character sequence composed of zero or more directives. Directives are composed of one or more whitespace characters, non-whitespace characters, and format specifications.

Whitespace characters are blank ( ), tab (\t), or newline (\n). When `scanf` encounters a whitespace character in the format string it will read (but not store) all consecutive whitespace characters up to the next non-whitespace character in the input.
Non-whitespace characters are all other ASCII characters except the percent sign (%). When `scanf` encounters a non-whitespace character in the format string it will read, but not store a matching non-whitespace character.

Format specifications tell `scanf` to read and convert characters from the input field into specific types of values, and store them in the locations specified by the address arguments.

Trailing whitespace is left unread unless explicitly matched in the format string.

The format specifiers must begin with a percent sign (%) and have the following form:

```
%[*][width][size]type
```

Each format specification begins with the percent character (%). The other fields are:

* an optional marker; if present, it suppresses interpretation and assignment of this input field.

**width**

an optional maximum field width: a decimal integer, which controls the maximum number of characters that will be read before converting the current input field. If the input field has fewer than width characters, `scanf` reads all the characters in the field, and then proceeds with the next field and its format specification. If a whitespace or a non-convertable character occurs before width character are read, the characters up to that character are read, converted, and stored. Then `scanf` proceeds to the next format specification.

**size**

h, l, and L are optional size characters which override the default way that `scanf` interprets the data type of the corresponding argument.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Type(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>d, i, o, u, x</td>
<td>convert input to short, store in short object</td>
</tr>
<tr>
<td>Type</td>
<td>Conversion Characters</td>
<td>Effect</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------</td>
<td>--------</td>
</tr>
<tr>
<td>h</td>
<td>D, I, O, U, X</td>
<td>no effect</td>
</tr>
<tr>
<td></td>
<td>e, f, c, s, n, p</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>d, i, o, u, x</td>
<td>convert input to long, store in long object</td>
</tr>
<tr>
<td></td>
<td>e, f, g</td>
<td>convert input to double, store in a double object</td>
</tr>
<tr>
<td>l</td>
<td>D, I, O, U, X</td>
<td>no effect</td>
</tr>
<tr>
<td></td>
<td>c, s, n, p</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>d, i, o, u, x</td>
<td>convert to long double, store in long double</td>
</tr>
<tr>
<td></td>
<td>all others</td>
<td>no effect</td>
</tr>
</tbody>
</table>

**Type**

A character to specify what kind of conversion `scanf` performs. Here is a table of the conversion characters:

- **%**
  - No conversion is done; the percent character (%) is stored.
- **c**
  - Scans one character. Corresponding `arg`: (char *arg).
- **s**
  - Reads a character string into the array supplied. Corresponding `arg`: (char *arg[ ]).
- **[pattern]**
  - Reads a non-empty character string into memory starting at `arg`. This area must be large enough to accept the sequence and a terminating null character which will be added automatically. (`pattern` is discussed in the paragraph following this table). Corresponding `arg`: (char *arg).
- **d**
  - Reads a decimal integer into the corresponding `arg`: (int *arg).
Reads a decimal integer into the corresponding arg: (long *arg).

o

Reads an octal integer into the corresponding arg: (int *arg).

O

Reads an octal integer into the corresponding arg: (long *arg).

u

Reads an unsigned decimal integer into the corresponding arg: (unsigned int *arg).

U

Reads an unsigned decimal integer into the corresponding arg: (unsigned long *arg).

x, X

Read a hexadecimal integer into the corresponding arg: (int *arg).

e, f, g

Read a floating point number into the corresponding arg: (float *arg).

E, F, G

Read a floating point number into the corresponding arg: (double *arg).

i

Reads a decimal, octal or hexadecimal integer into the corresponding arg: (int *arg).

I

Reads a decimal, octal or hexadecimal integer into the corresponding arg: (long *arg).

n
C LIBRARY

Stores the number of characters read in the corresponding arg: (int *arg).

Stores a scanned pointer. ANSI C leaves the details to each implementation; this implementation treats %p exactly the same as %u. Corresponding arg: (void **arg).

A pattern of characters surrounded by square brackets can be used instead of the s type character. pattern is a set of characters which define a search set of possible characters making up the scanf input field. If the first character in the brackets is a caret (^), the search set is inverted to include all ASCII characters except those between the brackets. There is also a range facility which you can use as a shortcut. %[0-9] matches all decimal digits. The hyphen must not be the first or last character in the set. The character prior to the hyphen must be lexically less than the character after it. Here are some pattern examples:

%[abcd]

matches strings containing only a, b, c, and d.

%[^abcd]

matches strings containing any characters except a, b, c, or d

%[A-DW-Z]

matches strings containing A, B, C, D, W, X, Y, Z

%[z-a]

matches the characters z, -, and a

Floating point numbers (for field types e, f, g, E, F, G) must correspond to the following general form:

\[ [+/-] dddd[.]ddd \[E|e[+|-]ddd \]

where objects inclosed in square brackets are optional, and ddd represents decimal, octal, or hexadecimal digits.

Returns
\texttt{scanf} returns the number of input fields successfully scanned, converted and stored; the return value does not include scanned fields which were not stored.

If \texttt{scanf} attempts to read at end-of-file, the return value is \texttt{EOF}.

If no fields were stored, the return value is 0.

\texttt{scanf} might stop scanning a particular field before reaching the normal field end character, or may terminate entirely.

\texttt{scanf} stops scanning and storing the current field and moves to the next input field (if any) in any of the following situations:

- The assignment suppressing character (*) appears after the \% in the format specification; the current input field is scanned but not stored.

- width characters have been read (width is a width specification, a positive decimal integer).

- The next character read cannot be converted under the current format (for example, if a \texttt{Z} is read when the format is decimal).

- The next character in the input field does not appear in the search set (or does appear in the inverted search set).

When \texttt{scanf} stops scanning the current input field for one of these reasons, the next character is considered unread and used as the first character of the following input field, or the first character in a subsequent read operation on the input.

\texttt{scanf} will terminate under the following circumstances:

- The next character in the input field conflicts with a corresponding non-whitespace character in the format string.

- The next character in the input field is E\texttt{OF}.

- The format string has been exhausted.

When the format string contains a character sequence that is not part of a format specification, the same character sequence must appear in the input; \texttt{scanf} will scan but not store the matched characters. If a conflict occurs, the first conflicting
C LIBRARY

character remains in the input as if it had never been read.

Portability

scanf is ANSI C.
tmpfile---create a temporary file

Synopsis

```c
#include <stdio.h>
FILE *tmpfile(void);
FILE *_tmpfile_r(void *reent);
```

Description

Create a temporary file (a file which will be deleted automatically), using a name generated by `tmpnam`. The temporary file is opened with the mode "wb+", permitting you to read and write anywhere in it as a binary file (without any data transformations the host system may perform for text files).

The alternate function `_tmpfile_r` is a reentrant version. The argument `reent` is a pointer to a reentrancy structure.

Returns

`tmpfile` normally returns a pointer to the temporary file. If no temporary file could be created, the result is NULL, and `errno` records the reason for failure.

Portability

Both ANSI C and the System V Interface Definition (Issue 2) require `tmpfile`. `tmpfile` also requires the global pointer `environ`.
tmpnam, tempnam---name for a temporary file

Synopsis

#include <stdio.h>
char *tmpnam(char *s);
char *tempnam(char *dir, char *pfx);
char *__tmpnam_r(void *reent, char *s);
char *__tempnam_r(void *reent, char *dir, char *pfx);

Description

Use either of these functions to generate a name for a temporary file. The generated name is guaranteed to avoid collision with other files (for up to TMP_MAX calls of either function).

tmpnam generates file names with the value of _P_tmpdir (defined in ‘stdio.h’) as the leading directory component of the path.

You can use the tmpnam argument s to specify a suitable area of memory for the generated filename; otherwise, you can call tmpnam(NULL) to use an internal static buffer.

tempnam allows you more control over the generated filename: you can use the argument dir to specify the path to a directory for temporary files, and you can use the argument pfx to specify a prefix for the base filename.

If dir is NULL, tempnam will attempt to use the value of environment variable TMPDIR instead; if there is no such value, tempnam uses the value of _P_tmpdir (defined in ‘stdio.h’).

If you don’t need any particular prefix to the basename of temporary files, you can pass NULL as the pfx argument to tempnam.

__tmpnam_r and __tempnam_r are reentrant versions of tmpnam and tempnam respectively. The extra argument reent is a pointer to a reentrancy structure.

Warnings

The generated filenames are suitable for temporary files, but do not in themselves
make files temporary. Files with these names must still be explicitly removed when you no longer want them.

If you supply your own data area $s$ for `tmpnam`, you must ensure that it has room for at least $L_{tmpnam}$ elements of type `char`.

**Returns**
Both `tmpnam` and `tempnam` return a pointer to the newly generated filename.

**Portability**
ANSI C requires `tmpnam`, but does not specify the use of `P_tmpdir`. The System V Interface Definition (Issue 2) requires both `tmpnam` and `tempnam`.

The global pointer `environ` is also required.
vprintf, vfprintf, vsprintf---format argument list

Synopsis

#include <stdio.h>
#include <stdarg.h>
int vprintf(const char * fmt, va_list list);
int vfprintf(FILE * fp, const char * fmt, va_list list);
int vsprintf(char * str, const char * fmt, va_list list);

int _vprintf_r(void * reent, const char * fmt, va_list list);
int _vfprintf_r(void * reent, FILE * fp, const char * fmt, va_list list);
int _vsprintf_r(void * reent, char * str, const char * fmt, va_list list);

Description

vprintf, vfprintf, and vsprintf are (respectively) variants of printf, fprintf, and sprintf. They differ only in allowing their caller to pass the variable argument list as a va_list object (initialized by va_start) rather than directly accepting a variable number of arguments.

Returns

The return values are consistent with the corresponding functions: vsprintf returns the number of bytes in the output string, save that the concluding NULL is not counted. vprintf and vfprintf return the number of characters transmitted. If an error occurs, vprintf and vfprintf return EOF. No error returns occur for vsprintf.

Portability

ANSI C requires all three functions.
Strings and Memory

This chapter describes string-handling functions and functions for managing areas of memory. The corresponding declarations are in ‘string.h’.

- `bcmp`: Compare two memory areas
- `bcopy`: Copy memory regions
- `bzero`: Initialize memory to zero
- `index`: Search for character in string
- `memchr`: Find character in memory
- `memcmp`: Compare two memory areas
- `memcpy`: Copy memory regions
- `memmove`: Move possibly overlapping memory
- `memset`: Set an area of memory
- `rindex`: Reverse search for character in string
- `strcat`: Concatenate strings
- `strchr`: Search for character in string
- `strcmp`: Character string compare
- `strcoll`: Locale specific character string compare
- `strcpy`: Copy string
- `strcspn`: Count chars not in string
- `strerror`: Convert error number to string
- `strlen`: Character string length
C LIBRARY

- `strlwr`: Convert string to lower case
- `strncat`: Concatenate strings
- `strncpy`: Character string compare
- `strncpy`: Counted copy string
- `strpbrk`: Find chars in string
- `strrchr`: Reverse search for character in string
- `strspn`: Find initial match
- `strstr`: Find string segment
- `strtok`: Get next token from a string
- `strupr`: Convert string to upper case
- `strxfrm`: Transform string
**bcmp---compare two memory areas**

**Synopsis**

```c
#include <string.h>
int bcmp(const char *s1, const char *s2, size_t n);
```

**Description**
This function compares not more than \( n \) characters of the object pointed to by \( s1 \) with the object pointed to by \( s2 \).

This function is identical to `memcmp`.

**Returns**
The function returns an integer greater than, equal to or less than zero according to whether the object pointed to by \( s1 \) is greater than, equal to or less than the object pointed to by \( s2 \).

**Portability**
bcopy---copy memory regions

Synopsis

#include <string.h>
void bcopy(const char *in, char *out, size_t n);

Description
This function copies n bytes from the memory region pointed to by in to the
memory region pointed to by out.

This function is implemented in term of memmove.

