

System Interface Calls

This chapter contains the reference pages for POSIX.1 calls which have been implemented calls on INTERIX. They are organized alphabetically by the name of the call.

Note:

- No distinction is made between system calls and library function calls; all are listed.
- In the *SEE ALSO* section, you can distinguish between manuals by the formatting of the man page name:
 - name* — indicates a utility name
 - name()* — indicates a system interface name (this section)
 - <name>** — indicates a header file name

Each reference page has the following structure:

NAME	The name of the call and a brief description. Some reference pages describe several closely-related calls.
SYNOPSIS	An example of typical use, including header requirements.
DESCRIPTION	A longer description of the call, what it does, and its arguments.
RETURN VALUE	The meanings of the possible return values. This section isn't included if the call is of type void .
ERRORS	A description of possible values of <i>errno</i> set by the call. This section isn't included if the call doesn't set <i>errno</i> .
EXAMPLES	Examples of the use of the function.
SEE ALSO	References to manual pages containing related information, usually other relevant calls or commands.

Sections may be included or excluded as necessary.

These pages are a simple quick reference to the calls in POSIX.1, not a tutorial. For more detail on the library functions, please refer to your compiler documentation.

Much of the text of these reference pages was derived from the reference pages shipped with the 4.4BSD distribution and as such bear a University of California copyright. (Please see the Copyrights document for complete information.)

NAME

errno – system error numbers and errno values

SYNOPSIS

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

DESCRIPTION

This section provides an overview of the system error returns.

ERRORS

Nearly all of the system calls provide an error number in the external variable *errno*, which is defined as:

```
extern int errno
```

When a system call detects an error, it returns an integer value indicating failure (usually -1) and sets the variable *errno* accordingly. This allows interpretation of the failure on receiving a -1 and to take action accordingly. Successful calls never set *errno*; once set, it remains until another error occurs. It should only be examined after an error. Note that a number of system calls overload the meanings of these error numbers, and that the meanings must be interpreted according to the type and circumstances of the call.

The following is a complete list of the errors and their names as given in **<sys/errno.h>**. They are listed in alphabetical order by symbolic name.

- | | |
|--------------------------|--|
| [E2BIG] | <i>Arg list too long.</i>
The number of bytes used for the argument and environment list of the new process exceeded the current limit of {ARG_MAX} in <limits.h> . |
| [EACCES] | <i>Permission denied.</i>
An attempt was made to access a file in a way forbidden by its file access permissions. |
| [EADDRINUSE] | <i>Address already in use.</i>
Normally, each address can only be used once. |
| [EADDRNOTAVAIL] | <i>Address cannot be assigned.</i>
Normally results from an attempt to create a socket with an address not on this machine. |
| [EAFNOSUPPORT] | <i>Address family not supported by protocol family.</i>
The address was incompatible with the requested protocol. |
| [EAGAIN] [EWOULDBLOCK] | <i>Resource temporarily unavailable.</i>
This is a temporary condition and later calls to the same routine may complete normally. |
| [EALREADY] | <i>Operation already in progress.</i>
An operation was attempted on a non-blocking object that already had an operation in progress. |
| [EBADF] | <i>Bad file descriptor.</i>
A file descriptor argument was out of range, referred to no open file, or a read (write) request was made to a file that was only open for writing (reading). |

[EBUSY]	<i>Resource busy.</i> An attempt to use a system resource which was in use at the time in a manner which would have conflicted with the request.
[ECHILD]	<i>No child processes.</i> A <code>wait()</code> or <code>waitpid()</code> function was executed by a process that had no existing or unwaited-for child processes.
[ECONNABORTED]	<i>Software caused connection abort.</i> A connection abort was caused internal to your host machine
[ECONNREFUSED]	<i>Connection refused.</i> No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.
[ECONNRESET]	<i>Connection reset by peer.</i> A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot.
[EDEADLK] [EDEADLOCK]	<i>Resource deadlock avoided.</i> An attempt was made to lock a system resource that would have resulted in a deadlock situation.
[EDESTADDRREQ]	<i>Destination address required.</i> An operation on a socket requires a destination address.
[EDOM]	<i>Numerical argument out of domain.</i> A numerical input argument was outside the defined domain of the mathematical function.
[EDQUOT]	<i>Disk quota exceeded.</i> One of the following failed because the user's quota of disk space was exhausted: a write to an ordinary file, the creation of a directory or the creation of a directory entry.
[EEXIST]	<i>File exists.</i> An existing file was mentioned in an inappropriate context, for instance, as the new link name in a <code>link()</code> function.
[EFAULT]	<i>Bad address.</i> The system detected an invalid address in attempting to use an argument of a call.
[EFBIG]	<i>File too large.</i> The size of a file exceeded the maximum size, defined for Windows NT as $2^{32}-1$ bytes.
[EFTYPE]	<i>Inappropriate file type or format.</i> An operation was attempted on a file whose type or format would not support it.
[EHOSTDOWN]	<i>Host is down.</i> A socket operation failed because the destination host was down.
[EHOSTNOTFOUND]	<i>Host could not be found.</i> A socket operation was attempted to a host that could not be located.

[EHOSTUNREACH]	<i>No route to host.</i> A socket operation was attempted to an unreachable host.
[EILSEQ]	<i>Invalid multi-byte character.</i> The call contained an multi-byte character that was invalid in the current locale.
[EINPROGRESS]	<i>Operation now in progress.</i> An operation that takes a long time to complete, such as a <i>connect()</i> , was attempted on a non-blocking object.
[EINTR]	<i>Interrupted function call.</i> An asynchronous signal (such as SIGINT) was caught by the process during the execution of an interruptible function. If the signal handler performs a normal return, the interrupted function call will seem to have returned the error condition.
[EINVAL]	<i>Invalid argument.</i> Some invalid argument was supplied. (For example, specifying an undefined signal to a <i>signal()</i> or <i>kill()</i> function).
[EIO]	<i>Input/output error.</i> Some physical input or output error occurred. This error will not be reported until a subsequent operation on the same file descriptor and may be lost (over written) by any subsequent errors.
[EISCONN]	<i>Socket is already connected.</i> A connect request was made on an already connected socket; or, a <i>sendto</i> or <i>sendmsg</i> request on a connected socket specified a destination when already connected.
[EISDIR]	<i>Is a directory.</i> An attempt was made to open a directory with write mode specified.
[EMFILE]	<i>Too many open files.</i> The number of files open exceeds {OPEN_MAX}.
[EMLINK]	<i>Too many links.</i> Maximum allowable hard links to a single file has been exceeded (limit depends upon the filesystem; a POSIX-conformant filesystem supports at least 8). You can check the maximum number of links allowed on the system with the command getconf LINK_MAX .
[EMSGSIZE]	<i>Message too long.</i> Message sent on a socket was too long for an internal buffer or another limit.
[ENAMETOOLONG]	<i>File name too long.</i> File name exceeded {PATH_MAX} or a component of a file name exceeded {NAME_MAX}. For truly portable software, it may not be safe to rely on hard-coded values of {NAME_MAX} and {PATH_MAX}; on a networked system, the values may change depending on the filesystem. A safer way to check for the values on the current filesystem is to use <i>pathconf()</i> or <i>fpathconf()</i> before constructing a file name.

[ENETDOWN]	<i>Network is down.</i> A socket operation encountered a dead network.
[ENETRESET]	<i>Network dropped connection on reset.</i> The host you were connected to crashed and rebooted.
[ENETUNREACH]	<i>Network is unreachable.</i> A socket operation was attempted to an unreachable network.
[ENFILE]	<i>Too many open files in system.</i> Maximum number of file descriptors allowable on the system has been reached and a requests for an open cannot be satisfied until at least one has been closed.
[ENOBUFFS]	<i>No buffer space available.</i> An operation on a socket or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.
[ENODATA]	<i>No message available.</i> No data message was available.
[ENODEV]	<i>Operation not supported by device.</i> An attempt was made to apply an inappropriate function to a device, for example, trying to read a write-only device such as a printer.
[ENOENT]	<i>No such file or directory.</i> A component of a specified pathname did not exist, or the pathname was an empty string.
[ENOEXEC]	<i>Exec format error.</i> A request was made to execute a file that, although it has the appropriate permissions, was not in the format required for an executable file.
[ENOLCK]	<i>No locks available.</i> No locks were available.
[ENOMEM]	<i>Cannot allocate memory.</i> The new process image required more memory than was allowed by the hardware or by system-imposed memory management constraints.
[ENOPROTOOPT]	<i>Protocol not available.</i> A bad option or level was specified for <i>getsockopt()</i> or <i>setsockopt()</i> .
[ENOSPC]	<i>Device out of space.</i> A <i>write()</i> to an ordinary file, the creation of a directory, or the creation of a directory entry failed because no more disk blocks were available on the file system, or the allocation of a file serial number for a newly created file failed because no more numbers were available on the file system.
[ENOSYS]	<i>Function not implemented.</i> Attempted a system call that was not available on this system.
[ENOTBLK]	<i>Not a block device.</i> A block device operation was attempted on a non-block device or file.

[ENOTCONN]	<i>Socket is not connected.</i> An request to send or receive data was disallowed because the socket was not connected and (when sending on a datagram socket) no address was supplied.
[ENOTDIR]	<i>Not a directory.</i> A component of the specified pathname existed, but it was not a directory, when a directory was expected.
[ENOTEMPTY]	<i>Directory not empty.</i> An operation such as an unlink was attempted on a directory that was not empty.
[ENOTSOCK]	<i>Not a socket.</i> A socket function was attempted on a descriptor that was not a socket.
[ENOTTY]	<i>Inappropriate I/O control operation for device.</i> A control function was attempted for a file or special device for which the operation was inappropriate.
[ENXIO]	<i>No such device or address.</i> Input or output on a special file referred to a device that did not exist, or made a request beyond the limits of the device. This error may also occur when, for example, a tape is not on-line or no disk is loaded in a drive.
[EOPNOTSUPP]	<i>Operation not supported on socket.</i> The type of socket (either address family or protocol) does not support the operation.
[EPERM]	<i>Operation not permitted.</i> An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resources.
[EPFNOSUPPORT]	<i>Protocol family not supported.</i> The protocol family has not been configured or no implementation for it exists.
[EPIPE]	<i>Broken pipe.</i> A write on a socket, pipe or FIFO for which there is no process to read the data.
[EPROCLIM]	<i>Too many processes.</i> The operation would cause the system to exceed the limit on the number of processes.
[EPROTONOSUPPORT]	<i>Protocol not supported.</i> The system does not support the specified protocol.
[EPROTOTYPE]	<i>Protocol wrong for socket.</i> The requested protocol doesn't support the semantics of the socket.
[ERANGE]	<i>Numerical result out of range.</i> A numerical result of the function was too large to fit in the available space (perhaps exceeded precision).

[EROFS]	<i>Read-only file system.</i> An attempt was made to modify a file or directory was made on a file system that was read-only at the time.
[ESHUTDOWN]	<i>Cannot send after socket shutdown.</i> A request to send data was disallowed because the socket had already been shut down.
[ESOCKTNOSUPPORT]	<i>Socket type not supported.</i> The system does not support the specified socket type.
[ESPIPE]	<i>Illegal seek.</i> An <i>lseek()</i> function was issued on a socket, pipe or FIFO.
[ESRCH]	<i>No such process.</i> No process could be found corresponding to that specified by the given process ID.
[ETIMEDOUT]	<i>Operation timed out.</i> A connect or send request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)
[ETOOMANYREFS]	<i>Too many references.</i> There were too many references; the system could not splice (System V).
[ETXTBUSY]	<i>Unkown error.</i> On INTERIX, this is an unknown error. Traditionally, it means that an attempt was made to execute a pure-procedure program that is currently open for writing, or an attempt was made to open for writing a pure-procedure program while it was being executed.
[EXDEV]	<i>Improper link.</i> A hard link to a file on another file system was attempted.

NAME

abort() – cause abnormal program termination

SYNOPSIS

```
#include <stdlib.h>

void abort (void)
```

DESCRIPTION

The *abort()* function terminates the program “abnormally.” If the SIGABRT signal handler is set to SIG_DFL or SIG_IGN, or if the handler returns, *abort()* has the effect of *fclose()* on every open stream before the program terminates. Otherwise, *abort()* has no effect on streams.

RETURN VALUES

The *abort* function never returns.

SEE ALSO

sigaction(), *exit()*

NAME

`abs()` – integer absolute value function

SYNOPSIS

```
#include <stdlib.h>

int abs (int j)
```

DESCRIPTION

The `abs()` function computes the absolute value of the integer *j*.

RETURN VALUES

The `abs()` function returns the absolute value.

SEE ALSO

`cabs()`, `floor()`, `labs()`

NAME

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.

If *addr* and *addrlen* are both non-NULL, *accept()* fills the argument *addr* 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, such as SOCK_STREAM.

If either argument is null, no information is returned.

You can *select()* a socket for the purposes of doing an *accept()* by selecting it for read.

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

SEE ALSO

bind(), *connect()*, *listen()*, *select()*, *socket()*

NAME

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 that the file named by *path* is actually accessible in terms of the the access permissions indicated by *mode*. *Mode* is the bitwise inclusive OR of the access permissions to be checked; the permissions are:

R_OK Read permission

W_OK Write permission

X_OK Execute/search permission

F_OK File existence

The function checks all of *path*'s components for access permissions (including F_OK).

The function uses the program's *real* user ID, not the effective user ID; it's normally used by *setuid* programs to check their actions.

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

The *access()* function returns 0 on success, or –1 if *path* cannot be found or if any of the desired access modes wouldn't be granted.