Portability
**bzero---initialize memory to zero**

**Synopsis**

```c
#include <string.h>
void bzero(char *b, size_t length);
```

**Description**

*bzero* initializes *length* bytes of memory, starting at address *b*, to zero.

**Returns**

*bzero* does not return a result.

**Portability**

*bzero* is in the Berkeley Software Distribution. Neither ANSI C nor the System V Interface Definition (Issue 2) require *bzero*.
index---search for character in string

Synopsis

```
#include <string.h>
char * index(const char *string, int c);
```

Description
This function finds the first occurrence of `c` (converted to a char) in the string pointed to by `string` (including the terminating null character).

This function is identical to `strchr`.

Returns
Returns a pointer to the located character, or a null pointer if `c` does not occur in `string`.

Portability
memchr---find character in memory

Synopsis

#include <string.h>
void *memchr(const void *src, int c, size_t length);

Description
This function searches memory starting at *src for the character c. The search only ends with the first occurrence of c, or after length characters; in particular, NULL does not terminate the search.

Returns
If the character c is found within length characters of *src, a pointer to the character is returned. If c is not found, then NULL is returned.

Portability
memchr> is ANSI C.
**Synopsis**

```c
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

**Description**

This function compares not more than \( n \) characters of the object pointed to by \( s1 \) with the object pointed to by \( s2 \).

**Returns**

The function returns an integer greater than, equal to or less than zero according to whether the object pointed to by \( s1 \) is greater than, equal to or less than the object pointed to by \( s2 \).

**Portability**

`memcmp` is ANSI C.
memcpy---copy memory regions

Synopsis

#include <string.h>
void* memcpy(void *out, const void *in, size_t n);

Description
This function copies \( n \) bytes from the memory region pointed to by \( in \) to the memory region pointed to by \( out \).

If the regions overlap, the behavior is undefined.

Returns

memcpy returns a pointer to the first byte of the \( out \) region.

Portability

memcpy is ANSI C.
memmove---move possibly overlapping memory

Synopsis

#include <string.h>
void *memmove(void *dst, const void *src, size_t length);

Description
This function moves length characters from the block of memory starting at *src
to the memory starting at *dst. memmove reproduces the characters correctly at
*dst even if the two areas overlap.

Returns
The function returns dst as passed.

Portability
memmove is ANSI C.
memset---set an area of memory

Synopsis

```c
#include <string.h>
void *memset(const void * dst, int c, size_t length);
```

Description

This function converts the argument `c` into an unsigned char and fills the first `length` characters of the array pointed to by `dst` to the value.

Returns

`memset` returns the value of `m`.

Portability

`memset` is ANSI C.
rindex---reverse search for character in string

Synopsis

```c
#include <string.h>
char * rindex(const char *string, int c);
```

Description
This function finds the last occurrence of `c` (converted to a char) in the string pointed to by `string` (including the terminating null character).

This function is identical to `strrchr`.

Returns
Returns a pointer to the located character, or a null pointer if `c` does not occur in `string`.

Portability
**strcat---concatenate strings**

**Synopsis**

```c
#include <string.h>
char *strcat(char *dst, const char *src);
```

**Description**

`strcat` appends a copy of the string pointed to by `src` (including the terminating null character) to the end of the string pointed to by `dst`. The initial character of `src` overwrites the null character at the end of `dst`.

**Returns**

This function returns the initial value of `dst`.

**Portability**

`strcat` is ANSI C.
strchr---search for character in string

Synopsis

#include <string.h>
char * strchr(const char *string, int c);

Description
This function finds the first occurrence of `c` (converted to a char) in the string pointed to by `string` (including the terminating null character).

Returns
Returns a pointer to the located character, or a null pointer if `c` does not occur in `string`.

Portability
`strchr` is ANSI C.
strcm---character string compare

Synopsis

#include <string.h>
int strcmp(const char *a, const char *b);

Description

strcmp compares the string at a to the string at b.

Returns

If *a sorts lexicographically after *b, strcmp returns a number greater than zero.
If the two strings match, strcmp returns zero. If *a sorts lexicographically before *b, strcmp returns a number less than zero.

Portability

strcmp is ANSI C.
strcoll—locale specific character string compare

Synopsis

```c
#include <string.h>
int strcoll(const char *stra, const char * strb);
```

Description
strcoll compares the string pointed to by stra to the string pointed to by strb, using an interpretation appropriate to the current LC_COLLATE state.

Returns
If the first string is greater than the second string, `strcoll` returns a number greater than zero. If the two strings are equivalent, `strcoll` returns zero. If the first string is less than the second string, `strcoll` returns a number less than zero.

Portability
`strcoll` is ANSI C.


strcpy---copy string

Synopsis

#include <string.h>
char *strcpy(char *dst, const char *src);

Description

strcpy copies the string pointed to by src (including the terminating null character) to the array pointed to by dst.

Returns

This function returns the initial value of dst.

Portability

strcpy is ANSI C.
**C LIBRARY**

**strcspn---count chars not in string**

**Synopsis**

```c
size_t strcspn(const char *s1, const char *s2);
```

**Description**

This function computes the length of the initial part of the string pointed to by `s1` which consists entirely of characters *NOT* from the string pointed to by `s2` (excluding the terminating null character).

**Returns**

`strcspn` returns the length of the substring found.

**Portability**

`strcspn` is ANSI C.
strerror---convert error number to string

Synopsis

#include <string.h>
char *strerror(int errnum);

Description

`strerror` converts the error number `errnum` into a string. The value of `errnum` is usually a copy of `errno`. If `errnum` is not a known error number, the result points to an empty string.

This implementation of `strerror` prints out the following strings for each of the values defined in ‘`errno.h`’:

E2BIG

Arg list too long

EACCES

Permission denied

EADV

Advertise error

EAGAIN

No more processes

EBADF

Bad file number

EBADMSG

Bad message

EBUSY

Device or resource busy
ECHILD
No children

ECOMM
Communication error

EDEADLK
Deadlock

EEXIST
File exists

EDOM
Math argument

EFAULT
Bad address

EFBIG
File too large

EIDRM
Identifier removed

EINTR
Interrupted system call

EINVAL
Invalid argument

EIO
I/O error

EISDIR
Is a directory
ELIBACC

Cannot access a needed shared library

ELIBBAD

Accessing a corrupted shared library

ELIBEXEC

Cannot exec a shared library directly

ELIBMAX

Attempting to link in more shared libraries than system limit

ELIBSCN

.lib section in a.out corrupted

EMFILE

Too many open files

EMLINK

Too many links

EMULTIHOP

Multihop attempted

ENAMETOOLONG

File or path name too long

ENFILE

Too many open files in system

ENODEV

No such device

ENOENT

No such file or directory
C LIBRARY

ENOEXEC

Exec format error

ENOLCK

No lock

ENOLINK

Virtual circuit is gone

ENOMEM

Not enough space

ENOMSG

No message of desired type

ENONET

Machine is not on the network

ENOPKG

No package

ENOSPC

No space left on device

ENOSR

No stream resources

ENOSTR

Not a stream

ENOSYS

Function not implemented

ENOTBLK

Block device required
ENOTDIR

    Not a directory

ENOTEMPTY

    Directory not empty

ENOTTY

    Not a character device

ENXIO

    No such device or address

EPERM

    Not owner

EPIPE

    Broken pipe

EPROTO

    Protocol error

ERANGE

    Result too large

EREMOTE

    Resource is remote

EROFS

    Read-only file system

ESPIPE

    Illegal seek

ESRCH

    No such process
C LIBRARY

Srmount error
ETIME
Stream ioctl timeout
ETXTBSY
Text file busy
EXDEV
Cross-device link

Returns
This function returns a pointer to a string. Your application must not modify that string.

Portability
ANSI C requires `strerror`, but does not specify the strings used for each error number.

Although this implementation of `strerror` is reentrant, ANSI C declares that subsequent calls to `strerror` may overwrite the result string; therefore portable code cannot depend on the reentrancy of this subroutine.

This implementation of `strerror` provides for user-defined extensibility. `errno.h` defines `__ELASTERROR`, which can be used as a base for user-defined error values. If the user supplies a routine named `_user_strerror`, and `errnum` passed to `strerror` does not match any of the supported values, `_user_strerror` is called with `errnum` as its argument.

`_user_strerror` takes one argument of type `int`, and returns a character pointer. If `errnum` is unknown to `_user_strerror`, `_user_strerror` returns `NULL`. The default `_user_strerror` returns `NULL` for all input values.
strlen---character string length

Synopsis

#include <string.h>
size_t strlen(const char *str);

Description
The strlen function works out the length of the string starting at *str by counting characters until it reaches a NULL character.

Returns
strlen returns the character count.

Portability
strlen is ANSI C.
strlwr---force string to lower case

Synopsis

#include <string.h>
char *strlwr(char *a);

Description

strlwr converts each character in the string at a to lower case.

Returns

strlwr returns its argument, a.

Portability

strlwr is not widely portable.
**strncat---concatenate strings**

**Synopsis**

```c
#include <string.h>
char *strncat(char *dst, const char *src, size_t length);
```

**Description**

`strncat` appends not more than `length` characters from the string pointed to by `src` (including the terminating null character) to the end of the string pointed to by `dst`. The initial character of `src` overwrites the null character at the end of `dst`. A terminating null character is always appended to the result.

**Warnings**

Note that a null is always appended, so that if the copy is limited by the `length` argument, the number of characters appended to `dst` is `n + 1`.

**Returns**

This function returns the initial value of `dst`.

**Portability**

`strncat` is ANSI C.
strncmp---character string compare

Synopsis

#include <string.h>
int strncmp(const char *a, const char * b, size_t length);

Description

strncmp compares up to length characters from the string at a to the string at b.

Returns

If *a sorts lexicographically after *b, strncmp returns a number greater than zero. If the two strings are equivalent, strncmp returns zero. If *a sorts lexicographically before *b, strncmp returns a number less than zero.

Portability

strncmp is ANSI C.
**strncpy---counted copy string**

**Synopsis**

```c
#include <string.h>
char *strncpy(char *dst, const char *src, size_t length);
```

**Description**

`strncpy` copies not more than `length` characters from the string pointed to by `src` (including the terminating null character) to the array pointed to by `dst`. If the string pointed to by `src` is shorter than `length` characters, null characters are appended to the destination array until a total of `length` characters have been written.

**Returns**

This function returns the initial value of `dst`.

**Portability**

`strncpy` is ANSI C.
strpbrk---find chars in string

Synopsis

#include <string.h>
char *strpbrk(const char *s1, const char *s2);

Description
This function locates the first occurrence in the string pointed to by \texttt{s1} of any character in string pointed to by \texttt{s2} (excluding the terminating null character).

Returns
\texttt{strpbrk} returns a pointer to the character found in \texttt{s1}, or a null pointer if no character from \texttt{s2} occurs in \texttt{s1}.

Portability
**strrchr---reverse search for character in string**

**Synopsis**

```c
#include <string.h>
char * strrchr(const char *string, int c);
```

**Description**

This function finds the last occurrence of \( c \) (converted to a char) in the string pointed to by `string` (including the terminating null character).

**Returns**

Returns a pointer to the located character, or a null pointer if \( c \) does not occur in `string`.

**Portability**

`strrchr` is ANSI C.
strspn---find initial match

Synopsis

```c
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Description
This function computes the length of the initial segment of the string pointed to by `s1` which consists entirely of characters from the string pointed to by `s2` (excluding the terminating null character).

Returns
`strspn` returns the length of the segment found.

Portability
`strspn` is ANSI C.
strstr---find string segment

Synopsis

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Description
Locates the first occurrence in the string pointed to by `s1` of the sequence of characters in the string pointed to by `s2` (excluding the terminating null character).

Returns
Returns a pointer to the located string segment, or a null pointer if the string `s2` is not found. If `s2` points to a string with zero length, the `s1` is returned.