ERRORS

Access to the file is denied if:

- | | |
|----------------|--|
| [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] | <i>Path</i> points outside the process's allocated address space. |
| [EINVAL] | The pathname is not valid. |
| [EIO] | An I/O error occurred while reading from or writing to the file system. |
| [ENAMETOOLONG] | A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters. |
| [ENOENT] | The named file does not exist. |
| [ENOTDIR] | A component of the path prefix is not a directory. |

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

SEE ALSO

chmod(), *stat()*

NAME

acosh() – inverse hyperbolic cosine function

SYNOPSIS

```
#include <math.h>
```

```
double acosh (double x)
```

DESCRIPTION

The *acosh()* function computes the inverse hyperbolic cosine of the real argument *x*. For a discussion of error due to roundoff, see *math*.

RETURN VALUES

The *acosh()* function returns the inverse hyperbolic cosine of *x*.

ERRORS

The *acosh()* function fails and sets *errno* for these reasons:

[EDOM] The value of *x* is less than 1.0.

SEE ALSO

asinh(), *atanh()*, *exp()*, *infnan()*, *math()*

NAME

acos() – arc cosine function

SYNOPSIS

```
#include <math.h>

double acos (double x)
```

DESCRIPTION

The *acos()* function computes the principal value of the arc cosine of *x*. A domain error occurs for arguments not in the range $[-1, +1]$. For a discussion of error due to roundoff, see *math()*.

RETURN VALUES

The *acos()* function returns the arc cosine in the range $[0, \pi]$ radians.

If *x* is outside that range, *acos(x)* sets the global variable *errno* to [EDOM] and returns an indefinite.

ERRORS

acos() can set *errno* to:

[EDOM] The value of *x* was less than -1 or greater than 1 .

SEE ALSO

sin(), *cos()*, *tan()*, *asin()*, *atan()*, *atan2()*, *sinh()*, *cosh()*, *tanh()*

NAME

alarm() – set signal timer alarm

SYNOPSIS

```
#include <unistd.h>

unsigned int alarm (unsigned int seconds)
```

DESCRIPTION

The *alarm()* function sets a countdown timer that waits *seconds* before asserting the terminating signal SIGALRM; *alarm()* returns immediately after setting the timer.

If there is a pending *alarm()* request, the call to *alarm()* replaces the prior call. In this case, the *alarm()* function returns the number of seconds left on the prior call's timer.

To cancel an alarm without setting a new one, pass 0 as the *seconds* argument.

The maximum number of *seconds* allowed is the maximum value of an **unsigned int**.

RETURN VALUE

If there is no other *alarm()* request pending, *alarm()* returns 0 after setting the timer.

If there is another *alarm()* request pending, *alarm()* returns the number of seconds remaining until the previous request would have generated a SIGALRM signal.

SEE ALSO

sigaction(), *signal()*, *sleep()*

NAME

asctime() – transform binary date and time value to ASCII

SYNOPSIS

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

char * asctime (const struct tm *tm)
```

DESCRIPTION

The *asctime()* function a timer structure *tm* pointed at by **tm* to the form

Thu Nov 24 18:22:48 1986\n\0

External declarations as well as the **tm** structure definition are in the **<time.h>** include file. The **tm** structure includes at least the following fields:

int	<i>tm_sec;</i>	seconds (0 – 60)
int	<i>tm_min;</i>	minutes (0 – 59)
int	<i>tm_hour;</i>	hours (0 – 23)
int	<i>tm_mday;</i>	day of month (1 – 31)
int	<i>tm_mon;</i>	month of year (0 – 11)
int	<i>tm_year;</i>	year – 1900
int	<i>tm_wday;</i>	day of week (Sunday = 0)
int	<i>tm_yday;</i>	day of year (0 – 365)
int	<i>tm_isdst;</i>	is summer time in effect?

SEE ALSO

date

ctime(), *getenv()*, *gmtime()*, *localtime()*, *time()*, *tzset()*

NAME

asinh() – inverse hyperbolic sine function

SYNOPSIS

```
#include <math.h>
```

```
double asinh (double x)
```

DESCRIPTION

The *asinh()* function computes the inverse hyperbolic sine of the real argument *x*. For a discussion of error due to roundoff, see *math()*.

RETURN VALUES

The *asinh()* function returns the inverse hyperbolic sine of *x*.

SEE ALSO

acosh(), *atanh()*, *exp()*, *infnan()*, *math*

NAME

asin() – arc sine function

SYNOPSIS

```
#include <math.h>
```

```
double asin (double x)
```

DESCRIPTION

The *asin()* function computes the principal value of the arc sine of *x*. A domain error occurs for arguments not in the range $[-1, +1]$.

RETURN VALUES

The *asin()* function returns the arc sine in the range $[-\pi/2, +\pi/2]$ radians.

On failure (an argument less than -1 or greater than $+1$), *asin()* returns an indefinite value and sets the global variable *errno*.

ERRORS

Asin() can set *errno* to the following value:

[EDOM]	Argument is out of the domain of the function $[-1,1]$.
--------	--

SEE ALSO

acos(), *atan()*, *atan2()*, *cos()*, *cosh()*, *sin()*, *sinh()*, *tan()*, *tanh()*

NAME

assert() – expression verification macro

SYNOPSIS

```
#include <assert.h>
```

```
assert (expression)
```

DESCRIPTION

The *assert()* macro tests the given *expression*. If it is true, *assert()* does nothing. If it is false, the calling process is terminated, a diagnostic message is written to the stderr and the function *_exit()* is called, effectively terminating the program.

At compile time, you may remove the *assert()* macro by #defining the **NDEBUG** macro before the inclusion of **<assert.h>**, usually by giving the compiler the **-DNDEBUG** option (see your compiler documentation).

DIAGNOSTICS

The following diagnostic message is written to stderr if *expression* is false:

```
"assertion \"%s\" failed: file \"%s\", line %d0, \\  
  \"expression\", __FILE__, __LINE__);
```

NAME

atan2() – arc tangent function of two variables

SYNOPSIS

```
#include <math.h>

double atan2 (double y, double x)
```

DESCRIPTION

The *atan2()* function computes the principal value of the arc tangent of y/x , using the signs of both arguments to determine the quadrant of the return value.

RETURN VALUES

The *atan2()* function, if successful, returns the arc tangent of y/x in the range $[-\pi, \pi]$ radians. If both x and y are zero, the global variable *errno* is set.

ERRORS

Atan2() can fail for the following reasons:

[EDOM]	Both x and y were zero.
[ERANGE]	The result underflows.

NOTES

The function *atan2()* defines "if $x > 0$ " *atan2*(0,0) = 0" on a VAX despite that previously *atan2*(0,0) may have generated an error message. The reasons for assigning a value to *atan2*(0,0) are these:

1. Programs that test arguments to avoid computing *atan2*(0,0) must be indifferent to its value. Programs that require it to be invalid are vulnerable to diverse reactions to that invalidity on diverse computer systems.
2. The *atan2()* function is used mostly to convert from rectangular (x,y) to polar (r,θ) coordinates that must satisfy $x = r\cos\theta$ and $y = r\sin\theta$. These equations are satisfied when $(x=0,y=0)$ is mapped to $(r=0,\theta=0)$ on a VAX. In general, conversions to polar coordinates should be computed thus:

```
r      := hypot(x,y);    ... :=  $\sqrt{x^2+y^2}$ 
 $\theta$  := atan2(y,x).
```

3. The foregoing formulas need not be altered to cope in a reasonable way with signed zeros and infinities on a machine that conforms to IEEE 754; the versions of *hypot()* and *atan2()* provided for such a machine are designed to handle all cases. That is why *atan2*($\pm 0,-0$) = $\pm$ for instance. In general the formulas above are equivalent to these:

```
r :=  $\sqrt{x*x+y*y}$ ;  if r = 0 then x := copysign(1,x);
```

SEE ALSO

acos(), *asin()*, *atan()*, *cos()*, *cosh()*, *math()*, *sin()*, *sinh()*, *tan()*, *tanh()*

NAME

atanh() – inverse hyperbolic tangent function

SYNOPSIS

```
#include <math.h>
```

```
double atanh (double x)
```

DESCRIPTION

The *atanh()* function computes the inverse hyperbolic tangent of the real argument *x*. For a discussion of error due to roundoff, see *math()*.

RETURN VALUES

The *atanh()* function returns the inverse hyperbolic tangent of *x* if successful. On error, it returns *NaN* and sets *errno* to indicate the error.

ERRORS

The *atanh()* function can set the following errors:

[EDOM]	The argument is not a number, or the absolute value of <i>x</i> is equal to 1.0.
--------	--

SEE ALSO

acosh(), *asinh()*, *exp()*, *infnan()*, *math*

NAME

atan() – arc tangent function of one variable

SYNOPSIS

```
#include <math.h>

double atan (double x)
```

DESCRIPTION

The *atan()* function computes the principal value of the arc tangent of *x*.

RETURN VALUES

The *atan()* function returns the arc tangent in the range $[-\pi/2, +\pi/2]$ radians. If *x* is 0, *atan()* returns 0.

ERRORS

The *atan()* function can fail for the following reasons:

[EDOM] The argument *x* is *NaN*.

[ERANGE] The result underflows.

SEE ALSO

acos(), *asin()*, *atan2()*, *cos()*, *cosh()*, *sin()*, *sinh()*, *tan()*, *tanh()*,

NAME

atexit() – register a function to be called on exit

SYNOPSIS

```
#include <stdlib.h>

int atexit (void (*function)(void))
```

DESCRIPTION

The *atexit()* function registers the given *function* to be called at program exit, whether via *exit()* or via return from the program's *main()*. Functions so registered are called in reverse order; no arguments are passed. At least 32 functions can always be registered, and more are allowed as long as sufficient memory can be allocated.

RETURN VALUES

On success, the *atexit()* function returns 0; on failure, it returns –1 and sets the value of *errno*.

ERRORS

atexit() can generate the following error:

[ENOMEM]	No memory was available to add the function to the list. The existing list of functions is unmodified.
----------	--

SEE ALSO

exit()

NAME

atof() – convert ASCII string to double

SYNOPSIS

```
#include <stdlib.h>
```

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

DESCRIPTION

The *atof()* function converts the initial portion of the string pointed to by *nptr* to **double** representation.

It is equivalent to:

```
strtod(nptr, (char **)NULL);
```

SEE ALSO

atoi(), *atol()*, *strtod()*, *strtol()*, *strtoul()*

NAME

atoi() – convert ASCII string to integer

SYNOPSIS

```
#include <stdlib.h>

int atoi (const char *nptr)
```

DESCRIPTION

The *atoi()* function converts the initial portion of the string pointed to by *nptr* to **integer** representation.

It is equivalent to:

```
(int)strtol(nptr, (char **)NULL, 10);
```

SEE ALSO

atof(), *atol()*, *strtod()*, *strtol()*, *strtoul()*

NAME

atol() – convert ASCII string to long integer

SYNOPSIS

```
#include <stdlib.h>

long atol (const char *nptr)
```

DESCRIPTION

The *atol()* function converts the initial portion of the string pointed to by *nptr* to **long integer** representation.

It is equivalent to:

```
strtol(nptr, (char **)NULL, 10);
```

SEE ALSO

atof(), *atoi()*, *strtod()*, *strtol()*, *strtoul()*

NAME

authenticateuser() – validate user information

SYNOPSIS

```
#include <opennt/security.h>
```

```
int authenticateuser (const struct usersec userdata)
```

DESCRIPTION

The *authenticateuser()* function validates the data in the data structure *userdata*. It returns 1 if the information is valid and 0 if it is not. This function is often used to validate information before calling one of the *exec_asuser* functions.

The *userdata* argument is a user security structure, **struct usersec**, which is defined in **<security.h>**:

```
struct usersec {  
    char *    user;  
    char *    domain;  
    char *    password;  
    int       logontype;        Defaults to 0  
    int       logonprovider;    Defaults to 0  
};
```

The **user** points to the user's logon name.

The **domain** points to the user's domain.

The **password** points to the user's password.

The **logontype** and **logonprovider** members are reserved for future use; for now, they should left at the default value of 0.

RETURN VALUES

If the information in the **struct usersec** structure is authentic and valid, *authenticateuser()* returns 1. Otherwise, it returns 0 and sets *errno* to indicate the error.

ERRORS

[EFAULT]	The <i>userdata</i> parameter is not in a valid part of the user address space.
[EINVAL]	One of the members of the structure was invalid, possibly too long.
[ENOMEM]	Not enough memory.
[EPERM]	The <i>user-domain-password</i> combination is invalid.

SEE ALSO

exec_asuser()

NAME

bcmp() – compare byte string

SYNOPSIS

```
#include <strings.h>
```

```
int bcmp (const void *b1, const void *b2, size_t len)
```

DESCRIPTION

The *bcmp()* function compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *len* bytes long. Zero-length strings are always identical.

The strings may overlap.

SEE ALSO

bcmp(), *memcmp()*, *strcasemp()*, *strcmp()*, *strcoll()*, *strxfrm()*

NAME

bcopy() – copy byte string

SYNOPSIS

```
#include <strings.h>
```

```
void bcopy (const void *src, void *dst, size_t len)
```

DESCRIPTION

The *bcopy()* function copies *len* bytes from string *src* to string *dst*. The two strings may overlap. If *len* is zero, no bytes are copied.

SEE ALSO

memcpy(), *memmove()*, *strcpy()*, *strncpy()*

NAME

`bind()` – bind a name to a socket

SYNOPSIS

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

int bind (int s, struct sockaddr *addr, int addrlen)
```

DESCRIPTION

Bind() assigns an address *addr* to an unnamed socket *s*. 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.