Portability

`strstr` is ANSI C.
C LIBRARY

strtok---get next token from a string

Synopsis

#include <string.h>
char *strtok(char *source, const char *delimiters)
char *strtok_r(char *source, const char *delimiters, 
               char **lasts)

Description

The `strtok` function is used to isolate sequential tokens in a null-terminated string, `*source`. These tokens are delimited in the string by at least one of the characters in `*delimiters`. The first time that `strtok` is called, `*source` should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead. The separator string, `*delimiters`, must be supplied each time, and may change between calls.

The `strtok` function returns a pointer to the beginning of each subsequent token in the string, after replacing the separator character itself with a NUL character. When no more tokens remain, a null pointer is returned.

The `strtok_r` function has the same behavior as `strtok`, except a pointer to placeholder `*lasts` must be supplied by the caller.

Returns

`strtok` returns a pointer to the next token, or NULL if no more tokens can be found.

Portability

`strtok` is ANSI C.
strupr---force string to uppercase

Synopsis

#include <string.h>
char *strupr(char *a);

Description

strupr converts each character in the string at a to upper case.

Returns

strupr returns its argument, a.

Portability

strupr is not widely portable.
strxfrm---transform string

Synopsis

#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);

Description

This function transforms the string pointed to by s2 and places the resulting string into the array pointed to by s1. The transformation is such that if the strcmp function is applied to the two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of a strcoll function applied to the same two original strings.

No more than n characters are placed into the resulting array pointed to by s1, including the terminating null character. If n is zero, s1 may be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

With a C locale, this function just copies.

Returns

The strxfrm function returns the length of the transformed string (not including the terminating null character). If the value returned is n or more, the contents of the array pointed to by s1 are indeterminate.

Portability

strxfrm is ANSI C.
Signal Handling

A **signal** is an event that interrupts the normal flow of control in your program. Your operating environment normally defines the full set of signals available (see `sys/signal.h`), as well as the default means of dealing with them—typically, either printing an error message and aborting your program, or ignoring the signal.

All systems support at least the following signals:

**SIGABRT**

Abnormal termination of a program; raised by the `<<abort>>` function.

**SIGFPE**

A domain error in arithmetic, such as overflow, or division by zero.

**SIGILL**

Attempt to execute as a function data that is not executable.

**SIGINT**

Interrupt; an interactive attention signal.

**SIGSEGV**

An attempt to access a memory location that is not available.

**SIGTERM**

A request that your program end execution.

Two functions are available for dealing with asynchronous signals—one to allow your program to send signals to itself (this is called **raising** a signal), and one to specify subroutines (called **handlers** to handle particular signals that you anticipate may occur—whether raised by your own program or the operating environment.

To support these functions, `signal.h` defines three macros:
SIG_DFL

Used with the `signal` function in place of a pointer to a handler subroutine, to select the operating environment’s default handling of a signal.

SIG_IGN

Used with the `signal` function in place of a pointer to a handler, to ignore a particular signal.

SIG_ERR

Returned by the `signal` function in place of a pointer to a handler, to indicate that your request to set up a handler could not be honored for some reason.

`signal.h` also defines an integral type, `sig_atomic_t`. This type is not used in any function declarations; it exists only to allow your signal handlers to declare a static storage location where they may store a signal value. (Static storage is not otherwise reliable from signal handlers.)

- `raise`: Send a signal
- `signal`: Specify handler subroutine for a signal
raise---send a signal

Synopsis

```
#include <signal.h>
int raise(int sig);

int _raise_r(void *reent, int sig);
```

Description

Send the signal `sig` (one of the macros from `sys/signal.h`). This interrupts your program’s normal flow of execution, and allows a signal handler (if you’ve defined one, using `signal`) to take control.

The alternate function `_raise_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

The result is 0 if `sig` was successfully raised, 1 otherwise. However, the return value (since it depends on the normal flow of execution) may not be visible, unless the signal handler for `sig` terminates with a `return` or unless `SIG_IGN` is in effect for this signal.

Portability

ANSI C requires `raise`, but allows the full set of signal numbers to vary from one implementation to another.

Required OS subroutines: `getpid`, `kill`. 
signal---specify handler subroutine for a signal

Synopsis

```c
#include <signal.h>
void ( * signal(int sig, void(* func)(int)) )(int);

void ( * _signal_r(void * reent,  
               int sig, void(* func)(int)) )(int);

int raise (int sig);

int _raise_r (void * reent, int sig);
```

Description

`signal`, `raise` provide a simple signal/raise implementation for embedded targets.

`signal` allows you to request changed treatment for a particular signal `sig`. You can use one of the predefined macros `SIG_DFL` (select system default handling) or `SIG_IGN` (ignore this signal) as the value of `func`; otherwise, `func` is a function pointer that identifies a subroutine in your program as the handler for this signal.

Some of the execution environment for signal handlers is unpredictable; notably, the only library function required to work correctly from within a signal handler is `signal` itself, and only when used to redefine the handler for the current signal value.

Static storage is likewise unreliable for signal handlers, with one exception: if you declare a static storage location as `volatile sig_atomic_t`, then you may use that location in a signal handler to store signal values.

If your signal handler terminates using `return` (or implicit return), your program’s execution continues at the point where it was when the signal was raised (whether by your program itself, or by an external event). Signal handlers can also use functions such as `exit` and `abort` to avoid returning.

`raise` sends the signal `sig` to the executing program. It returns zero if successful, non-zero if unsuccessful.
The alternate functions _signal_r, _raise_r are the reentrant versions. The extra argument reent is a pointer to a reentrancy structure.

**Returns**
If your request for a signal handler cannot be honored, the result is SIG_ERR; a specific error number is also recorded in errno.

Otherwise, the result is the previous handler (a function pointer or one of the predefined macros).

**Portability**
ANSI C requires raise, signal.
Time Functions

This chapter groups functions used either for reporting on time (elapsed, current, or compute time) or to perform calculations based on time.

The header file `time.h' defines three types. clock_t and time_t are both used for representations of time particularly suitable for arithmetic. (In this implementation, quantities of type clock_t have the highest resolution possible on your machine, and quantities of type time_t resolve to seconds.) size_t is also defined if necessary for quantities representing sizes.

`time.h' also defines the structure tm for the traditional representation of Gregorian calendar time as a series of numbers, with the following fields:

- tm_sec
  Seconds.
- tm_min
  Minutes.
- tm_hour
  Hours.
- tm_mday
  Day.
- tm_mon
  Month.
- tm_year
  Year (since 1900).
- tm_wday
  Day of week: the number of days since Sunday.
- tm_yday
  Day of year.
Number of days elapsed since last January 1.

\texttt{tm\_isdst}

Daylight Savings Time flag: positive means DST in effect, zero means DST not in effect, negative means no information about DST is available.

- \texttt{asctime}: Format time as string
- \texttt{clock}: Cumulative processor time
- \texttt{ctime}: Convert time to local and format as string
- \texttt{difftime}: Subtract two times
- \texttt{gmtime}: Convert time to UTC (GMT) traditional representation
- \texttt{localtime}: Convert time to local representation
- \texttt{mktime}: Convert time to arithmetic representation
- \texttt{strftime}: Flexible calendar time formatter
- \texttt{time}: Get current calendar time (as single number)
C LIBRARY

asctime---format time as string

Synopsis

#include <time.h>
char *asctime(const struct tm *clock);
char *asctime_r(const struct tm *clock, char *buf);

Description
Format the time value at clock into a string of the form


The string is generated in a static buffer; each call to asctime overwrites the string generated by previous calls.

Returns
A pointer to the string containing a formatted timestamp.

Portability
ANSI C requires asctime.
clock---cumulative processor time

Synopsis

#include <time.h>
clock_t clock(void);

Description
Calculates the best available approximation of the cumulative amount of time used by your program since it started. To convert the result into seconds, divide by the macro \texttt{CLOCKS\_PER\_SEC}.

Returns
The amount of processor time used so far by your program, in units defined by the machine-dependent macro \texttt{CLOCKS\_PER\_SEC}. If no measurement is available, the result is \texttt{-1}.

Portability
ANSI C requires \texttt{clock} and \texttt{CLOCKS\_PER\_SEC}.
ctime---convert time to local and format as string

Synopsis

```c
#include <time.h>
char *ctime(time_t clock);
char *ctime_r(time_t clock, char *buf);
```

Description

Convert the time value at `clock` to local time (like `localtime`) and format it into a string of the form

```
```

(like `asctime`).

Returns

A pointer to the string containing a formatted timestamp.

Portability

ANSI C requires `ctime`. 
difftime---subtract two times

Synopsis

```c
#include <time.h>
double difftime(time_t tim1, time_t tim2);
```

Description
Subtracts the two times in the arguments: `tim1 - tim2'.

Returns
The difference (in seconds) between `tim2' and `tim1', as a double.

Portability
ANSI C requires `difftime', and defines its result to be in seconds in all implementations.
gmtime---convert time to UTC traditional form

Synopsis

#include <time.h>
struct tm *gmtime(const time_t *clock);
struct tm *gmtime_r(const time_t *clock, struct tm *res);

Description

gmtime assumes the time at clock represents a local time. gmtime converts it to UTC (Universal Coordinated Time, also known in some countries as GMT, Greenwich Mean time), then converts the representation from the arithmetic representation to the traditional representation defined by struct tm.

gmtime constructs the traditional time representation in static storage; each call to gmtime or localtime will overwrite the information generated by previous calls to either function.

Returns

A pointer to the traditional time representation (struct tm).

Portability

ANSI C requires gmtime.
localtime---convert time to local representation

Synopsis

#include <time.h>
struct tm *localtime(time_t *clock);
struct tm *localtime_r(time_t *clock, struct tm *res);

Description

localtime converts the time at clock into local time, then converts its representation from the arithmetic representation to the traditional representation defined by struct tm.

localtime constructs the traditional time representation in static storage; each call to gmtime or localtime will overwrite the information generated by previous calls to either function.

mktime is the inverse of localtime.

Returns

A pointer to the traditional time representation (struct tm).

Portability

ANSI C requires localtime.
C LIBRARY

mktime---convert time to arithmetic representation

Synopsis

#include <time.h>
time_t mktime(struct tm *timp);

Description

mktime assumes the time at timp is a local time, and converts its representation from the traditional representation defined by struct tm into a representation suitable for arithmetic.

localtime is the inverse of mktime.

Returns

If the contents of the structure at timp do not form a valid calendar time representation, the result is −1. Otherwise, the result is the time, converted to a time_t value.

Portability

ANSI C requires mktime.
strftime---flexible calendar time formatter

Synopsis

```c
#include <time.h>
size_t strftime(char *s, size_t maxsize, const char *format, const struct tm *timp);
```

Description

`strftime` converts a `struct tm` representation of the time (at `timp`) into a string, starting at `s` and occupying no more than `maxsize` characters.

You control the format of the output using the string at `format`. `format` can contain two kinds of specifications: text to be copied literally into the formatted string, and time conversion specifications. Time conversion specifications are two-character sequences beginning with `%` (use `%%` to include a percent sign in the output). Each defined conversion specification selects a field of calendar time data from `timp`, and converts it to a string in one of the following ways:

%a

An abbreviation for the day of the week.

%A

The full name for the day of the week.

%b

An abbreviation for the month name.

%B

The full name of the month.

%c

A string representing the complete date and time, in the form

```
Mon Apr 01 13:13:13 1992
```

%d
The day of the month, formatted with two digits.

%H

The hour (on a 24-hour clock), formatted with two digits.

%I

The hour (on a 12-hour clock), formatted with two digits.

%j

The count of days in the year, formatted with three digits (from ‘001’ to ‘366’).

%m

The month number, formatted with two digits.

%M

The minute, formatted with two digits.

%p

Either ‘AM’ or ‘PM’ as appropriate.

%S

The second, formatted with two digits.

%U

The week number, formatted with two digits (from ‘00’ to ‘53’; week number 1 is taken as beginning with the first Sunday in a year). See also %W.

%w

A single digit representing the day of the week: Sunday is day 0.

%W

Another version of the week number: like ‘%U’, but counting week 1 as beginning with the first Monday in a year.
A string representing the complete date, in a format like

```
Mon Apr 01 1992
```

%X

A string representing the full time of day (hours, minutes, and seconds), in a format like

```
13:13:13
```

%y

The last two digits of the year.

%Y

The full year, formatted with four digits to include the century.

%Z

Defined by ANSI C as eliciting the time zone if available; it is not available in this implementation (which accepts ‘%Z’ but generates no output for it).

%%

A single character, ‘%’.

Returns
When the formatted time takes up no more than maxsize characters, the result is the length of the formatted string. Otherwise, if the formatting operation was abandoned due to lack of room, the result is 0, and the string starting at s corresponds to just those parts of *format that could be completely filled in within the maxsize limit.

Portability
ANSI C requires strftime, but does not specify the contents of *s when the formatted string would require more than maxsize characters.
time---get current calendar time (as single number)

Synopsis

```c
#include <time.h>
time_t time(time_t *t);
```

Description

time looks up the best available representation of the current time and returns it, encoded as a `time_t`. It stores the same value at `t` unless the argument is NULL.

Returns

A -1 result means the current time is not available; otherwise the result represents the current time.

Portability

ANSI C requires time.
Locale

A **locale** is the name for a collection of parameters (affecting collating sequences and formatting conventions) that may be different depending on location or culture. The "C" locale is the only one defined in the ANSI C standard.

This is a minimal implementation, supporting only the required "C" value for locale; strings representing other locales are not honored. ("" is also accepted; it represents the default locale for an implementation, here equivalent to "C").

`locale.h` defines the structure `lconv` to collect the information on a locale, with the following fields:

- **char *decimal_point**
  The decimal point character used to format "ordinary" numbers (all numbers except those referring to amounts of money). "." in the C locale.

- **char *thousands_sep**
  The character (if any) used to separate groups of digits, when formatting ordinary numbers. "" in the C locale.

- **char *grouping**
  Specifications for how many digits to group (if any grouping is done at all) when formatting ordinary numbers. The *numeric value* of each character in the string represents the number of digits for the next group, and a value of 0 (that is, the string’s trailing NULL) means to continue grouping digits using the last value specified. Use `CHAR_MAX` to indicate that no further grouping is desired. "" in the C locale.

- **char *int_curr_symbol**
  The international currency symbol (first three characters), if any, and the character used to separate it from numbers. "" in the C locale.

- **char *currency_symbol**
  The local currency symbol, if any. "" in the C locale.
The symbol used to delimit fractions in amounts of money. "" in the C locale.

Similar to thousands_sep, but used for amounts of money. "" in the C locale.

Similar to grouping, but used for amounts of money. "" in the C locale.

A string to flag positive amounts of money when formatting. "" in the C locale.

A string to flag negative amounts of money when formatting. "" in the C locale.

The number of digits to display when formatting amounts of money to international conventions. CHAR_MAX (the largest number representable as a char) in the C locale.

The number of digits to display when formatting amounts of money to local conventions. CHAR_MAX in the C locale.

1 indicates the local currency symbol is used before a positive or zero formatted amount of money; 0 indicates the currency symbol is placed after the formatted number. CHAR_MAX in the C locale.

1 indicates the local currency symbol must be separated from positive or zero numbers by a space; 0 indicates that it is immediately adjacent to numbers. CHAR_MAX in the C locale.
char n_cs_precedes

1 indicates the local currency symbol is used before a negative formatted amount of money; 0 indicates the currency symbol is placed after the formatted number. CHAR_MAX in the C locale.

char n_sep_by_space

1 indicates the local currency symbol must be separated from negative numbers by a space; 0 indicates that it is immediately adjacent to numbers. CHAR_MAX in the C locale.

char p_sign_posn

Controls the position of the positive sign for numbers representing money. 0 means parentheses surround the number; 1 means the sign is placed before both the number and the currency symbol; 2 means the sign is placed after both the number and the currency symbol; 3 means the sign is placed just before the currency symbol; and 4 means the sign is placed just after the currency symbol. CHAR_MAX in the C locale.

char n_sign_posn

Controls the position of the negative sign for numbers representing money, using the same rules as p_sign_posn. CHAR_MAX in the C locale.

•  setlocale: Select or query locale
setlocale, localeconv---select or query locale

Synopsis

```
#include <locale.h>
char *setlocale(int category, const char *locale);
lconv *localeconv(void);

char *_setlocale_r(void *reent,  
int category, const char *locale);
lconv *__localeconv_r(void *reent);
```

Description

`setlocale` is the facility defined by ANSI C to condition the execution environment for international collating and formatting information; `localeconv` reports on the settings of the current locale.

This is a minimal implementation, supporting only the required "C" value for `locale`; strings representing other locales are not honored unless MB_CAPABLE is defined in which case three new extensions are allowed for LC_CTYPE only: "C-JIS", "C-EUCJP", and "C-SJIS". ("" is also accepted; it represents the default locale for an implementation, here equivalent to "C".)

If you use `NULL` as the `locale` argument, `setlocale` returns a pointer to the string representing the current locale (always "C" in this implementation). The acceptable values for `category` are defined in ‘locale.h’ as macros beginning with "LC_", but this implementation does not check the values you pass in the `category` argument.

`localeconv` returns a pointer to a structure (also defined in ‘locale.h’) describing the locale-specific conventions currently in effect.

`_localeconv_r` and `_setlocale_r` are reentrant versions of `localeconv` and `setlocale` respectively. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`setlocale` returns either a pointer to a string naming the locale currently in effect
localeconv returns a pointer to a structure of type lconv, which describes the formatting and collating conventions in effect (in this implementation, always those of the C locale).

**Portability**

ANSI C requires `setlocale`, but the only locale required across all implementations is the C locale.
Miscellaneous Macros and Functions

This chapter describes miscellaneous routines not covered elsewhere.

- `unctrl`: Return printable representation of a character
unctrl---translate characters to upper case

Synopsis

#include <unctrl.h>
char *unctrl(int c);
int unctrllen(int c);

Description

unctrl is a macro which returns the printable representation of \( c \) as a string.
unctrllen is a macro which returns the length of the printable representation of \( c \).

Returns

unctrl returns a string of the printable representation of \( c \).
unctrllen returns the length of the string which is the printable representation of \( c \).

Portability

unctrl and unctrllen are not ANSI C.
Chapter 4 Math Library

Mathematical Functions

This chapter groups a wide variety of mathematical functions. The corresponding definitions and declarations are in ‘math.h’. Two definitions from ‘math.h’ are of particular interest.

1. The representation of infinity as a double is defined as HUGE_VAL; this number is returned on overflow by many functions.

2. The structure exception is used when you write customized error handlers for the mathematical functions. You can customize error handling for most of these functions by defining your own version of matherr; see the section on matherr for details.

Since the error handling code calls fputs, the mathematical subroutines require stubs or minimal implementations for the same list of OS subroutines as fputs: close, fstat, isatty, lseek, read, sbrk, write. See section ‘System Calls’ in The Cygnus C Support Library, for a discussion and for sample minimal implementations of these support subroutines.

Alternative declarations of the mathematical functions, which exploit specific machine capabilities to operate faster--but generally have less error checking and may reflect additional limitations on some machines--are available when you include ‘fastmath.h’ instead of ‘math.h’.

There are four different versions of the math library routines: IEEE, POSIX, X/Open, or SVID. The version may be selected at runtime by setting the global variable _LIB_VERSION, defined in ‘math.h’. It may be set to one of the following constants defined in ‘math.h’: _IEEE_, _POSIX_, _XOPEN_, or _SVID_. The _LIB_VERSION variable is not specific to any thread, and changing it will affect all threads.

The versions of the library differ only in how errors are handled.

In IEEE mode, the matherr function is never called, no warning messages are
printed, and **errno** is never set.

In POSIX mode, **errno** is set correctly, but the **matherr** function is never called and no warning messages are printed.

In X/Open mode, **errno** is set correctly, and **matherr** is called, but warning message are not printed.

In SVID mode, functions which overflow return 3.40282346638528860e+38, the maximum single precision floating point value, rather than infinity. Also, **errno** is set correctly, **matherr** is called, and, if **matherr** returns 0, warning messages are printed for some errors. For example, by default `$\log(-1.0)$’ writes this message on standard error output:

```
log:  DOMAIN error
```

The library is set to X/Open mode by default.

- **acos**: Arccosine
- **acosh**: Inverse hyperbolic cosine
- **asin**: Arcsine
- **asinh**: Inverse hyperbolic sine
- **atan**: Arctangent
- **atan2**: Arctangent of y/x
- **atanh**: Inverse hyperbolic tangent
- **jN**: Bessel functions (jN, yN)
- **cbrt**: Cube root
- **copysign**: Sign of Y, magnitude of X
- **cosh**: Hyperbolic cosine
- **erf**: Error function (erf, erfc)
• **exp**: Exponential
• **expm1**: Exponential of x, -1
• **fabs**: Absolute value (magnitude)
• **floor**: Floor and ceiling (floor, ceil)
• **fmod**: Floating-point remainder (modulo)
• **frexp**: Split floating-point number
• **gamma**: Logarithmic gamma function
• **hypot**: Distance from origin
• **ilogb**: Get exponent
• **infinity**: Floating infinity
• **isnan**: Check type of number
• **ldexp**: Load exponent
• **log**: Natural logarithms
• **log10**: Base 10 logarithms
• **log1p**: Log of 1 + X
• **matherr**: Modifiable math error handler
• **modf**: Split fractional and integer parts
• **nan**: Floating Not a Number
• **nextafter**: Get next representable number
• **pow**: X to the power Y
• **rint**: Round and remainder (rint, remainder)
MATH LIBRARY

- **scalbn**: Scale number
- **sin**: Sine or cosine (sin, cos)
- **sinh**: Hyperbolic sine
- **sqrt**: Positive square root
- **tan**: Tangent
- **tanh**: Hyperbolic tangent
acos, acosf---arc cosine

Synopsis

#include <math.h>
double acos(double x);
float acosf(float x);

Description

acos computes the inverse cosine (arc cosine) of the input value. Arguments to acos must be in the range -1 to 1.

acosf is identical to acos, except that it performs its calculations on floats.

Returns

If x is not between -1 and 1, the returned value is NaN (not a number) the global variable errno is set to EDOM, and a DOMAIN error message is sent as standard error output.

You can modify error handling for these functions using matherr.
acosh, acoshf---inverse hyperbolic cosine

Synopsis

#include <math.h>
double acosh(double x);
float acoshf(float x);

Description

acosh calculates the inverse hyperbolic cosine of x. acosh is defined as
x must be a number greater than or equal to 1.

acoshf is identical, other than taking and returning floats.

Returns

acosh and acoshf return the calculated value. If x less than 1, the return value is
NaN and errno is set to EDOM.

You can change the error-handling behavior with the non-ANSI matherr
function.

Portability

Neither acosh nor acoshf are ANSI C. They are not recommended for portable
programs.
asinh, asinhf---arc sine

Synopsis

```c
#include <math.h>
double asin(double x);
float asinf(float x);
```

Description

`asin` computes the inverse sine (arc sine) of the argument `x`. Arguments to `asin` must be in the range -1 to 1.

`asinf` is identical to `asin`, other than taking and returning floats.

You can modify error handling for these routines using `matherr`.

Returns

If `x` is not in the range -1 to 1, `asin` and `asinf` return NaN (not a number), set the global variable `errno` to `EDOM`, and issue a `DOMAIN` error message.

You can change this error treatment using `matherr`. 
asinh, asinhf---inverse hyperbolic sine

Synopsis

#include <math.h>
double asinh(double x);
float asinhf(float x);

Description

asinh calculates the inverse hyperbolic sine of x. asinh is defined as

asinhf is identical, other than taking and returning floats.

Returns

asinh and asinhf return the calculated value.

Portability

Neither asinh nor asinhf are ANSI C.
atan, atanf---arc tangent

Synopsis

#include <math.h>
double atan(double x);
float atanf(float x);

Description

atan computes the inverse tangent (arc tangent) of the input value.

atanf is identical to atan, save that it operates on floats.