The arguments are:

s The socket to be bound.
–addr A pointer to a socket address structure. INTERIX supports only TCP/IP sockets.
addrlen The length of the **sockaddr** structure pointed to by *addr*.

RETURN VALUES

If the bind is successful, the function returns 0. If there is an error, the function returns –1, and sets *errno* to indicate the error.

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.
[EINVAL]	The socket is already bound to an address.
[EACCES]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The <i>name</i> parameter is not in a valid part of the user address space.

The following errors are specific to sockets using the AF_LOCAL protocol family (previously known as AF_UNIX).

[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 {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	A prefix component of the path name does not exist.
[EIO]	An I/O error occurred while making the directory entry or allocating the file serial number.

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

[EISDIR] An empty pathname was specified.

SEE ALSO

connect(), listen(), socket(), getsockname()

NAME

bsearch() – binary search of a sorted table

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

The *bsearch()* function searches an array of *nmemb* objects, for a member that matches the object pointed to by *key*. *Base* points to the initial member of the array, and the size of each member of the array is specified by *size*.

The contents of the array should be in ascending sorted order according to the comparison function referenced by *compar*. (See *qsort()*). The *compar* routine is expected to have two arguments which point to the *key* object and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the *key* object is found, respectively, to be less than, to match, or be greater than the array member.

RETURN VALUES

The *bsearch()* function returns a pointer to a matching member of the array, or NULL if no match is found. If two members compare as equal, which member is matched is unspecified.

SEE ALSO

qsort()

NAME

bzero() – write zeroes to a byte string

SYNOPSIS

```
#include <strings.h>
```

```
void bzero (void *b, size_t len)
```

DESCRIPTION

The *bzero()* function writes *len* zero bytes to the string *b*. If *len* is zero, *bzero()* does nothing.

SEE ALSO

memset(), *swab()*

NAME

`calloc()` – allocate clean memory (zero initialized space)

SYNOPSIS

```
#include <stdlib.h>
```

```
void * calloc (size_t nmemb, size_t size)
```

DESCRIPTION

The `calloc()` function allocates space for an array of *nmemb* objects, each of whose size is *size*. The space is initialized to all bits zero.

RETURN VALUES

The `calloc()` function returns a pointer to the the allocated space if successful; otherwise it returns NULL.

SEE ALSO

free(), *malloc()*, *realloc()*

NAME

ceil() – smallest integral value greater than or equal to *x*

SYNOPSIS

```
#include <math.h>

double ceil (double x)
```

DESCRIPTION

The *ceil()* function computes the smallest integral value not less than *x*.

RETURN VALUES

The *ceil()* function returns a **double** that is the smallest integral value greater than or equal to *x*.

ERRORS

The *ceil()* function can fail for the following reasons:

[EDOM] The argument *x* is *NaN*.

[ERANGE] The result overflows.

SEE ALSO

abs(), *fabs()*, *floor()*

NAME

chdir() – change current working directory

SYNOPSIS

```
#include <unistd.h>

int chdir (const char *path)
```

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, (/).

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. |
| [EINVAL] | The pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters. |
| [ENOENT] | The named directory does not exist. |
| [EACCES] | Search permission is denied for any component of the pathname. |

NAME

chmod() – change mode of file

SYNOPSIS

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

int chmod (const char *path, mode_t mode)
```

DESCRIPTION

The function *chmod()* sets the file permission bits of the file specified by the pathname *path* to *mode*. *Chmod()* verifies that the process owner (user) either owns the file specified by *path*. A mode is created from OR'd permission bit masks defined in *<sys/stat.h>*:

Symbol	Value	Meaning
S_IRWXU	0000700	RWX mask for owner
S_IRUSR	0000400	R for owner
S_IWUSR	0000200	W for owner
S_IXUSR	0000100	X for owner
S_IRWXG	0000070	RWX mask for group
S_IRGRP	0000040	R for group
S_IWGRP	0000020	W for group
S_IXGRP	0000010	X for group
S_IRWXO	0000007	RWX mask for other
S_IROTH	0000004	R for other
S_IWOTH	0000002	W for other
S_IXOTH	0000001	X for other
S_ISUID	0004000	set user ID on execution
S_ISGID	0002000	set group ID on execution

Note that in the INTERIX implementation, the set-user-id and set-group-id bits are ignored.

Calling the *chmod()* function on a file with open file descriptors has no effect on the open file descriptors.

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

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

[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	<i>Path</i> points outside the process's allocated address space.
[EINVAL]	The pathname contains a character with the high-order bit set.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.

[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID does not match the owner of the file.
[EROFS]	The named file resides on a read-only file system.

NOTE

When you use *chmod()* to make a file writeable, the function will unset the Win32 Read-Only attribute, if it is set on the file. (If the Read-Only attribute is set, the *stat()* function will report that the file is not writeable, regardless of the permissions on the ACLs associated with the file.)

SEE ALSO

chmod

open(), *chown()*, *stat()*

NAME

chown() – change owner and group of a file

SYNOPSIS

```
#include <unistd.h>
```

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

DESCRIPTION

The owner ID and group ID of the file named by *path* or referenced by *fd* is changed as specified by the arguments *owner* and *group*. The owner of a file may change the *group* to a group of which he or she is a member.

Chown() clears the set-user-id and set-group-id bits on the file to prevent accidental or mischievous creation of set-user-id and set-group-id programs.

One of the owner or group ID's may be left unchanged by specifying it as -1.

RETURN VALUES

Zero is returned if the operation was successful; -1 is returned if an error occurs, with a more specific error code being placed in the global variable *errno*.

ERRORS

Chown() will fail and the file will be unchanged if:

[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 {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EPERM]	The effective user ID does not have the permissions required.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	<i>Path</i> 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

chgrp

chmod()

NAME

clearerr(), feof(), ferror(), fileno() – check and reset stream status

SYNOPSIS

```
#include <stdio.h>

void clearerr (FILE *stream)
int feof (FILE *stream)
int ferror (FILE *stream)
int fileno (FILE *stream)
```

DESCRIPTION

The function *clearerr()* clears the end-of-file and error indicators for the stream pointed to by *stream*.

The function *feof()* tests the end-of-file indicator for the stream pointed to by *stream*, returning non-zero if it is set. The end-of-file indicator can only be cleared by the function *clearerr()*.

The function *ferror()* tests the error indicator for the stream pointed to by *stream*, returning non-zero if it is set. The error indicator can only be reset by the *clearerr()* function.

The function *fileno()* examines the argument *stream* and returns its integer descriptor.

ERRORS

These functions should not fail and do not set the external variable *errno*.

SEE ALSO

open()

NAME

clock() – determine processor time used

SYNOPSIS

```
#include <time.h>

clock_t clock (void)
```

DESCRIPTION

The *clock()* function approximates the amount of processor time used since the invocation of the calling process, expressed as a **clock_t**.

To convert a **clock_t** value into seconds, divide by the macro `CLOCKS_PER_SECOND`.

RETURN VALUES

The *clock()* function returns the amount of time used; if an error occurs, the function returns `-1`.

NAME

close() – delete a descriptor

SYNOPSIS

```
#include <unistd.h>

int close (int d)
```

DESCRIPTION

The *close()* call deletes the file descriptor *d* and makes it available for re-use. If the calling process owns any outstanding record locks on the file indicated by *d*, the call unlocks them.

When all of the file descriptors pointing to a file are deleted, the file is removed. If the file is a pipe or a FIFO, any data remaining in the pipe or FIFO is discarded. 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()*, the associated naming information and queued data are discarded.

When a process exits, all associated file descriptors are freed, but since there is a limit on active descriptors per process, 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 *exec()*, 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. To do this, use the call:

```
fcntl(d, F_SETFD, 1)
```

which arranges that a descriptor will be closed after a successful *execve*; to restore the default (that is, not to close the descriptor), use the call:

```
fcntl(d, F_SETFD, 0)
```

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] | <i>D</i> is not an active descriptor. |
| [EINTR] | An interrupt was received. |

SEE ALSO

accept(), *socket()*, *exec()*, *fcntl()*, *open()*, *pipe()*

NAME

confstr() – get string-valued configurable variables

SYNOPSIS

```
#include <unistd.h>
```

```
size_t confstr (int name, char *buf, size_t len)
```

DESCRIPTION

The *confstr()* function provides a method for applications to get configuration defined string values.

The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file **<unistd.h>**. The *len* argument specifies the size of the buffer referenced by the argument *buf*. If *len* is non-zero, *buf* is a non-null pointer, and *name* has a value, up to *len* – 1 bytes of the value are copied into the buffer *buf*. The copied value is always null terminated.

The available values are as follows:

_CS_PATH

Return a value for the *PATH* environment variable that finds all the standard utilities.

When the macros *_ALL_SOURCE* or *_XOPEN_SOURCE* are defined, or if *_POSIX_C_SOURCE*=2, then these are also available:

_CS_SHELL

The POSIX shell.

_CS_INSTALLEDDIR

The directory in which INTERIX is installed.

_CS_TMPDIR

The INTERIX temporary directory.

_CS_ETCDIR

_CS_BINDIR

_CS_INCLUDEDIR

_CS_LIBDIR

_CS_USRDIR

_CS_PUBSDIR

RETURN VALUES

If the call to *confstr()* is not successful, –1 is returned and *errno* is set appropriately. Otherwise, if the variable does not have a configuration defined value, 0 is returned and *errno* is not modified. Otherwise, the buffer size needed to hold the entire configuration-defined value is returned. If this size is greater than the argument *len*, the string in *buf* was truncated.

ERRORS

The *confstr()* function may fail and set *error* for any of the errors specified for the library function *malloc()*.

In addition, the following errors may be reported:

[EINVAL] The value of the *name* argument is invalid.

NAME

connect() – initiate a connection on a socket

SYNOPSIS

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

int connect (int s, struct sockaddr *addr, int addrlen)
```

DESCRIPTION

The parameter *s* is a socket.

- If it is of type `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 `SOCK_STREAM`, this call attempts to make a connection to another socket.

The other socket is specified by *addr*, which is an address in the communications space of the socket. Each communications space interprets the *addr* parameter in its own way. Generally, stream sockets may successfully *connect()* only once; datagram sockets may use *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 `-1` is returned, and a more specific error code is stored in *errno*.

ERRORS

The *connect()* call fails if:

- | | |
|-----------------|--|
| [EBADF] | <i>S</i> is not a valid descriptor. |
| [ENOTSOCK] | <i>S</i> is a descriptor for a file, not a socket. |
| [EADDRNOTAVAIL] | The specified address is not available on this machine. |
| [EAFNOSUPPORT] | Addresses in the specified address family cannot be used with this socket. |
| [EISCONN] | The socket is already connected. |
| [ETIMEDOUT] | Connection establishment timed out without establishing a connection. |
| [ECONNREFUSED] | The attempt to connect was forcefully rejected. |
| [ENETUNREACH] | The network isn't reachable from this host. |
| [EADDRINUSE] | The address is already in use. |
| [EFAULT] | The <i>addr</i> parameter specifies an area outside the process address space. |

[EINPROGRESS] The socket is non-blocking and the connection cannot be completed immediately. It is possible to *select()* for completion by selecting the socket for writing.

[EALREADY] The socket is non-blocking and a previous connection attempt has not yet been completed.

The following errors are specific to sockets using the AF_LOCAL protocol family (previously known as AF_UNIX).

[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 {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} 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.

SEE ALSO

accept(), select(), socket(), getsockname()

NAME

cosh() – hyperbolic cosine function

SYNOPSIS

```
#include <math.h>

double cosh (double x)
```

DESCRIPTION

The *cosh()* function computes the hyperbolic cosine of *x*.

RETURN VALUES

The *cosh()* function returns the hyperbolic cosine.

If the argument is not a number, it returns *NaN* and sets *errno*. If the argument is $\pm\infty$, it returns $+\infty$ and sets *errno*.

ERRORS

The function *cosh()* can fail for the following reasons:

- | | |
|----------|-------------------------------------|
| [EDOM] | The argument is <i>NaN</i> . |
| [ERANGE] | The result would cause an overflow. |

SEE ALSO

acos(), *asin()*, *atan()*, *atan2()*, *cos()*, *sin()*, *sinh()*, *tan()*, *tanh()*, *math()*

NAME

cos() – cosine function

SYNOPSIS

```
#include <math.h>

double cos (double x)
```

DESCRIPTION

The `cos()` function computes the cosine of x (measured in radians). A large magnitude argument may yield a result with little or no significance.

RETURN VALUES

The `cos()` function returns the cosine value of x .

If the value of x is too large, the return value of `cos()` is indefinite.

ERRORS

The `cos()` call can set `errno` to the following errors:

[EDOM] The value of x was *NaN* or ∞ .

[ERANGE] The result would cause an underflow.

SEE ALSO

`asin()`, `acos()`, `atan()`, `atan2()`, `cosh()`, `sin()`, `sinh()`, `tan()`, `tanh()`

NAME

creat() – create a new file

SYNOPSIS

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

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

DESCRIPTION

The *creat()* call opens a file for writing. If the file doesn't exist, it is created with the permissions specified by *mode*; the owner and group IDs are set from the calling process's effective user and group IDs.

If the file does exist, it's truncated to zero length. The owner and group IDs aren't changed.

Creat() is the same as:

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

RETURN VALUES

On success, the *creat()* call returns an **integer** file descriptor which refers to the file. On failure, the call returns `-1` and sets *errno*.

ERRORS

The named file is opened unless:

[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]	The file does not exist and the directory in which it is to be created does not permit writing.
[EEXIST]	O_CREAT and O_EXCL were specified and the file exists.
[EINTR]	The <i>creat()</i> operation was interrupted by a signal.
[EISDIR]	The named file is a directory, and the arguments specify it is to be opened for writing.
[EMFILE]	The process has already reached its limit for open file descriptors.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
[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.
[ENOENT]	O_CREAT was not specified and the named file does not exist.
[ENOENT]	A component of the pathname that must exist does not exist.