Returns

Portability

atan is ANSI C. atanf is an extension.
atan2, atan2f---arc tangent of y/x

Synopsis

#include <math.h>
double atan2(double y, double x);
float atan2f(float y, float x);

Description

atan2 computes the inverse tangent (arc tangent) of y/x. atan2 produces the correct result even for angles near (that is, when x is near 0).

atan2f is identical to atan2, save that it takes and returns float.

Returns

atan2 and atan2f return a value in radians, in the range of

If both x and y are 0.0, atan2 causes a DOMAIN error.

You can modify error handling for these functions using matherr.

Portability

atan2 is ANSI C. atan2f is an extension.
atanh, atanhf---inverse hyperbolic tangent

Synopsis

#include <math.h>
double atanh(double x);
float atanhf(float x);

Description

atanh calculates the inverse hyperbolic tangent of x.

atanhf is identical, other than taking and returning float values.

Returns

atanh and atanhf return the calculated value.

If is greater than 1, the global errno is set to EDOM and the result is a NaN. A DOMAIN error is reported.

If is 1, the global errno is set to EDOM; and the result is infinity with the same sign as x. A SING error is reported.

You can modify the error handling for these routines using matherr.

Portability

Neither atanh nor atanhf are ANSI C.
Synopsis

#include <math.h>

double j0(double x);
float j0f(float x);
double j1(double x);
float j1f(float x);
double jn(int n, double x);
float jnf(int n, float x);
double y0(double x);
float y0f(float x);
double y1(double x);
float y1f(float x);
double yn(int n, double x);
float ynf(int n, float x);

Description

The Bessel functions are a family of functions that solve the differential equation

These functions have many applications in engineering and physics.

jn calculates the Bessel function of the first kind of order n. j0 and j1 are special
cases for order 0 and order 1 respectively.

Similarly, yn calculates the Bessel function of the second kind of order n, and y0 and y1 are special cases for order 0 and 1.

jnf, j0f, j1f, ynf, y0f, and y1f perform the same calculations, but on float
rather than double values.

Returns

The value of each Bessel function at x is returned.

Portability

None of the Bessel functions are in ANSI C.
cbrt, cbrtf---cube root

Synopsis

#include <math.h>
double cbrrt(double x);
float cbrtf(float x);

Description
cbrt computes the cube root of the argument.

Returns
The cube root is returned.

Portability
cbrt is in System V release 4. cbrtf is an extension.
copysign, copysignf---sign of y, magnitude of x

Synopsis

```c
#include <math.h>
double copysign (double x, double y);
float copysignf (float x, float y);
```

Description

copysign constructs a number with the magnitude (absolute value) of its first argument, x, and the sign of its second argument, y.

copysignf does the same thing; the two functions differ only in the type of their arguments and result.

Returns

copysign returns a double with the magnitude of x and the sign of y. copysignf returns a float with the magnitude of x and the sign of y.

Portability

copysign is not required by either ANSI C or the System V Interface Definition (Issue 2).
cosh, coshf---hyperbolic cosine

Synopsis

```
#include <math.h>
double cosh(double x);
float coshf(float x)
```

Description

cosh computes the hyperbolic cosine of the argument \( x \). \( \cosh(x) \) is defined as

Angles are specified in radians. coshf is identical, save that it takes and returns float.

Returns

The computed value is returned. When the correct value would create an overflow, cosh returns the value HUGE_VAL with the appropriate sign, and the global value errno is set to ERANGE.

You can modify error handling for these functions using the function matherr.

Portability

cosh is ANSI. coshf is an extension.
erf, erff, erfc, erfcf---error function

Synopsis

#include <math.h>
double erf(double x);
float erff(float x);
double erfc(double x);
float erfcf(float x);

Description

erf calculates an approximation to the "error function", which estimates the probability that an observation will fall within $x$ standard deviations of the mean (assuming a normal distribution).

erfc calculates the complementary probability; that is, $erfc(x) = 1 - erf(x)$.
erfc is computed directly, so that you can use it to avoid the loss of precision that would result from subtracting large probabilities (on large $x$) from 1.

erff and erfcf differ from erf and erfc only in the argument and result types.

Returns

For positive arguments, erf and all its variants return a probability--a number between 0 and 1.

Portability

None of the variants of erf are ANSI C.
exp, expf---exponential

Synopsis

```c
#include <math.h>
double exp(double x);
float expf(float x);
```

Description

`exp` and `expf` calculate the exponential of `x`, that is, is the base of the natural system of logarithms, approximately 2.71828).

You can use the (non-ANSI) function `matherr` to specify error handling for these functions.

Returns

On success, `exp` and `expf` return the calculated value. If the result underflows, the returned value is 0. If the result overflows, the returned value is `HUGE_VAL`. In either case, `errno` is set to `ERANGE`.

Portability

`exp` is ANSI C. `expf` is an extension.
expm1, expm1f---exponential minus 1

Synopsis

#include <math.h>
double expm1(double x);
float expmlf(float x);

Description

expm1 and expm1f calculate the exponential of \( x \) and subtract 1, that is, is the base of the natural system of logarithms, approximately 2.71828). The result is accurate even for small values of \( x \), where using \( \exp(x) - 1 \) would lose many significant digits.

Returns

e raised to the power \( x \), minus 1.

Portability

Neither \texttt{expm1} nor \texttt{expmlf} is required by ANSI C or by the System V Interface Definition (Issue 2).
fabs, fabsf---absolute value (magnitude)

Synopsis

#include <math.h>
double fabs(double x);
float fabsf(float x);

Description

tabs andfabsf calculate the absolute value (magnitude) of the argument x, by
direct manipulation of the bit representation of x.

Returns

The calculated value is returned. No errors are detected.

Portability

tabs is ANSI. fabsf is an extension.
floor, floorf, ceil, ceilf---floor and ceiling

Synopsis

#include <math.h>
double floor(double x);
float floorf(float x);
double ceil(double x);
float ceilf(float x);

Description
floor and floorf find the nearest integer less than or equal to x. ceil and ceilf find the nearest integer greater than or equal to x.

Returns
floor and ceil return the integer result as a double. floorf and ceilf return the integer result as a float.

Portability
floor and ceil are ANSI. floorf and ceilf are extensions.
fmod, fmodf---floating-point remainder (modulo)

Synopsis

#include <math.h>
double fmod(double x, double y)
float fmodf(float x, float y)

Description
The fmod and fmodf functions compute the floating-point remainder of x/y (x modulo y).

Returns
The fmod function returns the value for the largest integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

fmod(x, 0) returns NaN, and sets errno to EDOM.

You can modify error treatment for these functions using matherr.

Portability
fmod is ANSI C. fmodf is an extension.
frexp, frexpf---split floating-point number

Synopsis

```c
#include <math.h>
double frexp(double val, int *exp);
float frexpf(float val, int *exp);
```

Description

All non zero, normal numbers can be described as \( m \times 2^p \). \textit{frexp} represents the double \( val \) as a mantissa \( m \) and a power of two \( p \). The resulting mantissa will always be greater than or equal to 0.5, and less than 1.0 (as long as \( val \) is nonzero). The power of two will be stored in \( *exp \).

\textit{frexpf} is identical, other than taking and returning floats rather than doubles.

Returns

\textit{frexp} returns the mantissa \( m \). If \( val \) is 0, infinity, or Nan, \textit{frexp} will set \( *exp \) to 0 and return \( val \).

Portability

\textit{frexp} is ANSI. \textit{frexpf} is an extension.
gamma, gammaf, lgamma, lgammaf, gamma_r,

Synopsis

#include <math.h>
double gamma(double x);
float gammaf(float x);
double lgamma(double x);
float lgammaf(float x);
double gamma_r(double x, int *signgamp);
float gammaf_r(float x, int *signgamp);
double lgamma_r(double x, int *signgamp);
float lgammaf_r(float x, int *signgamp);

Description
gamma calculates the natural logarithm of the gamma function of x. The gamma function \((\exp(\gamma(x)))\) is a generalization of factorial, and retains the property that Accordingly, the results of the gamma function itself grow very quickly. gamma is defined as to extend the useful range of results representable.

The sign of the result is returned in the global variable signgam, which is declared in math.h.

gammaf performs the same calculation as gamma, but uses and returns float values.

lgamma and lgammaf are alternate names for gamma and gammaf. The use of lgamma instead of gamma is a reminder that these functions compute the log of the gamma function, rather than the gamma function itself.

The functions gamma_r, gammaf_r, lgamma_r, and lgammaf_r are just like gamma, gammaf, lgamma, and lgammaf, respectively, but take an additional argument. This additional argument is a pointer to an integer. This additional argument is used to return the sign of the result, and the global variable signgam is not used. These functions may be used for reentrant calls (but they will still set the global variable errno if an error occurs).

Returns
Normally, the computed result is returned.

When $x$ is a nonpositive integer, \texttt{gamma} returns \texttt{HUGE\_VAL} and \texttt{errno} is set to \texttt{EDOM}. If the result overflows, \texttt{gamma} returns \texttt{HUGE\_VAL} and \texttt{errno} is set to \texttt{ERANGE}.

You can modify this error treatment using \texttt{matherr}.

\textbf{Portability}
Neither \texttt{gamma} nor \texttt{gammaf} is ANSI C.
hypot, hypotf---distance from origin

Synopsis

```c
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
```

Description

`hypot` calculates the Euclidean distance between the origin (0,0) and a point represented by the Cartesian coordinates (x,y). `hypotf` differs only in the type of its arguments and result.

Returns

Normally, the distance value is returned. On overflow, `hypot` returns `HUGE_VAL` and sets `errno` to `ERANGE`.

You can change the error treatment with `matherr`.

Portability

`hypot` and `hypotf` are not ANSI C.
ilogb, ilogbf---get exponent of floating point number

Synopsis

#include <math.h>
int ilogb(double val);
int ilogbf(float val);

Description

All non zero, normal numbers can be described as \( m \times 2^p \). \texttt{ilogb} and \texttt{ilogbf} examine the argument \texttt{val}, and return \( p \). The functions \texttt{frexp} and \texttt{frexpf} are similar to \texttt{ilogb} and \texttt{ilogbf}, but also return \( m \).

Returns

\texttt{ilogb} and \texttt{ilogbf} return the power of two used to form the floating point argument. If \texttt{val} is 0, they return – \texttt{INT_MAX} (\texttt{INT_MAX} is defined in \texttt{limits.h}). If \texttt{val} is infinite, or NaN, they return \texttt{INT_MAX}.

Portability

Neither \texttt{ilogb} nor \texttt{ilogbf} is required by ANSI C or by the System V Interface Definition (Issue 2).
infinity, infinityf---representation of infinity

Synopsis

#include <math.h>
double infinity(void);
float infinityf(void);

Description

infinity and infinityf return the special number IEEE infinity in double and single precision arithmetic respectively.
isnan, isnanf, isinf, isinff, finite, finitef---test for exceptional numbers

Synopsis

#include <ieeefp.h>
int isnan(double arg);
int isinf(double arg);
int finite(double arg);
int isnanf(float arg);
int isinff(float arg);
int finitef(float arg);

Description
These functions provide information on the floating point argument supplied.

There are five major number formats -

zero

- a number which contains all zero bits.

subnormal

- Is used to represent number with a zero exponent, but a non zero fraction.

normal

- A number with an exponent, and a fraction

infinity

- A number with an all 1’s exponent and a zero fraction.

NAN

- A number with an all 1’s exponent and a non zero fraction.

isnan returns 1 if the argument is a nan. isinf returns 1 if the argument is infinity. finite returns 1 if the argument is zero, subnormal or normal. The isnanf, isinff and finitef perform the same operations as their isnan, isinf and finite counterparts, but on single precision floating point numbers.
ldexp, ldexpf---load exponent

Synopsis

#include <math.h>
double ldexp(double val, int exp);
float ldexpf(float val, int exp);

Description

ldexp calculates the value ldexpf is identical, save that it takes and returns
float rather than double values.

Returns

ldexp returns the calculated value.

Underflow and overflow both set errno to ERANGE. On underflow, ldexp and
ldexpf return 0.0. On overflow, ldexp returns plus or minus HUGE_VAL.

Portability

ldexp is ANSI, ldexpf is an extension.
log, logf---natural logarithms

Synopsis

#include <math.h>
double log(double x);
float logf(float x);

Description
Return the natural logarithm of \( x \), that is, its logarithm base \( e \) (where \( e \) is the base of the natural system of logarithms, 2.71828...). \( \text{log} \) and \( \text{logf} \) are identical save for the return and argument types.

You can use the (non-ANSI) function \( \text{matherr} \) to specify error handling for these functions.

Returns
Normally, returns the calculated value. When \( x \) is zero, the returned value is \(-\text{HUGE}_\text{VAL}\) and \( \text{errno} \) is set to \text{ERANGE}. When \( x \) is negative, the returned value is \(-\text{HUGE}_\text{VAL}\) and \( \text{errno} \) is set to \text{EDOM}. You can control the error behavior via \( \text{matherr} \).

Portability
\( \text{log} \) is ANSI, \( \text{logf} \) is an extension.
log10, log10f---base 10 logarithms

Synopsis

#include <math.h>
double log10(double x);
float log10f(float x);

Description

log10 returns the base 10 logarithm of x. It is implemented as log(x) / log(10).

log10f is identical, save that it takes and returns float values.

Returns

log10 and log10f return the calculated value.

See the description of log for information on errors.

Portability

log10 is ANSI C. log10f is an extension.
log1p, log1pf---log of 1 + x

Synopsis

#include <math.h>
double log1p(double x);
float log1pf(float x);

Description

log1p calculates the natural logarithm of 1+x. You can use log1p rather than \`log(1+x)` for greater precision when x is very small.

log1pf calculates the same thing, but accepts and returns float values rather than double.

Returns

log1p returns a double, the natural log of 1+x. log1pf returns a float, the natural log of 1+x.

Portability

Neither log1p nor log1pf is required by ANSI C or by the System V Interface Definition (Issue 2).
matherr---modifiable math error handler

Synopsis

#include <math.h>
int matherr(struct exception *e);

Description

matherr is called whenever a math library function generates an error. You can replace matherr by your own subroutine to customize error treatment. The customized matherr must return 0 if it fails to resolve the error, and non-zero if the error is resolved.

When matherr returns a nonzero value, no error message is printed and the value of errno is not modified. You can accomplish either or both of these things in your own matherr using the information passed in the structure *e.

This is the exception structure (defined in `math.h'):

    struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
        int err;
    };

The members of the exception structure have the following meanings:

    type

        The type of mathematical error that occurred; macros encoding error types are also defined in `math.h'.

    name

        a pointer to a null-terminated string holding the name of the math library function where the error occurred.

    arg1, arg2

        The arguments which caused the error.
The error return value (what the calling function will return).

If set to be non-zero, this is the new value assigned to `errno`.

The error types defined in `math.h` represent possible mathematical errors as follows:

**DOMAIN**

An argument was not in the domain of the function; e.g. `log(-1.0)`.

**SING**

The requested calculation would result in a singularity; e.g. `pow(0.0, -2.0)`

**OVERFLOW**

A calculation would produce a result too large to represent; e.g. `exp(1000.0)`.

**UNDERFLOW**

A calculation would produce a result too small to represent; e.g. `exp(-1000.0)`.

**TLOSS**

Total loss of precision. The result would have no significant digits; e.g. `sin(10e70)`.

**PLOSS**

Partial loss of precision.

**Returns**

The library definition for `matherr` returns 0 in all cases.

You can change the calling function’s result from a customized `matherr` by modifying `e->retval`, which propagates back to the caller.
If \texttt{matherr} returns 0 (indicating that it was not able to resolve the error) the caller sets \texttt{errno} to an appropriate value, and prints an error message.

\textbf{Portability}

\texttt{matherr} is not ANSI C.
modf, modff---split fractional and integer parts

Synopsis

#include <math.h>
double modf(double val, double *ipart);
float modff(float val, float *ipart);

Description

modf splits the double val apart into an integer part and a fractional part, returning the fractional part and storing the integer part in *ipart. No rounding whatsoever is done; the sum of the integer and fractional parts is guaranteed to be exactly equal to val. That is, if . realpart = modf(val, &intpart); then ‘realpart+intpart’ is the same as val. modff is identical, save that it takes and returns float rather than double values.

Returns

The fractional part is returned. Each result has the same sign as the supplied argument val.

Portability

modf is ANSI C. modff is an extension.
nan, nanf---representation of infinity

Synopsis

```c
#include <math.h>
double nan(void);
float nanf(void);
```

Description

 nan and nanf return an IEEE NaN (Not a Number) in double and single precision arithmetic respectivly.
nextafter, nextafterf---get next number

Synopsis

#include <math.h>
double nextafter(double val, double dir);
float nextafterf(float val, float dir);

Description

nextafter returns the double) precision floating point number closest to val in
the direction toward dir. nextafterf performs the same operation in single
precision. For example, nextafter(0.0,1.0) returns the smallest positive
number which is representable in double precision.

Returns

Returns the next closest number to val in the direction toward dir.

Portability

Neither nextafter nor nextafterf is required by ANSI C or by the System V
Interface Definition (Issue 2).
pow, powf---x to the power y

Synopsis

```c
#include <math.h>
double pow(double x, double y);
float pow(float x, float y);
```

Description

`pow` and `powf` calculate $x$ raised to the power $y$.

Returns

On success, `pow` and `powf` return the value calculated.

When the argument values would produce overflow, `pow` returns `HUGE_VAL` and set `errno` to `ERANGE`. If the argument $x$ passed to `pow` or `powf` is a negative noninteger, and $y$ is also not an integer, then `errno` is set to `EDOM`. If $x$ and $y$ are both 0, then `pow` and `powf` return 1.

You can modify error handling for these functions using `matherr`.

Portability

`pow` is ANSI C. `powf` is an extension.
rint, rintf, remainder, remainderf---round and remainder

Synopsis

#include <math.h>
double rint(double x);
float rintf(float x);
double remainder(double x, double y);
float remainderf(float x, float y);

Description
rint and rintf returns their argument rounded to the nearest integer. remainder and remainderf find the remainder of x/y; this value is in the range -y/2 .. +y/2.

Returns
rint and remainder return the integer result as a double.

Portability
rint and remainder are System V release 4. rintf and remainderf are extensions.
scalbn, scalbnf---scale by integer

Synopsis

```c
#include <math.h>
double scalbn(double x, int y);
float scalbnf(float x, int y);
```

Description

`scalbn` and `scalbnf` scale `x` by `n`, returning `x` times 2 to the power `n`. The result is computed by manipulating the exponent, rather than by actually performing an exponentiation or multiplication.

Returns

`x` times 2 to the power `n`.

Portability

Neither `scalbn` nor `scalbnf` is required by ANSI C or by the System V Interface Definition (Issue 2).
sqrt, sqrtf---positive square root

Synopsis

#include <math.h>
double sqrt(double x);
float sqrtf(float x);

Description

sqrt computes the positive square root of the argument. You can modify error handling for this function with matherr.

Returns

On success, the square root is returned. If x is real and positive, then the result is positive. If x is real and negative, the global value errno is set to EDOM (domain error).

Portability

sqrt is ANSI C. sqrtf is an extension.
sin, sinf, cos, cosf---sine or cosine

Synopsis

```c
#include <math.h>
double sin(double x);
float sinf(float x);
double cos(double x);
float cosf(float x);
```

Description

sin and cos compute (respectively) the sine and cosine of the argument x. Angles are specified in radians.

sinf and cosf are identical, save that they take and return float values.

Returns

The sine or cosine of x is returned.

Portability

sin and cos are ANSI C. sinf and cosf are extensions.
sinh, sinhf---hyperbolic sine

Synopsis

#include <math.h>
double sinh(double x);
float  sinhf(float x);

Description

\( \text{sinh} \) computes the hyperbolic sine of the argument \( x \). Angles are specified in radians. \( \text{sinh}(x) \) is defined as

\( \text{sinhf} \) is identical, save that it takes and returns \( \text{float} \) values.

Returns

The hyperbolic sine of \( x \) is returned.

When the correct result is too large to be representable (an overflow), \( \text{sinh} \) returns \( \text{HUGE}_\text{VAL} \) with the appropriate sign, and sets the global value \( \text{errno} \) to \( \text{ERANGE} \).

You can modify error handling for these functions with \text{matherr}.

Portability

\( \text{sinh} \) is ANSI C. \( \text{sinhf} \) is an extension.
tan, tanf---tangent

Synopsis

#include <math.h>
double tan(double x);
float tanf(float x);

Description

tan computes the tangent of the argument x. Angles are specified in radians.

tanf is identical, save that it takes and returns float values.

Returns

The tangent of x is returned.

Portability

tan is ANSI. tanf is an extension.
tanh, tanhf---hyperbolic tangent

Synopsis

```c
#include <math.h>
double tanh(double x);
float tanhf(float x);
```

Description

tanh computes the hyperbolic tangent of the argument x. Angles are specified in radians.

tanh(x) is defined as

\[ \frac{\sinh(x)}{\cosh(x)} \]

tanhf is identical, save that it takes and returns float values.

Returns

The hyperbolic tangent of x is returned.

Portability

tanh is ANSI C. tanhf is an extension.
Chapter 5 Compression Library

zlibVersion---get ZLIB version

Synopsis

#include <zlib.h>

ZEXTERN char * ZEXPORT zlibVersion();

Description

The application can compare zlibVersion and ZLIB_VERSION for consistency. If the first character differs, the library code actually used is not compatible with the zlib.h header file used by the application. This check is automatically made by deflateInit and inflateInit.
deflateInit---initialize stream for compression

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT deflateInit(z_streamp strm, int level)

Description

Initializes the internal stream state for compression. The fields zalloc, zfree and
opaque must be initialized before by the caller.

If zalloc and zfree are set to Z_NULL, deflateInit updates them to use default
allocation functions.

The compression level must be Z_DEFAULT_COMPRESSION, or between 0 and 9:

1 gives best speed, 9 gives best compression, 0 gives no compression at all
(the input data is simply copied a block at a time).

Z_DEFAULT_COMPRESSION requests a default compromise between
speed and compression (currently equivalent to level 6).

deflateInit returns Z_OK if success, Z_MEM_ERROR if there was not
enough memory, Z_STREAM_ERROR if level is not a valid compression
level,
Z_VERSION_ERROR if the zlib library version (zlib_version) is
incompatible with the version assumed by the caller (ZLIB_VERSION).
msg is set to null if there is no error message. deflateInit does not perform
any compression: this will be done by deflate().
deflate---compress stream

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT deflate (z_streamp strm, int flush);

Description

Deflate compresses as much data as possible, and stops when the input buffer becomes empty or the output buffer becomes full. It may introduce some output latency (reading input without producing any output) except when forced to flush.

The detailed semantics are as follows. deflate performs one or both of the following actions:

- Compress more input starting at next_in and update next_in and avail_in accordingly. If not all input can be processed (because there is not enough room in the output buffer), next_in and avail_in are updated and processing will resume at this point for the next call of deflate().

- Provide more output starting at next_out and update next_out and avail_out accordingly. This action is forced if the parameter flush is non zero.

Forcing flush frequently degrades the compression ratio, so this parameter should be set only when necessary (in interactive applications). Some output may be provided even if flush is not set.

Before the call of deflate(), the application should ensure that at least one of the actions is possible, by providing more input and/or consuming more output, and updating avail_in or avail_out accordingly; avail_out should never be zero before the call. The application can consume the compressed output when it wants, for example when the output buffer is full (avail_out == 0), or after each call of deflate(). If deflate returns Z_O and with zero avail_out, it must be called again after making room in the output buffer because there might be more output pending.

If the parameter flush is set to Z_SYNC_FLUSH, all pending output is flushed to the output buffer and the output is aligned on a byte boundary, so that the
COMPRESSION LIBRARY

decompressor can get all input data available so far. (In particular avail_in is zero after the call if enough output space has been provided before the call.) Flushing may degrade compression for some compression algorithms and so it should be used only when necessary.