[ENOSPC]	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 file serial numbers on the file system on which the file is being created.
[EIO]	An I/O error occurred while making the directory entry or allocating the file serial number for O_CREAT.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The named file resides on a read-only file system, and the file is to be modified.

SEE ALSO*open(), close()*

NAME

ctermid() – generate terminal pathname

SYNOPSIS

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

```
char * ctermid (char *buf)
```

DESCRIPTION

The *ctermid()* function generates a string that, when used as a pathname, refers to the current controlling terminal of the calling process.

If *buf* is the NULL pointer, the function returns a pointer to a static area. Otherwise, the pathname is copied into the memory referenced by *buf*. The argument *buf* is assumed to be at least {L_ctermid} (as defined in the include file **<stdio.h>**) bytes long.

RETURN VALUES

This implementation of the *ctermid()* function always returns an empty string.

NOTES

The current implementation always returns an empty string, as allowed under POSIX 1003.1-1990. A future implementation will behave as described above.

SEE ALSO

ttyname()

NAME

ctime() – transform binary date and time value to ASCII

SYNOPSIS

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

char * ctime (const time_t *clock)
```

DESCRIPTION

The function *ctime()* takes as an argument a time value representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970; see *time()*).

Ctime() function adjusts the time value for the current time zone and returns a pointer to a 26-character string of the form:

```
Thu Nov 24 18:22:48 1986\n\0
```

All the fields have constant width.

SEE ALSO

date

asctime(), *difftime()*, *getenv()*, *gmtime()*, *localtime()*, *mktime()*, *time()*

NAME

cuserid() – return user ID (login name) as a string

SYNOPSIS

```
#include <stdio.h>

char *cuserid (char *s)
```

DESCRIPTION

The *cuserid()* function returns the login name of the real or effective owner of the process.

If *s* is NULL, then the function returns a pointer to a static buffer, which may be overwritten by subsequent calls to *cuserid()*.

If *s* is not NULL, it's assumed to be a buffer that is at least {L_cuserid} bytes long; {L_cuserid} is defined in **<stdio.h>**.

RETURNS

If *s* is not NULL, *cuserid()* returns *s*; if *s* is NULL, *cuserid()* returns a pointer to an internal buffer.

If the login name was found, it will be in the appropriate buffer. If the login name could not be found, the buffer contains a null byte '\0'.

SEE ALSO

getlogin(), *getpwnam()*, *getpwuid()*, *getuid()*, *geteuid()*

NAME

daemon() – run in the background

SYNOPSIS

```
int daemon (int nochdir, int noclose)
```

DESCRIPTION

The *daemon()* call is an interface to allow a program to become a system daemon. The function causes the calling program to *fork()*; the parent exits and the child then performs a *setsid()*. This disassociates the process from its current process group, session, and controlling terminal. On successful completion of this call, the process is the session leader of a group in which it is the only member, and the session has no controlling terminal.

If *nochdir* is zero, the process changes its current working directory to the installed root of INTERIX. Otherwise the current working directory is unchanged.

If *noclose* is zero, the process redirects standard input, standard output, and standard error to **/dev/null**. Otherwise the standard file handles remain directed at the terminal that was previously the controlling terminal.

RETURN VALUES

The *daemon* function returns 0 on success; on failure, it returns -1 and sets the global variable *errno*.

ERRORS

The *daemon()* function can fail for the following reasons:

- | | |
|----------|---|
| [EAGAIN] | The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent. |
| [EAGAIN] | The system-imposed limit on the total number of processes under execution by a single user would be exceeded. |
| [EPERM] | The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process. |

SEE ALSO

fork(), *setsid()*

NAME

difftime() – compute difference between two times

SYNOPSIS

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

```
double difftime (time_t time1, time_t time0)
```

DESCRIPTION

The *difftime()* function returns the difference between two calendar times, (*time1* – *time0*), expressed in seconds.

External declarations as well as the **tm** structure definition are in the **<time.h>** include file.

SEE ALSO

date

asctime(), *ctime()*, *gettimeofday()*, *gmtime()*, *getenv()*, *localtime()*, *time()*

NAME

opendir(), readdir(), rewinddir(), closedir() – directory operations

SYNOPSIS

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

DIR * opendir (const char *filename)
struct dirent * readdir (DIR *dirp)
void rewinddir (DIR *dirp)
int closedir (DIR *dirp)
```

DESCRIPTION

These calls create, remove, read, and search directories. They make use of the **DIR** type, defined in the header **<dirent.h>**, which represents a *directory stream*. A directory stream is an ordered sequence of all the directory entries in a particular directory. Each directory entry represents a file.

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. *readdir()* returns *NULL* 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. It does *not* return entries for the directories *dot* (the current directory) or *dot-dot* (the parent directory).

The *rewinddir()* function resets the position of the named *directory stream* to the beginning of the directory.

The *closedir()* function closes the named *directory stream* and frees the structure associated with the *dirp* pointer, returning 0 on success.

RETURN VALUES

Opendir() returns a pointer to a **DIR** on success. On failure, it returns *NULL* and sets *errno*.

Readdir() returns a pointer to a **struct dirent** on success; when it reaches the end of the directory, it returns *NULL* and *doesn't* set *errno*. On failure, it returns *NULL* and *does* set *errno*.

Closedir() returns 0 on success; on error, it returns *-1* and sets *errno*.

Rewinddir() doesn't return a value.

EXAMPLE

To search the current directory for the entry "name":

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

ERRORS

Opendir() can set *errno* to the following values:

- | | |
|----------------|---|
| [EACCES] | The process doesn't have search permission for a component of the path <i>filename</i> , or it doesn't have read permission for the directory itself. |
| [EMFILE] | The process has too many file descriptors open. |
| [ENFILE] | The system has too many file descriptors open. |
| [ENAMETOOLONG] | A component of <i>filename</i> exceeded {NAME_MAX} characters, or the entire pathname exceeded {PATH_MAX} characters. |
| [ENOENT] | The directory named by <i>filename</i> doesn't exist, or <i>filename</i> is an empty string. |
| [ENOTDIR] | Some component of <i>filename</i> isn't a directory. |

Closedir() and *readdir()* can set *errno* to the following values:

- | | |
|---------|--|
| [EBADF] | <i>Dirp</i> doesn't refer to an open directory stream. |
|---------|--|

SEE ALSO

open(), *close()*, *lseek()*, *read()*

NAME

`div()` – return quotient and remainder from division

SYNOPSIS

```
#include <stdlib.h>
```

```
div_t div (int num, int denom)
```

DESCRIPTION

The `div()` function computes the value $num / denom$ and returns the quotient and remainder in a structure named **div_t** that contains two **int** members named *quot* and *rem*. If your application requires numbers to be rounded in the same way on all systems, use

```
div(a,b).quot
```

instead of

```
a/b
```

To get the remainder, use

```
div(a,b).rem
```

SEE ALSO

ldiv()

NAME

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 which typically refers to a file or a socket. 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, the descriptor is first deallocated as if a *close()* call were done first.

RETURN VALUES

Both functions return -1 if an error occurs, and both set the external variable *errno* to indicate 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

open(), *close()*, *fcntl()*, *pipe()*, *socket()*

NAME

endutxent(), getutxent(), getutxid(), getutxline(), pututxline(), setutxent() – user accounting database functions

SYNOPSIS

```
#include <utmpx.h>

void endutxent(void)
struct utmpx *getutxent (void)
struct utmpx *getutxid (const struct utmpx *id)
struct utmpx *getutxline (const struct utmpx *line)
struct utmpx *pututxline (const struct utmpx *utmpx)
```

DESCRIPTION

These functions provide access to the user accounting database, usually stored in the file **\$OPENNT_ROOT/var/adm/utmpx**. These functions are not reentrant.

The *getutxent()* function reads the next entry. If the **utmpx** file isn't open, the function opens it. If the function reaches the end of the **utmpx** file, it fails.

The *getutxid()* function searches forward in the database from the current point for a matching entry. The *id* is a **utmpx** structure; the match depends upon the value of the **ut_type** value in the structure.

- If the value is **BOOT_TIME**, **OLD_TIME**, or **NEW_TIME**, the search stops at the next entry with a matching value.
- If the value is **INIT_PRPROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS**, the search stops at the next entry with one of those types *and* whose **ut_id** member matches the **ut_id** member in *id*.

The *getutxid()* function fails if it reaches the end of the database without a match.

The *getutxline()* function searches forward in the database from the current point for an entry with a value of **ut_line** that matches the value in *line*, and that has a **ut_type** of **LOGIN_PROCESS** or **USER_PROCESS**. The function fails if it reaches the end of the database without finding a match.

The *pututxline()* function writes out the information in the structure *utmpx* to the database. The process must have the appropriate privileges to do this. It searches the database for a matching entry (using *getutxid()*); if it finds a match, the existing information is replaced with the information in *utmpx*. If it doesn't find a match, the new information is added at the end of the file.

The *setutxent()* function resets input to the beginning of the database.

The *endutxent()* function closes the database.

The functions *getutxid()* and *getutxline()* may cache the data. When using *getutxline()* to search for multiple occurrences, zero out the static data after each success, or *getutxline()* will repeatedly return a pointer to the same **utmpx** structure. (The implicit read done by *pututxline()* doesn't modify the static structure.)

The **utmpx** structure is described in the file **<utmpx.h>**. It consists of at least these members:

char	ut_user[]	user login name
char	ut_id[]	initialization process identifier
char	ut_line[]	device name
pid_t	ut_pid	process ID
short int	ut_type	type of entry — see list
struct timeval	ut_tv	time entry was made.

The **ut_type** member is used for matches, as described above. It can have the following values:

EMPTY	No valid information
BOOT_TIME	Time of system boot (not stored)
OLD_TIME	Time when system clock changed (not stored)
NEW_TIME	Time after system clock changed (not stored)
USER_PROCESS	Identifies a process
INIT_PROCESS	Process was spawned by the init process
LOGIN_PROCESS	Session leader of the logged-in user
DEAD_PROCESS	Session leader has exited

The values **BOOT_TIME**, **OLD_TIME**, **NEW_TIME**, and **INIT_PROCESS** do not show up in INTERIX systems.

RETURN VALUE

Upon completing successfully, the functions *getutxent()*, *getutxid()*, and *getutxline()* return a pointer to a **utmpx** structure which contains a copy of the requested entry. The structure is in a static area which may be overwritten by subsequent calls. On failure, they return a null pointer.

Upon completing successfully, the function *pututxline()* returns a pointer to a **utmpx** structure containing the user information entered in the user database. On failure, it returns a null pointer.

The *endutxent()* and *setutxent()* functions return no value.

ERRORS

The *pututxline()* function may fail for this reason:

[EPERM]	The process does not have privileges to write to \$OPENNT_ROOT/var/adm/utmpx .
---------	---

NOTES

This set of APIs is new to INTERIX with release 2.1. Programs written to use these APIs will not run on INTERIX 2.0 systems.

SEE ALSO

who

NAME

erf(), erfc() – error function operators

SYNOPSIS

```
#include <math.h>

double erf (double x)
double erfc (double x)
```

DESCRIPTION

These functions calculate the error function of x .

The *erf()* calculates the error function of x ; where

$$\text{erf}(x) := (2/\sqrt{\pi}) \int_0^x \exp(-t^2) dt.$$

The *erfc()* function calculates the complementary error function of x ; that is *erfc()* subtracts the result of the error function *erf(x)* from 1.0. This is useful, since for large x places disappear.

SEE ALSO

math

NAME

`err()`, `verr()`, `errx()`, `verrx()`, `warn()`, `vwarn()`, `warnx()`, `vwarnx()` – formatted error messages

SYNOPSIS

```
#include <err.h>

void err (int eval, const char *fmt, ...)
void verr (int eval, const char *fmt, va_list args)
void errx (int eval, const char *fmt, ...)
void verrx (int eval, const char *fmt, va_list args)
void warn (const char *fmt, ...)
void vwarn (const char *fmt, va_list args)
void warnx (const char *fmt, ...)
void vwarnx (const char *fmt, va_list args)
```

DESCRIPTION

The `err()` and `warn()` family of functions display a formatted error message on the standard error output. In all cases, the last component of the program name, a colon character, and a space are output first. If the `fmt` argument is not NULL, the formatted error message, a colon character, and a space are output. In the case of the `err()`, `verr()`, `warn()`, and `vwarn()` functions, the error message string affiliated with the current value of the global variable `errno` is output. In all cases, the output is followed by a newline character.

RETURN VALUES

The `err()`, `verr()`, `errx()`, and `verrx()` functions do not return, but exit with the value of the argument `eval`.

EXAMPLES

Display the current `errno` information string and exit:

```
if ((p = malloc(size)) == NULL)
    err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
    err(1, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
    errx(1, "too early, wait until %s", start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
    warnx("%s: %s: trying the block device",
        raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
    err(1, "%s", block_device);
```

SEE ALSO

`strerror()`

NAME

execl(), execl(), execlp(), execv(), execve(), execvp() – execute a file

SYNOPSIS

```
#include <unistd.h>
extern char **environ;

int execl (const char *path, const char *arg ...)
int execl (const char *path, const char *arg ... char *const envp[])
int execlp (const char *file, const char *arg ...)
int execv (const char *path, char *const argv[])
int execve (const char *path, char *const argv[], char *const envp[])
int execvp (const char *file, char *const argv[])
```

DESCRIPTION

The *exec* family of functions replaces the current process image with a new process image. The first argument is the name of the file to be executed; most of the functions require a full pathname, but *execlp()* and *execvp()* will search the directories named in the *PATH* environment variable.