If flush is set to Z_FULL_FLUSH, all output is flushed as with Z_SYNC_FLUSH, and the compression state is reset so that decompression can restart from this point if previous compressed data has been damaged or if random access is desired. Using Z_FULL_FLUSH too often can seriously degrade the compression.

If deflate returns with avail_out == 0, this function must be called again with the same value of the flush parameter and more output space (updated avail_out), until the flush is complete (deflate returns with non-zero avail_out).

If the parameter flush is set to Z_FINISH, pending input is processed, pending output is flushed and deflate returns with Z_STREAM_END if there was enough output space; if deflate returns with Z_OK, this function must be called again with Z_FINISH and more output space (updated avail_out) but no more input data, until it returns with Z_STREAM_END or an error. After deflate has returned Z_STREAM_END, the only possible operations on the stream are deflateReset or deflateEnd.

Z_FINISH can be used immediately after deflateInit if all the compression is to be done in a single step. In this case, avail_out must be at least 0.1% larger than avail_in plus 12 bytes. If deflate does not return Z_STREAM_END, then it must be called again as described above.

deflate() sets strm->adler to the adler32 checksum of all input read so far (that is, total_in bytes).

deflate() may update data_type if it can make a good guess about the input data type (Z_ASCII or Z_BINARY). In doubt, the data is considered binary. This field is only for information purposes and does not affect the compression algorithm in any manner.

deflate() returns Z_OK if some progress has been made (more input processed or more output produced), Z_STREAM_END if all input has been consumed and all
output has been produced (only when flush is set to Z_FINISH),
Z_STREAM_ERROR if the stream state was inconsistent (for example if next_in
or next_out was NULL), Z_BUF_ERROR if no progress is possible (for example
avail_in or avail_out was zero).
deflateEnd---end compression on stream

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT deflateEnd (z_streamp strm);

Description

All dynamically allocated data structures for this stream are freed.

This function discards any unprocessed input and does not flush any pending output.

deflateEnd returns Z_OK if success, Z_STREAM_ERROR if the stream state was inconsistent, Z_DATA_ERROR if the stream was freed prematurely (some input or output was discarded). In the error case, msg may be set but then points to a static string (which must not be deallocated).
inflateInit---initialize decompression on stream

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT inflateInit (z_streamp strm);

Description

Initializes the internal stream state for decompression. The fields next_in, avail_in, zalloc, zfree and opaque must be initialized before by the caller. If next_in is not Z_NULL and avail_in is large enough (the exact value depends on the compression method), inflateInit determines the compression method from the zlib header and allocates all data structures accordingly; otherwise the allocation will be deferred to the first call of inflate. If zalloc and zfree are set to Z_NULL, inflateInit updates them to use default allocation functions.

inflateInit returns Z_OK if success, Z_MEM_ERROR if there was not enough memory, Z_VERSION_ERROR if the zlib library version is incompatible with the version assumed by the caller. msg is set to null if there is no error message. inflateInit does not perform any decompression apart from reading the zlib header if present: this will be done by inflate(). (So next_in an avail_in may be modified, but next_out and avail_out are unchanged.)
inflatedecompress stream

Synopsis

```
#include <zlib.h>
ZEXTERN int ZEXPORT inflate (z_streamp strm, int flush);
```

Description

inflate decompresses as much data as possible, and stops when the input buffer becomes empty or the output buffer becomes full. It may some introduce some output latency (reading input without producing any output) except when forced to flush.

The detailed semantics are as follows. inflate performs one or both of the following actions:

- Decompress more input starting at next_in and update next_in and avail_in accordingly. If not all input can be processed (because there is not enough room in the output buffer), next_in is updated and processing will resume at this point for the next call of inflate().

- Provide more output starting at next_out and update next_out and avail_out accordingly. inflate() provides as much output as possible, until there is no more input data or no more space in the output buffer (see below about the flush parameter).

Before the call of inflate(), the application should ensure that at least one of the actions is possible, by providing more input and/or consuming more output, and updating the next_* and avail_* values accordingly.

The application can consume the uncompressed output when it wants, for example when the output buffer is full (avail_out == 0), or after each call of inflate(). If inflate returns Z_OK and with zero avail_out, it must be called again after making room in the output buffer because there might be more output pending.
If the parameter flush is set to Z_SYNC_FLUSH, inflate flushes as much output as possible to the output buffer. The flushing behavior of inflate is not specified for values of the flush parameter other than Z_SYNC_FLUSH and Z_FINISH, but the current implementation actually flushes as much output as possible anyway.

inflate() should normally be called until it returns Z_STREAM_END or an error. However if all decompression is to be performed in a single step (a single call of inflate), the parameter flush should be set to Z_FINISH. In this case all pending input is processed and all pending output is flushed; avail_out must be large enough to hold all the uncompressed data. (The size of the uncompressed data may have been saved by the compressor for this purpose.) The next operation on this stream must be inflateEnd to deallocate the decompression state. The use of Z_FINISH is never required, but can be used to inform inflate that a faster routine may be used for the single inflate() call.

If a preset dictionary is needed at this point (see inflateSetDictionary below), inflate sets strm-adler to the adler32 checksum of the dictionary chosen by the compressor and returns Z_NEED_DICT; otherwise it sets strm->adler to the adler32 checksum of all output produced so far (that is, total_out bytes) and returns Z_OK, Z_STREAM_END or an error code as described below. At the end of the stream, inflate() checks that its computed adler32 checksum is equal to that saved by the compressor and returns Z_STREAM_END only if the checksum is correct.

inflate() returns Z_OK if some progress has been made (more input processed or more output produced), Z_STREAM_END if the end of the compressed data has been reached and all uncompressed output has been produced, Z_NEED_DICT if a preset dictionary is needed at this point, Z_DATA_ERROR if the input data was corrupted (input stream not conforming to the zlib format or incorrect adler32 checksum), Z_STREAM_ERROR if the stream structure was inconsistent (for example if next_in or next_out was NULL), Z_MEM_ERROR if there was not enough memory, Z_BUF_ERROR if no progress is possible or if there was not enough room in the output buffer when Z_FINISH is used. In the Z_DATA_ERROR case, the application may then call inflateSync to look for a good compression block.
Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT inflateEnd (z_streamp strm);

Description

All dynamically allocated data structures for this stream are freed. This function discards any unprocessed input and does not flush any pending output.

inflateEnd returns Z_OK if success, Z_STREAM_ERROR if the stream state was inconsistent. In the error case, msg may be set but then points to a static string (which must not be deallocated).
deflateInit2---detailed initialization of compression on stream

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT deflateInit2 (z_streamp strm, int level, int method, int windowBits, int memLevel, int strategy);

Description

The following functions are needed only in some special applications.

This is another version of deflateInit with more compression options. The fields next_in, zalloc, zfree and opaque must be initialized before by the caller.

The method parameter is the compression method. It must be Z_DEFLATED in this version of the library.

The windowBits parameter is the base two logarithm of the window size (the size of the history buffer). It should be in the range 8..15 for this version of the library. Larger values of this parameter result in better compression at the expense of memory usage. The default value is 15 if deflateInit is used instead.

The memLevel parameter specifies how much memory should be allocated for the internal compression state. memLevel=1 uses minimum memory but is slow and reduces compression ratio; memLevel=9 uses maximum memory for optimal speed. The default value is 8. See zconf.h for total memory usage as a function of windowBits and memLevel.

The strategy parameter is used to tune the compression algorithm. Use the value Z_DEFAULT_STRATEGY for normal data, Z_FILTERED for data produced by a filter (or predictor), or Z_HUFFMAN_ONLY to force Huffman encoding only (no string match). Filtered data consists mostly of small values with a somewhat random distribution. In this case, the compression algorithm is tuned to compress them better. The effect of Z_FILTERED is to force more
COMPRESS LIBRARY
Huffman coding and less string matching; it is somewhat intermediate between Z_DEFAULT and Z_HUFFMAN_ONLY. The strategy parameter only affects the compression ratio but not the correctness of the compressed output even if it is not set appropriately.

deflateInit2 returns Z_OK if success, Z_MEM_ERROR if there was not enough memory, Z_STREAM_ERROR if a parameter is invalid (such as an invalid method). msg is set to null if there is no error message. deflateInit2 does not perform any compression: this will be done by deflate().
deflateSetDictionary---initialize compression dictionary

Synopsis
#include <zlib.h>
ZEXTERN int ZEXPORT deflateSetDictionary (z_streamp strm,
const Bytef *dictionary uInt dictLength);

Description
Initializes the compression dictionary from the given byte sequencewithout
producing any compressed output. This function must be called immediately after
deflateInit, deflateInit2 or deflateReset, before any call of deflate. The compressor
and decompressor must use exactly the same dictionary (see
inflateSetDictionary).

The dictionary should consist of strings (byte sequences) that are likely to be
encountered later in the data to be compressed, with the most commonly used
strings preferably put towards the end of the dictionary. Using a dictionary is most
useful when the data to be compressed is short and can be predicted with good
accuracy; the data can then be compressed better than with the default empty
dictionary.

Depending on the size of the compression data structures selected by deflateInit or
deflateInit2, a part of the dictionary may in effect be discarded, for example if the
dictionary is larger than the window size in deflate or deflate2. Thus the strings
most likely to be useful should be put at the end of the dictionary, not at the front.

Upon return of this function, strm->adler is set to the Adler32 value of the
dictionary; the decompressor may later use this value to determine which
dictionary has been used by the compressor. (The Adler32 value applies to the
whole dictionary even if only a subset of the dictionary is actually used by the
compressor.)

deflateSetDictionary returns Z_OK if success, or Z_STREAM_ERROR if a
parameter is invalid (such as NULL dictionary) or the stream state is inconsistent
(for example if deflate has already been called for this stream or if the
compression method is bsort). deflateSetDictionary does not perform any compression: this will be done by deflate().
deflateCopy---duplicate compression stream

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT deflateCopy (z_streamp dest, z_streamp source);

Description

Sets the destination stream as a complete copy of the source stream.

This function can be useful when several compression strategies will be tried, for example when there are several ways of pre-processing the input data with a filter. The streams that will be discarded should then be freed by calling deflateEnd. Note that deflateCopy duplicates the internal compression state which can be quite large, so this strategy is slow and can consume lots of memory.

deflateCopy returns Z_OK if success, Z_MEM_ERROR if there was not enough memory, Z_STREAM_ERROR if the source stream state was inconsistent (such as zalloc being NULL). msg is left unchanged in both source and destination.
**Synopsis**

```c
#include <zlib.h>
ZEXTERN int ZEXPORT deflateReset (z_streamp strm);
```

**Description**

This function is equivalent to deflateEnd followed by deflateInit, but does not free and reallocate all the internal compression state. The stream will keep the same compression level and any other attributes that may have been set by deflateInit2.

deflateReset returns Z_OK if success, or Z_STREAM_ERROR if the source stream state was inconsistent (such as zalloc or state being NULL).
deflateParams---set compression parameters

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT deflateParams (z_streamp strm, int level, int strategy);

Description

Dynamically update the compression level and compression strategy. The interpretation of level and strategy is as in deflateInit2. This can be used to switch between compression and straight copy of the input data, or to switch to a different kind of input data requiring a different strategy. If the compression level is changed, the input available so far is compressed with the old level (and may be flushed); the new level will take effect only at the next call of deflate().

Before the call of deflateParams, the stream state must be set as for a call of deflate(), since the currently available input may have to be compressed and flushed. In particular, strm->avail_out must be non-zero.

deflateParams returns Z_OK if success, Z_STREAM_ERROR if the source stream state was inconsistent or if a parameter was invalid, Z_BUF_ERROR if strm->avail_out was zero.
inflatedInit2---detailed initialization of decompression stream

Synopsis

```
#include <zlib.h>
ZEXTERN int ZEXPORT inflateInit2 (z_streamp strm, int windowBits);
```

Description

This is another version of inflateInit with an extra parameter. The fields next_in, avail_in, zalloc, zfree and opaque must be initialized before by the caller.