The second argument to the function is the argument vector (represented in a C *main()* function by *argv*). The *execl** functions take a **const char *** argument, while the *execv** functions take a **char *const** array as the argument. The third argument, where present, is an environment for the new process image. For *execl()*, *execv()*, *execlp()*, and *execvp()*, the new process image inherits the environment of the calling process image, stored in *extern char **environ*.

The arguments are:

const char *path

For the functions *execl()*, *execv()*, *execlp()*, and *execve()*, the *path* is a pathname that points to the file to be executed. If the file doesn't exist or has the wrong permissions, the functions return -1 and set *errno*. The *path* can also specify a data file for an interpreter program; an *interpreter file* begins with a line of the form:

```
#! interpreter [arguments]
```

For more information about interpreter files, see “Interpreter Files” later in this reference page.

const char *file

For the functions *execlp()* and *execvp()*, the *file* is the name of the file to be executed; if *file* doesn't begin with a slash (/), the call constructs the pathname by searching the directories named in the *PATH* environment variable.

Once the file is found, the *exec* functions try to execute it as a POSIX process, then as a Win32 process, and last as an interpreter file.

const char *arg ...

The *arg* and subsequent ellipse can be thought of as *arg0, arg1, ..., argn*—a variable-length list of pointers to null-terminated character strings that represent the argument list available to the new program. The first argument should be the filename of the new process; the last argument should be a NULL pointer.

Execle() treats this argument slightly differently: after the NULL pointer, there is *another* character pointer which contains the new process's environment, as if it were declared as

```
char *const envp[]
```

`char *const argv[]`
An array of character pointers to null-terminated strings; the last member of the array is a NULL pointer. *argv*[0] should contain the name of the file being executed.

`char *const envp[]`
An array of character pointers to null-terminated strings that make up the environment array of the new process image.

The total space available for the combined argument and environment lists of the new process image is {ARG_MAX}.

When the call succeeds, it creates a new process image; if the new process is a C program, the entry point is *main(argc,argv)*, where *argc* is the number of elements in the argument list (not counting the terminating NULL pointer) and *argv* is the argument list itself. The functions also set up

```
extern char **environ
```

which contains the process's environment.

Once these calls have successfully completed, they mark the *st_atime* field of *path*, since it was successfully *open()*'ed.

Inherited Properties

Here are the guidelines as to which attributes and resources are and are not inherited by the new process:

Calling Process Attribute	Behavior
Open file descriptors	Remain open, unless close-on-exec flag FD_CLOEXEC is set
Directory streams	Closed
Signals set to default action	Set to default action
Signals set to be ignored	Set to be ignored
Signals set to be caught	Set to default action
Real user ID	Inherited
Real group ID	Inherited
Supplementary group ID	Inherited
Effective user ID	Inherited, unless <i>path</i> has set-user-ID set
Effective group ID	Inherited, unless <i>path</i> has set-group-ID set
Process ID	Inherited
Parent process ID	Inherited
Process group ID	Inherited
Session membership	Inherited
Time left until alarm clock signal	Inherited
Current working directory	Inherited

Calling Process Attribute	Behavior
Root directory	Inherited
File mode creation mask	Inherited
Process signal mask	Inherited
Pending signals	Inherited

If the *saved-set-user-ID* mode bit is set on the new process image file, then the effective user ID of the new process image is set to the owner ID of the file.

If the *saved-set-group-ID* mode bit is set on the new process image file, then the effective group ID of the new process image is set to the group ID of the file.

If `{_POSIX_SAVED_IDS}` is defined, the effective user ID and the effective group ID are saved for used by the *setuid()* function.

Interpreter Files

If the file specified by *path* is a text file and the first line has the form:

```
#! interpreter [arguments]
```

then the *exec* functions run the program specified by *interpreter* with the given *arguments*. The maximum length of the line is 255 characters; the interpreter can be invoked with as many arguments as will fit on the command line.

The *interpreter* line specifies an interpreter according to the following rules:

1. If the *interpreter* string begins with `//` it is taken as an absolute path to the interpreter.
2. If the string begins with `/`, INTERIX searches in the INTERIX root directory space.
3. If the file is not found there, the path is assumed to be on the local drive.
4. If the file is still not found, the *exec* fails.

The first line can be recursive; that is, *interpreter* can specify a file that begins with a `#!` interpreter line, and so on, nested eight levels. For each level, the *exec** functions first try to create a process image of the file as a POSIX executable, then as a Win32 executable, and last as an interpreter file.

WIN32 EXEC LIMITATIONS

A POSIX process can *exec* a Win32 image file, but there are several limitations.

The POSIX process will remain in the POSIX subsystem's process table until the corresponding Win32 process exits. The exit status of this process will always indicate a normal return, as if by a call to *exit()*, with the exit value being the low order 8 bits of the Win32 process's 32 bit exit status. There is currently no way to detect the fact that a Win32 process was terminated by an exception.

None of the processes spawned by the Win32 process will show up in the POSIX subsystem's process table.

All POSIX signals sent to the Win32 process are ignored.

If the Win32 process is a console application, then the only file descriptors that can be inherited are file descriptors 0, 1 and 2, corresponding to the Win32 standard input, standard output and standard error. The only types of file descriptors that can be inherited are those associated with a regular file, a controlling console tty, a controlling

pseudo tty (stdout and stderr only), **/dev/null**, or a pipe.

If the Win32 process is a gui application, then no file descriptors are inherited.

If the Windows NT security token (which corresponds to the POSIX real uid) of the Win32 process is that of the interactive user, then any windows created by the process will be visible on the screen. Otherwise, windows will be invisible and non-interactive. Win32 processes spawned from a *telnet* session, a POSIX service program, or a process whose real uid was set by a call to *exec*_asuser*, do not have the ability to create interactive windows.

A POSIX process cannot exec DOS, Win16 or OS/2 image files.

RETURN VALUES

On success, the *exec* functions don't return. If they fail, they set *errno* and return -1.

ERRORS

The *exec* functions can set *errno* to any of the following values:

[E2BIG]	The argument list was too long.
[EACCES]	Search permission was denied for a directory in the <i>path</i> or the new process image file wasn't executable.
[ENAMETOOLONG]	Some component of <i>path</i> was greater than {NAME_MAX}, or the entire pathname exceeded {PATH_MAX}.
[ENOENT]	One or more components of <i>path</i> doesn't exist, or the <i>path</i> or <i>file</i> argument is an empty string.
[ENOEXEC]	For <i>execl()</i> , <i>execv()</i> , <i>execle()</i> , and <i>execve()</i> , the new process image file has the appropriate access permission, but is not in the proper format; or, more than eight levels of recursion have been encountered in processing an interpreter file.
[ENOMEM]	Not enough memory to load the new process image.
[ENOTDIR]	Some component of the path prefix of <i>path</i> is not a directory.

SEE ALSO

sh

exit(), *fork()*

NAME

execl_asuser(), execlp_asuser(), execl_e_asuser(), execv_asuser(), execve_asuser(),
execvp_asuser() – execute a file as another user

SYNOPSIS

```
#include <opennt/security.h>
extern char **environ;

int execl_asuser (const struct usersec userdata, const char *path,
                  const char *arg ...)
int execlp_asuser (const struct usersec userdata, const char *file,
                  const char *arg ...)
int execl_e_asuser (const struct usersec userdata, const char *path,
                   const char *arg ... char *const envp[])
int execv_asuser (const struct usersec userdata, const char *path,
                 char *const argv[])
int execve_asuser (const struct usersec userdata, const char *path,
                  char *const argv[], char *const envp[])
int execvp_asuser (const struct usersec userdata, const char *file,
                  char *const argv[])
```

DESCRIPTION

The *exec_asuser* family of functions replaces the current process image with a new process image, run as the user specified in the structure *userdata*. These functions do a similar job to the calls *setuid()*/*exec()* on a BSD or SVR4 system. However, they do not behave identically, so do not rely heavily on this analogy.

Except for the initial *userdata* argument, the arguments to the *exec_asuser* functions are identical to the corresponding *exec()* functions; see that reference page for more details.

The *exec_asuser* functions are required because the security on Windows NT prevents use of the *setuid* and *getguid* bits on files; there is no way to get the appropriate authority. Unlike a *setuid()* call, you must know the user's password in order to execute a file as that user.

The first argument is a user security structure, **struct usersec**, which is defined in **<sec.h>**:

```
struct usersec {
    char *    user;
    char *    domain;
    char *    password;
    int       logontype;           Defaults to 0
    int       logonprovider;      Defaults to 0
};
```

The new process image maintains the following attributes and resources:

Calling Process Attribute**Behavior**

Remain open, unless close-on-exec flag *FD_CLOEXEC* is set

Calling Process Attribute	Behavior
Open file descriptors	
Directory streams	Closed
Signals set to default action	Set to default action
Signals set to be ignored	Set to be ignored
Signals set to be caught	Set to default action
Real user ID	Determined by usersec
Real group ID	Determined by usersec
Supplementary group ID	Determined by usersec
Effective user ID	Determined by usersec
Effective group ID	Determined by usersec
Process ID	Inherited
Parent process ID	Inherited
Process group ID	Inherited
Session membership	Inherited
Time left until alarm clock signal	Inherited
Current working directory	Inherited
Root directory	Inherited
File mode creation mask	Inherited
Process signal mask	Inherited
Pending signals	Inherited

There can be a performance degradation if a process changes identity to a user who does not have permission to be located in the current working directory. The best solution is to *chdir()* after a call to *exec_asuser()*, to a directory known to be permitted for the new identity.

RETURN VALUES

On success, the *exec_asuser* functions don't return. If they fail, they set *errno* and return -1.

ERRORS

The *exec_asuser* functions can set *errno* to any of the following values:

[E2BIG]	The argument list was too long.
[EACCES]	Search permission was denied for a directory in the <i>path</i> or the new process image file wasn't executable.
[EINVAL]	The values for <i>logontype</i> or <i>logonprovider</i> were bad.
[ENAMETOOLONG]	Some component of <i>path</i> was greater than {NAME_MAX}, or the entire pathname exceeded {PATH_MAX}.
[ENOENT]	One or more components of <i>path</i> doesn't exist, or the <i>path</i> or <i>file</i> argument is an empty string.
[ENOEXEC]	For <i>execl_asuser()</i> , <i>execv_asuser()</i> , <i>execle_asuser()</i> , and <i>execve_asuser()</i> , the new process image file has the appropriate access permission, but is not in the proper format.
[ENOMEM]	Not enough memory to load the new process image.

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

[EPERM] The *user-domain-password* combination is invalid.

SEE ALSO

authenticateuser(), *exec()*, *fork()*

NAME

`_exit()` – terminate the calling process

SYNOPSIS

```
#include <unistd.h>

void _exit (int status)
```

DESCRIPTION

The `_exit()` function terminates a process with the following consequences:

- All of the descriptors open in the calling process are closed. This may entail delays, for example, waiting for output to drain; a process in this state may not be killed, as it is already dying.
- If the parent process of the calling process has an outstanding `wait` call or catches the SIGCHLD signal, it is notified of the calling process's termination and the `status` is set as defined by `wait()`.
- The parent process-ID of all of the calling process's existing child processes are set to 1; the initialization process inherits each of these processes.
- If the termination of the process causes any process group to become orphaned (usually because the parents of all members of the group have now exited) and if any member of the orphaned group is stopped, the SIGHUP signal and the SIGCONT signal are sent to all members of the newly-orphaned process group.
- If the process is a controlling process, the SIGHUP signal is sent to the foreground process group of the controlling terminal, and all current access to the controlling terminal is revoked.

Most C programs call the library routine `exit()` rather than `_exit()`, which flushes buffers, closes streams, unlinks temporary files, etc., before calling `_exit()`.

RETURN VALUE

`_exit()` can never return.

SEE ALSO

`fork()`, `wait()`, `exit()`

NAME

`exit()` – perform normal program termination

SYNOPSIS

```
#include <stdlib.h>

void exit (int status)
```

DESCRIPTION

Exit() terminates a process.

Before termination it performs the following functions in the order listed:

1. Call the functions registered with the *atexit()* function, in the reverse order of their registration.
2. Flush all open output streams.
3. Close all open streams.
4. Unlink all files created with the *tmpfile()* function.

RETURN VALUES

The *exit()* function never returns.

SEE ALSO

_exit(), *atexit()*, *tmpfile()*

NAME

exp(), expm1(), log(), log10(), log1p(), pow() – exponential, logarithm, power functions

SYNOPSIS

```
#include <math.h>

double exp (double x)
double expm1 (double x)
double log (double x)
double log10 (double x)
double log1p (double x)
double pow (double x, double y)
```

DESCRIPTION

The *exp()* function computes the exponential value of the given argument *x*.

The *expm1()* function computes the value $\exp(x)-1$ accurately even for tiny argument *x*.

The *log()* function computes the value for the natural logarithm of the argument *x*.

The *log10()* function computes the value for the logarithm of argument *x* to base 10.

The *log1p()* function computes the value of $\log(1+x)$ accurately even for tiny argument *x*.

The *pow()* computes the value of *x* to the exponent *y*.

ERROR (due to Roundoff etc.)

The functions *exp(x)*, *expm1(x)* and *log1p(x)* are accurate to within an *ulp*, and *log10(x)* to within about 2 *ulps*; an *ulp* is one *Unit in the Last Place*. The error in *pow(x,y)* is below about 2 *ulps* when its magnitude is moderate, but increases as *pow(x,y)* approaches the over/underflow thresholds until almost as many bits could be lost as are occupied by the floating-point format's exponent field; that is 8 bits for VAX D and 11 bits for IEEE 754 Double. No such drastic loss has been exposed by testing; the worst errors observed have been below 20 *ulps* for VAX D, 300 *ulps* for IEEE 754 Double. Moderate values of *pow()* are accurate enough that *pow(integer,integer)* is exact until it is bigger than 2^{56} on a VAX, 2^{53} for IEEE 754.

RETURN VALUES

These functions will return the appropriate computation unless an error occurs or an argument is out of range. The functions *exp()*, *expm1()* and *pow()* detect if the computed value will overflow, set the global variable *errno*. The function *pow(x,y)* checks to see if $x < 0$ and *y* is not an integer, in the event this is true, the global variable *errno* is set. The variable *errno* is set to EDOM by *log()* unless $x > 0$, by *log1p()* unless $x > -1$.

ERRORS

The *exp()* function can fail for the following reasons:

- [EDOM] The argument *x* is *NaN*.
- [ERANGE] The result underflows or overflows.

The *log()* and *log10()* functions can fail for the following reasons:

[EDOM] The argument x is negative or NaN .

[ERANGE] The argument x is zero.

The $\log1p()$ function can fail for the following reasons:

[EDOM] The argument x is less than -1.0 or is NaN .

[ERANGE] The argument x is -1.0 .

The $\text{pow}()$ function can fail for the following reasons:

[EDOM] Either x is negative and y is non-integral; x is zero and y is negative; or y is NaN .

[ERANGE] The correct value would cause underflow or overflow.

NOTES

The functions $\exp(x)-1$ and $\log(1+x)$ are called $\text{expm1}()$ and $\text{logp1}()$ in BASIC on the Hewlett-Packard HP-71B and APPLE Macintosh, EXP1 and LN1 in Pascal, exp1 and log1 in C on APPLE Macintoshes, where they have been provided to make sure financial calculations of $((1+x)^n-1)/x$, namely $\text{expm1}(n*\log1p(x))/x$, will be accurate when x is tiny. They also provide accurate inverse hyperbolic functions.

The function $\text{pow}(x)0$ returns $x^{**}0 = 1$ for all x including $x = 0, \infty$ (not found on a VAX), and NaN (the reserved operand on a VAX). Previous implementations of $\text{pow}()$ may have defined $x^{**}0$ to be undefined in some or all of these cases. Here are reasons for returning $x^{**}0 = 1$ always:

1. Any program that already tests whether x is zero (or infinite or NaN) before computing $x^{**}0$ cannot care whether $0^{**}0 = 1$ or not. Any program that depends upon $0^{**}0$ to be invalid is dubious anyway since that expression's meaning and, if invalid, its consequences vary from one computer system to another.
2. Some Algebra texts (e.g. Sigler's) define $x^{**}0 = 1$ for all x , including $x = 0$. This is compatible with the convention that accepts $a[0]$ as the value of polynomial

$$p(x) = a[0]*x^{**}0 + a[1]*x^{**}1 + a[2]*x^{**}2 + \dots + a[n]*x^{**}n$$

at $x = 0$ rather than reject $a[0]*0^{**}0$ as invalid.

3. Analysts will accept $0^{**}0 = 1$ despite that $x^{**}y$ can approach anything or nothing as x and y approach 0 independently. The reason for setting $0^{**}0 = 1$ anyway is this:

If $x(z)$ and $y(z)$ are
any
 functions analytic (expandable
 in power series) in z around $z = 0$, and if there
 $x(0) = y(0) = 0$, then $x(z)^{**}y(z) \rightarrow 1$ as $z \rightarrow 0$.

4. If $0^{**}0 = 1$, then $\infty^{**}0 = 1/0^{**}0 = 1$ too; and then $NaN^{**}0 = 1$ too because $x^{**}0 = 1$ for all finite and infinite x , i.e., independently of x .

SEE ALSO

math, *infnan()*

NAME

`fabs()` – floating-point absolute value function

SYNOPSIS

```
#include <math.h>
```

```
double fabs (double x)
```

DESCRIPTION

The `fabs()` function computes the absolute value of a floating-point number `x`.

RETURN VALUES

The `fabs()` function returns the absolute value of `x`.

SEE ALSO

`abs()`, `ceil()`, `floor()`

NAME

fclose() – close a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (FILE *stream)
```

DESCRIPTION

The *fclose()* function dissociates the named *stream* from its underlying file or set of functions. If the stream was being used for output, any buffered data is written first, using *fflush()*.

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, EOF is returned and the global variable *errno* is set to indicate the error. In either case no further access to the stream is possible.

ERRORS

[EAGAIN]	The O_NONBLOCK flag is set for the file and the process would be delayed in the flush (write) operation.
[EBADF]	The argument <i>stream</i> is not an open stream.
[EFBIG]	Flushing output would result in a file that exceeds the maximum file size.
[EINTR]	A signal interrupted the close operation.
[EIO]	
[ENOSPC]	There is no space on the device to write pending output.
[EPIPE]	<i>stream</i> is actually a pipe (or FIFO) that is not open for reading by any process.

SEE ALSO

close(), *fflush()*, *fopen()*, *setbuf()*

NAME

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*. *Cmd* is one of `F_DUPFD`, `F_GETFD`, `F_SETFD`, `F_GETFL`, `F_SETFL`, or one of the commands for advisory file locking, `F_GETLK`, `F_SETLK`, or `F_SETLKW`.

Values for *cmd* are:

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 *execv()* 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 file descriptor flags of *fd* to the values set in *arg* (an **int**). If the `FD_CLOEXEC` flag is set to 1, the file descriptor is closed upon execution of an *exec()*-family function. If the flag is 0, the descriptor is not closed.

F_GETFL

Get descriptor status flags, as described below. *Arg* is ignored.

F_SETFL

Set descriptor status flags to *arg*.

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 [EAGAIN].

O_APPEND

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

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

following structure:

```

struct flock {
    off_t    l_start;    starting offset
    off_t    l_len;      len = 0 means until end of file
    pid_t    l_pid;      lock owner
    short    l_type;      lock type: read / write, etc.
    short    l_whence;    type of l_start
};

```

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 [EACCES].

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.

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_whence* is SEEK_SET, SEEK_CUR, or SEEK_END to indicate that the relative offset, *l_start* bytes, will be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. If *l_len* is negative, the result is undefined. The *l_pid* field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, the value of *l_whence* is 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_len* is set to zero. If *l_whence* and *l_start* point to the beginning of the file, and *l_len* is zero, the entire file is locked.

There is at most one type of lock set for each byte in the file. Before a successful return from an `F_SETLK` or an `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 `F_SETLK` or an `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.

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

RETURN VALUES

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

<code>F_DUPFD</code>	A new file descriptor.
<code>F_GETFD</code>	Value of flag (only the low-order bit is defined).
<code>F_GETFL</code>	Value of flags.

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

ERRORS

Fcntl() will fail if:

- | | |
|----------|--|
| [EACCES] | The argument <i>arg</i> is <code>F_SETLK</code> , the type of lock (<i>l_type</i>) is a shared lock (<code>F_RDLCK</code>) or exclusive lock (<code>F_WRLCK</code>), 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] | <i>Fd</i> is not a valid open file descriptor.

The argument <i>cmd</i> is <code>F_SETLK</code> or <code>F_SETLKW</code> , the type of lock (<i>l_type</i>) is a shared lock (<code>F_RDLCK</code>), and <i>fd</i> is not a valid file descriptor open for reading.

The argument <i>cmd</i> is <code>F_SETLK</code> or <code>F_SETLKW</code> , the type of lock (<i>l_type</i>) is an exclusive lock (<code>F_WRLCK</code>), and <i>fd</i> is not a valid file descriptor open for writing. |
| [EMFILE] | <i>Cmd</i> is <code>F_DUPFD</code> and the maximum allowed number of file descriptors are currently open. |
| [EINTR] | The argument <i>cmd</i> is <code>F_SETLKW</code> , and the function was interrupted by a signal. |
| [EINVAL] | <i>Cmd</i> is <code>F_DUPFD</code> and <i>arg</i> is negative or greater than the maximum allowable number <code>{OPEN_MAX}</code> .

The argument <i>cmd</i> is <code>F_GETLK</code> , <code>F_SETLK</code> , or <code>F_SETLKW</code> and the data to which <i>arg</i> points is not valid, or <i>fd</i> refers to a file that does not support locking. |
| [EMFILE] | The argument <i>cmd</i> is <code>F_DUPED</code> and the maximum number of file descriptors permitted for the process are already in use, or no file descriptors greater than or equal to <i>arg</i> 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. This error is not possible under INTERIX because there is no system-imposed limit.

SEE ALSO

close(), *exec()*, *open()*

NAME

`fopen()`, `fdopen()`, `freopen()` – stream open functions

SYNOPSIS

```
#include <stdio.h>

FILE * fopen (char *path, char *mode)
FILE * fdopen (int fd, char *mode)
FILE * freopen (char *path, char *mode, FILE *stream)
```

DESCRIPTION

The `fopen()` function opens the file whose name is the string pointed to by *path* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

- “r” Open text file for reading. The stream is positioned at the beginning of the file.
- “r+” Open for reading and writing. The stream is positioned at the beginning of the file.
- “w” Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- “w+” Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- “a” Open for writing. The file is created if it does not exist. The stream is positioned at the end of the file.
- “a+” Open for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file.

The *mode* string can also include the letter “b” either as a third character or as a character between the characters in any of the two-character strings described above.

Any created files will have mode `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH` (0666), as modified by the process’ `umask()` value (see `umask()`).

Reads and writes may be intermixed on read / write streams in any order, and do not require an intermediate seek as in previous versions of *stdio*. This is not portable to other systems, however; ANSI C requires that a file positioning function intervene between output and input, unless an input operation encounters end-of-file.

The `fdopen()` function associates a stream with the existing file descriptor, *fd*. The *mode* of the stream must be compatible with the mode of the file descriptor.

The `freopen()` function opens the file whose name is the string pointed to by *path* and associates the stream pointed to by *stream* with it. The original stream (if it exists) is closed. The *mode* argument is used just as in the `fopen` function. The primary use of the `freopen()` function is to change the file associated with a standard text stream (`stderr`, `stdin`, or `stdout`).

RETURN VALUES

Upon successful completion `fopen()`, `fdopen()` and `freopen()` return a **FILE** pointer. Otherwise, `NULL` is returned and the global variable `errno` is set to indicate the error.

ERRORS

[EINVAL] The *mode* provided to *fopen()*, *fdopen()*, or *freopen()* was invalid.

The *fopen()* function may fail and set *errno* for any of the errors specified for the routines *malloc()* and *open()*. The *fdopen()* function may fail and set *errno* for any of the errors specified for the routines *malloc()* and *fcntl()*. The *freopen()* function may also fail and set *errno* for any of the errors specified for the routines *open()*, *fclose()* and *fflush()*.

SEE ALSO

open(), *fclose()*, *fflush()*, *fopen()*, *fseek()*

NAME

fflush() – flush a stream

SYNOPSIS

```
#include <stdio.h>

int fflush (FILE *stream)
```

DESCRIPTION

The function *fflush()* forces a write of all buffered data for the given output or update *stream* via the stream's underlying write function. The open status of the stream is unaffected.

If the *stream* argument is NULL, *fflush()* flushes *all* open output streams.

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, EOF is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EBADF] *Stream* is not a stream open for writing.

Fflush() can also fail for any of the reasons associated with the *write()* function.

SEE ALSO

fclose(), *fopen()*, *setbuf()*, *write()*

NAME

ffs() – find first bit set in a bit string

SYNOPSIS

```
#include <strings.h>

int ffs (int value)
```

DESCRIPTION

The *ffs()* function finds the first bit set in *value* and returns the index of that bit. Bits are numbered starting from 1, starting at the right-most bit.

RETURN VALUES

Ffs() returns the index to the first bit set; the right-most bit is numbered 1. A return value of 0 means that the argument was zero.

NAME

`fgets()`, `gets()` – get a line from a stream

SYNOPSIS

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

```
char * fgets (char *str, size_t size, FILE *stream)
```

```
char * gets (char *str)
```

DESCRIPTION

The `fgets()` function reads at most one less than the number of characters specified by *size* from the given *stream* and stores them in the string *str*. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. In any case a `\0` character is appended to end the string.

The `gets()` function is equivalent to `fgets()` with an infinite *size* and a *stream* of *stdin*, except that the newline character (if any) is not stored in the string. It is the caller's responsibility to ensure that the input line, if any, is sufficiently short to fit in the string.

Because the `gets()` function is a security violation, this implementation warns the user by writing the string, "warning: this program uses `gets()`, which is unsafe.\r\n" to standard error. This warning can be disabled by setting the environment variable `DISABLE_GETS_WARNING`.

RETURN VALUES

Upon successful completion, `fgets()` and `gets()` return a pointer to the string. If end-of-file or an error occurs before any characters are read, they return `NULL`. The `fgets()` and functions `gets()` do not distinguish between end-of-file and error, and callers must use `feof()` and `ferror()` to determine which occurred.

ERRORS

[EBADF] The given *stream* is not a readable stream.

The function `fgets()` may also fail and set *errno* for any of the errors specified for the routines `fflush()`, `fstat()`, `read()` or `malloc()`.

The function `gets()` may also fail and set *errno* for any of the errors specified for the routine `getchar()`.

NOTES

Since it is usually impossible to ensure that the next input line is less than some arbitrary length, and because overflowing the input buffer is almost invariably a security violation, programs should *never* use `gets()`. The `gets()` function exists to conform to ANSI-C.

SEE ALSO

`feof()`, `ferror()`

NAME

floor() – largest integral value not greater than *x*

SYNOPSIS