The windowBits parameter is the base two logarithm of the maximum window size (the size of the history buffer). It should be in the range 8..15 for this version of the library. The default value is 15 if inflateInit is used instead. If a compressed stream with a larger window size is given as input, inflate() will return with the error code Z_DATA_ERROR instead of trying to allocate a larger window.

inflateInit2 returns Z_OK if success, Z_MEM_ERROR if there was not enough memory, Z_STREAM_ERROR if a parameter is invalid (such as a negative memLevel). msg is set to null if there is no error message. inflateInit2 does not perform any decompression apart from reading the zlib header if present: this will be done by inflate(). (So next_in and avail_in may be modified, but next_out and avail_out are unchanged.)
deflateSetDictionary---set decompression dictionary

Synopsis

```
#include <zlib.h>
ZEXTERN int ZEXPORT inflateSetDictionary (z_streamp strm, const Bytef *dictionary, uInt dictLength);
```

Description

Initializes the decompression dictionary from the given uncompressed byte sequence. This function must be called immediately after a call of inflate if this call returned Z_NEED_DICT. The dictionary chosen by the compressor can be determined from the Adler32 value returned by this call of inflate. The compressor and decompressor must use exactly the same dictionary (see deflateSetDictionary).

inflateSetDictionary returns Z_OK if success, Z_STREAM_ERROR if a parameter is invalid (such as NULL dictionary) or the stream state is inconsistent, Z_DATA_ERROR if the given dictionary doesn't match the expected one (incorrect Adler32 value). inflateSetDictionary does not perform any decompression: this will be done by subsequent calls of inflate().
inflatesync---skip invalid data

Synopsis

```
#include <zlib.h>
ZEXTERN int ZEXPORT inflatesync (z_streamp strm);
```

Description

Skips invalid compressed data until a full flush point (see above the description of deflate with Z_FULL_FLUSH) can be found, or until all available input is skipped. No output is provided.

inflatesync returns Z_OK if a full flush point has been found, Z_BUF_ERROR if no more input was provided, Z_DATA_ERROR if no flush point has been found, or Z_STREAM_ERROR if the stream structure was inconsistent. In the success case, the application may save the current current value of total_in which indicates where valid compressed data was found. In the error case, the application may repeatedly call inflatesync, providing more input each time, until success or end of the input data.
inflateReset---end decompression and reset

Synopsis

```
#include <zlib.h>
ZEXTERN int ZEXPORT inflateReset (z_streamp strm);
```

Description

This function is equivalent to inflateEnd followed by inflateInit, but does not free and reallocate all the internal decompression state. The stream will keep attributes that may have been set by inflateInit2.

inflateReset returns Z_OK if success, or Z_STREAM_ERROR if the source stream state was inconsistent (such as zalloc or state being NULL).
Compress---compress memory buffer

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT compress (Bytef *dest, uLongf *destLen, const Bytef *source, uLong sourceLen);

Description

Compresses the source buffer into the destination buffer. sourceLen is the byte length of the source buffer. Upon entry, destLen is the total size of the destination buffer, which must be at least 0.1% larger than sourceLen plus 12 bytes. Upon exit, destLen is the actual size of the compressed buffer.

This function can be used to compress a whole file at once if the input file is mmap’ed.

compress returns Z_OK if success, Z_MEM_ERROR if there was not enough memory, Z_BUF_ERROR if there was not enough room in the output buffer.
compress2---detailed compress memory buffer

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT compress2 (Bytef *dest, uLongf *destLen,
const Bytef *source, uLong sourceLen, int level);

Description

Compresses the source buffer into the destination buffer. The level parameter has
the same meaning as in deflateInit. sourceLen is the byte length of the source
buffer. Upon entry, destLen is the total size of the destination buffer, which must
be at least 0.1% larger than sourceLen plus 12 bytes. Upon exit, destLen is the
actual size of the compressed buffer.

compress2 returns Z_OK if success, Z_MEM_ERROR if there was not enough
memory, Z_BUF_ERROR if there was not enough room in the output buffer,
Z_STREAM_ERROR if the level parameter is invalid.
umcompress---uncompress Buffer

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT uncompress (Bytef *dest, uLongf *destLen, const Bytef *source, uLong sourceLen);

Description

Decompresses the source buffer into the destination buffer. sourceLen is the byte length of the source buffer. Upon entry, destLen is the total size of the destination buffer, which must be large enough to hold the entire uncompressed data. (The size of the uncompressed data must have been saved previously by the compressor and transmitted to the decompressor by some mechanism outside the scope of this compression library.)

Upon exit, destLen is the actual size of the compressed buffer.

This function can be used to decompress a whole file at once if the input file is mmap’ed.

uncompress returns Z_OK if success, Z_MEM_ERROR if there was not enough memory, Z_BUF_ERROR if there was not enough room in the output buffer, or Z_DATA_ERROR if the input data was corrupted.
gzopen---open gzip file

Synopsis

#include <zlib.h>

typedef voidp gzFile;

ZEXTERN gzFile ZEXPORT gzopen (const char *path, const char *mode);

Description

Opens a gzip (.gz) file for reading or writing. The mode parameter is as in fopen ("rb" or "wb") but can also include a compression level ("wb9") or a strategy: 'f' for filtered data as in "wb6f", 'h' for Huffman only compression as in "wb1h". (See the description of deflateInit2 for more information about the strategy parameter.)

gzopen can be used to read a file which is not in gzip format; in this case gzread will directly read from the file without decompression.

gzopen returns NULL if the file could not be opened or if there was insufficient memory to allocate the (de)compression state; errno can be checked to distinguish the two cases (if errno is zero, the zlib error is Z_MEM_ERROR). */
gzdopen---open gzip file from file descriptor

Synopsis

#include <zlib.h>
ZEXTERN gzFile ZEXPORT gzdopen (int fd, const char *mode);

Description

gzdopen() associates a gzFile with the file descriptor fd. File
descriptors are obtained from calls like open, dup, creat, pipe or fileno (in the file
has been previously opened with fopen). The mode parameter is as in gzopen.

The next call of gzclose on the returned gzFile will also close the file descriptor
fd, just like fclose(fdopen(fd), mode) closes the file descriptor fd. If you want to
keep fd open, use gzdopen(dup(fd), mode).

gzdopen returns NULL if there was insufficient memory to allocate
the (de)compression state.
gzsetparams---set parameters for gzip file

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT gzsetparams (gzFile file, int level, int strategy)

Description

Dynamically update the compression level or strategy. See the description of deflateInit2 for the meaning of these parameters.

gzsetparams returns Z_OK if success, or Z_STREAM_ERROR if the file was not opened for writing.
gzread---read from gzip file

Synopsis

```c
#include <zlib.h>
ZEXTERN int ZEXPORT  gzread (gzFile file, voidp buf, unsigned len);
```

Description

Reads the given number of uncompressed bytes from the compressed file. If the input file was not in gzip format, gzread copies the given number of bytes into the buffer.

gzread returns the number of uncompressed bytes actually read (0 for end of file, -1 for error).
gzwrite---write to gzip file

Synopsis

```c
#include <zlib.h>
ZEXTERN int ZEXPORT gzwrite (gzFile file, const voidp buf, unsigned len);
```

Description
Writes the given number of uncompressed bytes into the compressed file.

gzwrite returns the number of uncompressed bytes actually written (0 in case of error).
gzprintf---printf to gzip file

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORTVA gzprintf (gzFile file, const char *format, ...

Description
Converts, formats, and writes the args to the compressed file under control of the format string, as in fprintf.

gzprintf returns the number of uncompressed bytes actually written (0 in case of error).
gzputs---put string to gzip file

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT gzputs (gzFile file, const char *s);

Description

Writes the given null-terminated string to the compressed file, excluding the terminating null character.

gzputs returns the number of characters written, or -1 in case of error.
gzgets---get line from gzip file

Synopsis

#include <zlib.h>
ZEXTERN char * ZEXPORT gzgets (gzFile file, char *buf, int len);

Description

Reads bytes from the compressed file until len-1 characters are read, or a newline character is read and transferred to buf, or an end-of-file condition is encountered. The string is then terminated with a null character.

gzgets returns buf, or Z_NULL in case of error.
gzgetc---get character from gzip file

Synopsis

```c
#include <zlib.h>
ZEXTERN int ZEXPORT gzgetc (gzFile file);
```

Description

 Reads one byte from the compressed file. gzgetc returns this byte or -1 in case of end of file or error.
**gzflush---flush gzip file**

**Synopsis**
```c
#include <zlib.h>
ZEXTERN int ZEXPORT gzflush (gzFile file, int flush);
```

**Description**
Flushes all pending output into the compressed file. The parameter flush is as in the deflate() function. The return value is the zlib error number (see function gzerror below). gzflush returns Z_OK if the flush parameter is Z_FINISH and all output could be flushed.

gzflush should be called only when strictly necessary because it can degrade compression.
gzseek---seek on gzip file

Synopsis

#include <zlib.h>
ZEXTERN z_off_t ZEXPORT gzseek (gzFile file, z_off_t offset, int whence);

Description

Sets the starting position for the next gzread or gzwrite on the given compressed file. The offset represents a number of bytes in the uncompressed data stream. The whence parameter is defined as in lseek(2); the value SEEK_END is not supported. If the file is opened for reading, this function is emulated but can be extremely slow. If the file is opened for writing, only forward seeks are supported; gzseek then compresses a sequence of zeroes up to the new starting position.

gzseek returns the resulting offset location as measured in bytes from the beginning of the uncompressed stream, or -1 in case of error, in particular if the file is opened for writing and the new starting position would be before the current position.
gzrewind---rewind gzip file

Synopsis

```c
#include <zlib.h>
ZEXTERN int ZEXPORT gzrewind (gzFile file);
```

Description

Rewinds the given file. This function is supported only for reading.

gzrewind(file) is equivalent to (int)gzseek(file, 0L, SEEK_SET)
gztell---get gzip file position

Synopsis

#include <zlib.h>
ZEXTERN z_off_t ZEXPORT  gztell (gzFile file);

Description

Returns the starting position for the next gzread or gzwrite on the given compressed file. This position represents a number of bytes in the uncompressed data stream.

gztell(file) is equivalent to gzseek(file, 0L, SEEK_CUR)
gzeof---detect gzip file EOF

Synopsis

#include <zlib.h>
ZEXTERN int ZEXPORT gzeof (gzFile file);

Description

Returns 1 when EOF has previously been detected reading the given input stream, otherwise zero.
gzclose---close gzip file

Synopsis

```
#include <zlib.h>
ZEXTERN int ZEXPORT gzclose (gzFile file);
```

Description

Flushes all pending output if necessary, closes the compressed file and deallocates all the (de)compression state. The return value is the zlib error number (see function gzerror below).
gzerror---gzip file error

Synopsis

#include <zlib.h>
ZEXTERN const char * ZEXPORT gzerror (gzFile file, int *errnum);

Description

Returns the error message for the last error which occurred on the given compressed file. errnum is set to zlib error number. If an error occurred in the file system and not in the compression library, errnum is set to Z_ERRNO and the application may consult errno to get the exact error code.
Adler32---compute Adler-32 checksum

Synopsis

```c
#include <zlib.h>
ZEXTERN uLong ZEXPORT adler32 (uLong adler, const Bytef *buf, uInt len);
```

Description

Update a running Adler-32 checksum with the bytes buf[0..len-1] and return the updated checksum. If buf is NULL, this function returns the required initial value for the checksum.

An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much faster. Usage example:

```c
uLong adler = adler32(0L, Z_NULL, 0);

while (read_buffer(buffer, length) != EOF) {
    adler = adler32(adler, buffer, length);
}
if (adler != original_adler) error();
```
Synopsis

#include <zlib.h>
ZEXTERN uLong ZEXPORT crc32(uLong crc, const Bytef *buf, uInt len);

Description

Update a running crc with the bytes buf[0..len-1] and return the updated crc. If buf is NULL, this function returns the required initial value for the crc. Pre- and post-conditioning (one’s complement) is performed within this function so it shouldn’t be done by the application.

Usage example:

uLong crc = crc32(0L, Z_NULL, 0);

while (read_buffer(buffer, length) != EOF) {
    crc = crc32(crc, buffer, length);
}
if (crc != original_crc) error();