```
#include <math.h>
```

```
double floor (double x)
```

DESCRIPTION

The *floor()* function computes the largest integral value not greater than *x*.

RETURN VALUES

The *floor()* function returns the largest integral value expressed as a **double**.

If the result would cause an overflow, *floor()* returns `–HUGE_VAL` and sets *errno*. If *x* is ± 0 , *floor()* returns the value of *x*.

ERRORS

The *floor()* function can fail for the following reasons:

[EDOM] The argument *x* is *NaN*.

[ERANGE] The result would cause an overflow.

SEE ALSO

abs(), *ceil()*, *fabs()*, *ieee()*, *math()*, *rint()*

NAME

fmod() – floating-point remainder function

SYNOPSIS

```
#include <math.h>
```

```
double fmod (double x, double y)
```

DESCRIPTION

The *fmod()* function computes the floating-point remainder of x/y .

RETURN VALUES

The *fmod()* function returns the value $x - i * y$, for some integer i such that, if y is non-zero, the result has the same sign as x and magnitude less than the magnitude of y . If y is zero, whether a domain error occurs or the *fmod()* function returns zero is implementation-defined.

NAME

fnmatch() – match filename or pathname

SYNOPSIS

```
#include <fnmatch.h>
```

```
int fnmatch (const char *pattern, const char *string, int flags)
```

DESCRIPTION

The *fnmatch()* function matches patterns according to the rules used by the shell. It checks the string specified by the *string* argument to see if it matches the pattern specified by the *pattern* argument.

The *flags* argument modifies the interpretation of *pattern* and *string*. The value of *flags* is the bitwise inclusive OR of any of the following constants, which are defined in the include file **<fnmatch.h>**.

FNM_NOESCAPE

Normally, every occurrence of a backslash (\) followed by a character in *pattern* is replaced by that character. This is done to negate any special meaning for the character. If the FNM_NOESCAPE flag is set, a backslash character is treated as an ordinary character.

FNM_PATHNAME

Slash characters in *string* must be explicitly matched by slashes in *pattern*. If this flag is not set, then slashes are treated as regular characters.

FNM_PERIOD

Leading periods in strings match periods in patterns. The definition of “leading” is related to the specification of FNM_PATHNAME. A period is always “leading” if it is the first character in *string*. Additionally, if FNM_PATHNAME is set, a period is “leading” if it immediately follows a slash.

RETURN VALUES

The *fnmatch()* function returns zero if *string* matches the pattern specified by *pattern*, otherwise, it returns the value FNM_NOMATCH.

SEE ALSO

sh

glob(), *regexec()*

BUGS

The pattern *** matches the empty string, even if FNM_PATHNAME is specified.

NAME

fork() – create a new process

SYNOPSIS

```
#include <unistd.h>

pid_t fork (void)
```

DESCRIPTION

Fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an *lseek()* on a descriptor in the child process can affect a subsequent *read()* or *write()* by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child process's values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* are set to zero.
- The child does not inherit any file locks set by the parent.
- Any pending alarms have been cleared for the child process.
- Signals pending for the child process have been initialized to the empty set.

RETURN VALUES

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

ERRORS

Fork() will fail and no child process will be created if:

- | | |
|----------|---|
| [EAGAIN] | The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent. |
| [EAGAIN] | The system-imposed limit on the total number of processes under execution by a single user would be exceeded. |

SEE ALSO

exec(), *wait()*

NAME

`fputs()`, `puts()` – output a line to a stream

SYNOPSIS

```
#include <stdio.h>

int fputs (const char *str, FILE *stream)
int puts (const char *str)
```

DESCRIPTION

The function `fputs()` writes the string pointed to by `str` to the stream pointed to by `stream`.

The function `puts()` writes the string `str` and a terminating newline character, to the stream `stdout`.

RETURN VALUES

The `fputs()` function returns 0 on success and EOF on error; `puts()` returns a nonnegative integer on success and EOF on error.

ERRORS

`fputs()` and `puts()` can fail and set `errno` to the following values:

[EBADF]	The <i>stream</i> is not a writable stream.
[EPIPE]	An attempt is made to write to a pipe that is not open for reading by any process.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
[EFAULT]	Part of the data to be written to the file points outside the process's allocated address space.
[EINVAL]	The pointer associated with <i>stream</i> was negative.
[ENOSPC]	There is no free space remaining on the file system containing the file.
[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.

SEE ALSO

`putc()`, `ferror()`

NAME

`fread()`, `fwrite()` – binary stream input/output

SYNOPSIS

```
#include <stdio.h>

int fread (void *ptr, size_t size, size_t nmemb, FILE *stream)
int fwrite (const void *ptr, size_t size, size_t nmemb,
            FILE *stream)
```

DESCRIPTION

The function `fread()` reads *nmemb* objects, each *size* bytes long, from the stream pointed to by *stream*, storing them at the location given by *ptr*.

The function `fwrite()` writes *nmemb* objects, each *size* bytes long, to the stream pointed to by *stream*, obtaining them from the location given by *ptr*. The functions `fread()` and `fwrite()` advance the file position indicator for the stream by the number of bytes read or written.

RETURN VALUES

`Fread()` and `fwrite()` return the number of objects read or written. If an error occurs, or the end-of-file is reached, the return value is a short object count (or zero).

Because the function `fread()` does not distinguish between end-of-file and error, callers must use `feof()` and `ferror()` to determine which occurred.

The function `fwrite()` returns a value less than *nmemb* only if a write error has occurred.

ERRORS

`Fread()` can set *errno* to the following values:

[EBADF]	<i>Stream</i> is not a valid file or socket descriptor open for reading.
[EFAULT]	<i>Ptr</i> points outside the allocated address space.
[EIO]	An I/O error occurred while reading from the file system.
[EINTR]	Before any data arrived, a read from a slow device was interrupted by the delivery of a signal.
[EINVAL]	The pointer associated with <i>stream</i> was negative.
[EAGAIN]	The file was marked for non-blocking I/O, and no data were ready to be read.

`Fwrite()` can set *errno* to the following values:

[EBADF]	<i>Stream</i> 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.
[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 the data to be written to the file points outside the process's allocated address space.
[EINVAL]	The pointer associated with <i>stream</i> was negative.

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

SEE ALSO

read(), write()

NAME

`free()` – free up memory allocated with `malloc`, `calloc` or `realloc`

SYNOPSIS

```
#include <stdlib.h>

void free (void *ptr)
```

DESCRIPTION

The `free()` function causes the space pointed to by *ptr* to be deallocated, that is, made available for further allocation. If *ptr* is a NULL pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the `calloc()`, `malloc()`, or `realloc()` function, or if the space has been deallocated by a call to `free()` or `realloc()`, problems may result.

RETURN VALUES

The `free()` function returns no value.

SEE ALSO

`calloc()`, `malloc()`, `realloc()`

NAME

frexp() – convert floating-point number to fractional and integral components

SYNOPSIS

```
#include <math.h>
```

```
double frexp (double value, int *exp)
```

DESCRIPTION

The *frexp()* function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the *int* object pointed to by *exp*.

Frexp() is the inverse of *ldexp()*.

RETURN VALUES

The *frexp()* function returns the value *x*, such that *x* is a **double** with magnitude in the interval $[1/2, 1]$ or zero, and *value* equals *x* times 2 raised to the power **exp*. If *value* is zero, both parts of the result are zero.

SEE ALSO

ldexp(), *modf()*

NAME

`user_from_uid()`, `group_from_gid()` – get user or group name from uid or gid

SYNOPSIS

```
#include <stdlib.h>

int pwcache_enable ;

char * user_from_uid(uid_t userid, int d)
char * group_from_gid(gid_t groupid, int d)
```

DESCRIPTION

The `user_from_uid()` returns the user name associated with the argument *userid*. If the global variable `pwcache_enable` has been set to the value 1 (the default), the user name is cached so that multiple calls with the same user ID do not make additional calls to `getpwuid()`. If there is no user associated with *userid*, `user_from_uid()` returns a pointer to a string that represents *userid*. If `pwcache_enable` is set to 0, caching is disabled.

The `group_from_gid()` returns the group name associated with the argument *groupid*. If the global variable `pwcache_enable` has been set to the value 1 (the default), the user name is cached so that multiple calls with the same user ID do not make additional calls to `getgruid()`. If there is no user associated with *groupid*, `group_from_gid()` returns a pointer to a string that represents *groupid*. If `pwcache_enable` is set to 0, caching is disabled.

userid A user ID number.

d This argument is reserved for future use; please pass a 0 here.

RETURN VALUE

`User_from_uid()` and `group_from_gid()` always return a pointer to a valid string. Because they call `getpwuid()` and `getgrgid()`, the `errno` value may be set with any of the errors possible from those functions.

Under some circumstances, the pointer will be to a static buffer which may be overwritten by the next call to this function or to `getpwuid()` or `getgrgid()`.

SEE ALSO

`getgrgid()`, `getpwuid()`

NAME

`fgetpos()`, `fseek()`, `fsetpos()`, `ftell()`, `rewind()` – reposition a stream

SYNOPSIS

```
#include <stdio.h>

int fseek (FILE *stream, long offset, int whence)
long ftell (FILE *stream)
void rewind (FILE *stream)
int fgetpos (FILE *stream, fpos_t *pos)
int fsetpos (FILE *stream, fpos_t *pos)
```

DESCRIPTION

The `fseek()` function sets the file position indicator for the stream pointed to by *stream*. The new position, measured in bytes, is obtained by adding *offset* bytes to the position specified by *whence*. If *whence* is set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the `fseek()` function clears the end-of-file indicator for the stream and undoes any effects of the `ungetc()` function on the same stream.

The `ftell()` function obtains the current value of the file position indicator for the stream pointed to by *stream*.

The `rewind()` function sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to:

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared (see `clearerr()`).

The `fgetpos()` and `fsetpos()` functions are alternate interfaces equivalent to `ftell()` and `fseek()` (with *whence* set to `SEEK_SET`), setting and storing the current value of the file offset into or from the object referenced by *pos*. On some systems an “**fpos_t**” object may be a complex object and these routines may be the only way to portably reposition a text stream.

RETURN VALUES

The `rewind()` function returns no value. Upon successful completion, `fgetpos()`, `fseek()`, `fsetpos()` return 0, and `ftell()` returns the current offset. Otherwise, `-1` is returned and the global variable `errno` is set to indicate the error.

ERRORS

[EBADF]	The <i>stream</i> specified is not a seekable stream.
[EINVAL]	The <i>whence</i> argument to <code>fseek()</code> was not <code>SEEK_SET</code> , <code>SEEK_END</code> , or <code>SEEK_CUR</code> .

The function `fgetpos()`, `fseek()`, `fsetpos()`, and `ftell()` may also fail and set `errno` for any of the errors specified for the routines `fflush()`, `fstat()`, `lseek()`, and `malloc()`.

SEE ALSO

`lseek()`

NAME

`fsync()` – synchronize disk and in-memory versions of a file

SYNOPSIS

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

DESCRIPTION

The `fsync()` call flushes the buffer for the specified file descriptor causes all modified data and attributes of *fd* to be moved to a permanent storage device. It flushes to disk the buffers for the specified file descriptor.

Use `fsync()` in programs that require a file to be in a known state, for example, in building a simple transaction facility.

RETURN VALUES

The `fsync()` call returns 0 for success. On an error, it returns -1 and sets *errno* to indicate the error.

ERRORS

The `fsync()` call fails for the following reasons:

- | | |
|----------|--|
| [EBADF] | The file descriptor <i>fd</i> was invalid. |
| [EINTR] | The function was interrupted by a signal. |
| [EINVAL] | The file referred to by <i>fd</i> doesn't support the operation. |
| [EIO] | An I/O error occurred. |

NAME

ftruncate() – truncate a file to a specified length

SYNOPSIS

```
#include <unistd.h>

int ftruncate (int fd, off_t length)
```

DESCRIPTION

Ftruncate() truncates the file or referenced by *fd* to at most *length* bytes in size. If the file previously was larger than this size, the extra data is lost. The file described by *fd* must have been opened for writing.

RETURN VALUES

Ftruncate() returns 0 on success. On failure, it returns -1 and sets the global variable *errno*.

ERRORS

Ftruncate() succeeds unless:

- | | |
|----------|---|
| [EBADF] | The <i>fd</i> is not a valid descriptor. |
| [EINVAL] | The file descriptor <i>fd</i> is not open for writing, or the file descriptor <i>fd</i> points to a socket, not a file. |
| [ESPIPE] | The <i>fd</i> refers to a FIFO or a pipe. |

SEE ALSO

open()

NAME

fts_open(), fts_read(), fts_children(), fts_set(), fts_close() – traverse a file hierarchy

SYNOPSIS

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

FTS * fts_open (char *const *path_argv, int options, int
               *compar(const FTSENT **, const FTSENT **))
FTSENT * fts_read (FTS *ftsp)

FTSENT * fts_children (FTS *ftsp, int options)

int fts_set (FTS ftsp, FTSENT *f, int options)

int fts_close (FTS *ftsp)
```

DESCRIPTION

The *fts()* functions are provided for traversing file hierarchies. A simple overview is that the *fts_open()* function returns a “handle” on a file hierarchy, which is then supplied to the other *fts()* functions. The function *fts_read()* returns a pointer to a structure describing one of the files in the file hierarchy. The function *fts_children()* returns a pointer to a linked list of structures, each of which describes one of the files contained in a directory in the hierarchy. In general, directories are visited two distinguishable times; in pre-order (before any of their descendants are visited) and in post-order (after all of their descendants have been visited). Files are visited once. It is possible to walk the hierarchy “logically” or physically, order the walk of the hierarchy or prune and/or re-visit portions of the hierarchy.

Two structures are defined (and **typedef**'d) in the include file **<fts.h>**. The first is **FTS**, the structure that represents the file hierarchy itself. The second is **FTSENT**, the structure that represents a file in the file hierarchy. Normally, an **FTSENT** structure is returned for every file in the file hierarchy. In this manual page, “file” and “**FTSENT structure**” are generally interchangeable. The **FTSENT** structure contains at least the following fields, which are described in greater detail below:

```
typedef struct _ftsent {
    u_short    fts_info;        flags for FTSENT structure
    char *     fts_accpath;     access path
    char *     fts_path;       root path
    short      fts_pathlen;     strlen(fts_path)
    char *     fts_name;       file name
    short      fts_namelen;     strlen(fts_name)
    short      fts_level;      depth (-1 to N)
    int        fts_errno;      file errno
    long       fts_number;     local numeric value
}
```

```

        void *      fts_pointer;    local address value
        struct ftsent * fts_parent;  parent directory
        struct ftsent * fts_link;    next file structure
        struct ftsent * fts_cycle;   cycle structure
        struct stat *   fts_statp;   stat() information
    } FTSENT;

```

These fields are defined as follows:

fts_info One of the following flags describing the returned **FTSENT** structure and the file it represents. With the exception of directories without errors (FTS_D), all of these entries are terminal, that is, they will not be revisited, nor will any of their descendants be visited.

FTS_D A directory being visited in pre-order.

FTS_DC A directory that causes a cycle in the tree. (The *fts_cycle* field of the **FTSENT** structure will be filled in as well.)

FTS_DEFAULT Any **FTSENT** structure that represents a file type not explicitly described by one of the other *fts_info* values.

FTS_DNR A directory which cannot be read. This is an error return, and the *fts_errno* field will be set to indicate what caused the error.

FTS_DOT A file named . or .. which was not specified as a file name to *fts_open()* (see FTS_SEEDOT).

FTS_DP A directory being visited in post-order. The contents of the **FTSENT** structure will be unchanged from when it was returned in pre-order, i.e. with the *fts_info* field set to FTS_D.

FTS_ERR This is an error return, and the *fts_errno* field will be set to indicate what caused the error.

FTS_F A regular file.

FTS_NS A file for which no *stat()* information was available. The contents of the *fts_statp* field are undefined. This is an error return, and the *fts_errno* field will be set to indicate what caused the error.

FTS_NSOK A file for which no *stat()* information was requested. The contents of the *fts_statp* field are undefined.

FTS_SL A symbolic link.

FTS_SLNONE A symbolic link with a non-existent target. The contents of the *fts_statp* field reference the file characteristic information for the symbolic link itself.

fts_accpath A path for accessing the file from the current directory.

fts_path The path for the file relative to the root of the traversal. This path contains the path specified to *fts_open()* as a prefix.

fts_pathlen

	The length of the string referenced by <i>fts_path</i> .
<i>fts_name</i>	The name of the file.
<i>fts_namelen</i>	The length of the string referenced by <i>fts_name</i> .
<i>fts_level</i>	The depth of the traversal, numbered from -1 to N, where this file was found. The FTSENT structure representing the parent of the starting point (or root) of the traversal is numbered -1, and the FTSENT structure for the root itself is numbered 0.
<i>fts_errno</i>	Upon return of a FTSENT structure from the <i>fts_children()</i> or <i>fts_read()</i> functions, with its <i>fts_info</i> field set to FTS_DNR, FTS_ERR or FTS_NS, the <i>fts_errno</i> field contains the value of the external variable <i>errno</i> specifying the cause of the error. Otherwise, the contents of the <i>fts_errno</i> field are undefined.
<i>fts_number</i>	This field is provided for the use of the application program and is not modified by the <i>fts()</i> functions. It is initialized to 0.
<i>fts_pointer</i>	This field is provided for the use of the application program and is not modified by the <i>fts()</i> functions. It is initialized to NULL.
<i>fts_parent</i>	A pointer to the FTSENT structure referencing the file in the hierarchy immediately above the current file, i.e. the directory of which this file is a member. A parent structure for the initial entry point is provided as well, however, only the <i>fts_level</i> , <i>fts_number</i> and <i>fts_pointer</i> fields are guaranteed to be initialized.
<i>fts_link</i>	Upon return from the <i>fts_children()</i> function, the <i>fts_link</i> field points to the next structure in the NULL-terminated linked list of directory members. Otherwise, the contents of the <i>fts_link</i> field are undefined.
<i>fts_cycle</i>	If a directory causes a cycle in the hierarchy (see FTS_DC), either because of a hard link between two directories, or a symbolic link pointing to a directory, the <i>fts_cycle</i> field of the structure will point to the FTSENT structure in the hierarchy that references the same file as the current FTSENT structure. Otherwise, the contents of the <i>fts_cycle</i> field are undefined.
<i>fts_statp</i>	A pointer to <i>stat()</i> information for the file.

A single buffer is used for all of the paths of all of the files in the file hierarchy. Therefore, the *fts_path* and *fts_accpath* fields are guaranteed to be NULL-terminated *only* for the file most recently returned by *fts_read()*. To use these fields to reference any files represented by other **FTSENT** structures will require that the path buffer be modified using the information contained in that **FTSENT** structure's *fts_pathlen* field. Any such modifications should be undone before further calls to *fts_read()* are attempted. The *fts_name* field is always NULL -terminated.

FTS_OPEN

The *fts_open()* function takes a pointer to an array of character pointers naming one or more paths which make up a logical file hierarchy to be traversed. The array must be terminated by a NULL pointer.

There are a number of options, at least one of which (either FTS_LOGICAL or FTS_PHYSICAL) must be specified. The options are selected by OR'ing the following values:

FTS_COMFOLLOW

This option causes any symbolic link specified as a root path to be followed immediately whether or not FTS_LOGICAL is also specified.

FTS_LOGICAL

This option causes the *fts()* routines to return **FTSENT** structures for the targets of symbolic links instead of the symbolic links themselves. If this option is set, the only symbolic links for which **FTSENT** structures are returned to the application are those referencing non-existent files. Either FTS_LOGICAL or FTS_PHYSICAL *must* be provided to the *fts_open()* function.

FTS_NOCHDIR

As a performance optimization, the *fts()* functions change directories as they walk the file hierarchy. This has the side-effect that an application cannot rely on being in any particular directory during the traversal. The FTS_NOCHDIR option turns off this optimization, and the *fts()* functions will not change the current directory. Note that applications should not themselves change their current directory and try to access files unless FTS_NOCHDIR is specified and absolute pathnames were provided as arguments to *fts_open()*.

FTS_NOSTAT

By default, returned **FTSENT** structures reference file characteristic information (the *statp* field) for each file visited. This option relaxes that requirement as a performance optimization, allowing the *fts()* functions to set the *fts_info* field to FTS_NSOK and leave the contents of the *statp* field undefined.

FTS_PHYSICAL

This option causes the *fts()* routines to return **FTSENT** structures for symbolic links themselves instead of the target files they point to. If this option is set, **FTSENT** structures for all symbolic links in the hierarchy are returned to the application. Either FTS_LOGICAL or FTS_PHYSICAL *must* be provided to the *fts_open()* function.

FTS_SEEDOT

By default, unless they are specified as path arguments to *fts_open()*, any files named *.* or *..* encountered in the file hierarchy are ignored. This option causes the *fts()* routines to return **FTSENT** structures for them.

FTS_XDEV

This option prevents *fts()* from descending into directories that have a different device number than the file from which the descent began.

The argument *compar()* specifies a user-defined function which may be used to order the traversal of the hierarchy. It takes two pointers to pointers to **FTSENT** structures as arguments and should return a negative value, zero, or a positive value to indicate if the file referenced by its first argument comes before, in any order with respect to, or after, the file referenced by its second argument. The *fts_accpath*, *fts_path* and *fts_pathlen* fields of the **FTSENT** structures may *never* be used in this comparison. If the *fts_info* field is set to FTS_NS or FTS_NSOK, the *fts_statp* field may not either. If the *compar()* argument is NULL, the directory traversal order is in the order listed in *path_argv* for the root paths, and in the order listed in the directory for everything else.

FTS_READ

The *fts_read()* function returns a pointer to an **FTSENT** structure describing a file in the

hierarchy. Directories (that are readable and do not cause cycles) are visited at least twice, once in pre-order and once in post-order. All other files are visited at least once. (Hard links between directories that do not cause cycles or symbolic links to symbolic links may cause files to be visited more than once, or directories more than twice.)

If all the members of the hierarchy have been returned, *fts_read()* returns NULL and sets the external variable *errno* to 0. If an error unrelated to a file in the hierarchy occurs, *fts_read()* returns NULL and sets *errno* appropriately. If an error related to a returned file occurs, a pointer to an **FTSENT** structure is returned, and *errno* may or may not have been set (see *fts_info*).

The **FTSENT** structures returned by *fts_read()* may be overwritten after a call to *fts_close()* on the same file hierarchy stream, or, after a call to *fts_read()* on the same file hierarchy stream unless they represent a file of type directory, in which case they will not be overwritten until after a call to *fts_read()* after the **FTSENT** structure has been returned by the function *fts_read()* in post-order.

FTS_CHILDREN

The *fts_children()* function returns a pointer to an **FTSENT** structure describing the first entry in a NULL-terminated linked list of the files in the directory represented by the **FTSENT** structure most recently returned by *fts_read()*. The list is linked through the *fts_link* field of the **FTSENT** structure, and is ordered by the user-specified comparison function, if any. Repeated calls to *fts_children()* will recreate this linked list.

As a special case, if *fts_read()* has not yet been called for a hierarchy, *fts_children()* will return a pointer to the files in the logical directory specified to *fts_open()*, i.e. the arguments specified to *fts_open()*. Otherwise, if the **FTSENT** structure most recently returned by *fts_read()* is not a directory being visited in pre-order, or the directory does not contain any files, *fts_children()* returns NULL and sets *errno* to zero. If an error occurs, *fts_children()* returns NULL and sets *errno* appropriately.

The **FTSENT** structures returned by *fts_children()* may be overwritten after a call to *fts_children()*, *fts_close()* or *fts_read()* on the same file hierarchy stream.

Option may be set to the following value:

FTS_NAMEONLY

Only the names of the files are needed. The contents of all the fields in the returned linked list of structures are undefined with the exception of the *fts_name* and *fts_namelen* fields.

FTS_SET

The function *fts_set()* allows the user application to determine further processing for the file *f* of the stream *ftsp*. The *fts_set()* function returns 0 on success, and -1 if an error occurs. *Option* must be set to one of the following values:

FTS_AGAIN

Re-visit the file; any file type may be re-visited. The next call to *fts_read()* will return the referenced file. The *fts_stat* and *fts_info* fields of the structure will be reinitialized at that time, but no other fields will have been changed. This option is meaningful only for the most recently returned file from *fts_read()*. Normal use is for post-order directory visits, where it causes the directory to be re-visited (in both pre and post-order) as well as all of its descendants.

FTS_FOLLOW

The referenced file must be a symbolic link. If the referenced file is the one

most recently returned by *fts_read()*, the next call to *fts_read()* returns the file with the *fts_info* and *fts_statp* fields reinitialized to reflect the target of the symbolic link instead of the symbolic link itself. If the file is one of those most recently returned by *fts_children()*, the *fts_info* and *fts_statp* fields of the structure, when returned by *fts_read()*, will reflect the target of the symbolic link instead of the symbolic link itself. In either case, if the target of the symbolic link does not exist the fields of the returned structure will be unchanged and the *fts_info* field will be set to *FTS_SLNONE*.

If the target of the link is a directory, the pre-order return, followed by the return of all of its descendants, followed by a post-order return, is done.

FTS_SKIP

No descendants of this file are visited. The file may be one of those most recently returned by either *fts_children()* or *fts_read()*.

FTS_CLOSE

The *fts_close()* function closes a file hierarchy stream *ftsp* and restores the current directory to the directory from which *fts_open()* was called to open *ftsp*. The *fts_close()* function returns 0 on success, and -1 if an error occurs.

ERRORS

The function *fts_open()* may fail and set *errno* for any of the errors specified for the library functions *open()* and *malloc()*.

The function *fts_close()* may fail and set *errno* for any of the errors specified for the library functions *chdir()* and *close()*.

The functions *fts_read()* and *fts_children()* may fail and set *errno* for any of the errors specified for the library functions *chdir()*, *malloc()*, *opendir()*, *readdir()* and *stat()*.

In addition, *fts_children()*, *fts_open()* and *fts_set()* may fail and set *errno* as follows:

[EINVAL] The options were invalid.

SEE ALSO

find

chdir(), *stat()*, *qsort()*

NAME

ftw() – walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw (const char *path, int (*fn)(const char *,
    const struct stat *ptr, int flag), int ndirs);
```

DESCRIPTION

The *ftw()* call recursively descends the directory hierarchy rooted in *path*, calling the function pointed to by *fn* for each object. A null or empty string for *path* is an error. For each call, the function pointed to by *fn* is passed a pointer to the name of the object (a null-terminated character string), a pointer to a **stat** structure that contains information about the object, and an integer *flag* which defines the type of the object. Possible values of *flag* are:

FTW_D The object is a directory.

FTW_DNR

The object is a directory that cannot be read. In this case, children of the directory are not read.

FTW_F The object is a file.

FTW_NS

The object was not successfully *stat()*'ed. In this case, the **stat** function will contain undefined values. For example, a file with read but not execute/search permission can cause this value.

FTW_SL

The object is a symbolic link. Since symbolic links are not supported under INTERIX, this is unlikely to occur.

The argument *ndirs* gives the maximum number of directory streams or file descriptors available for *ftw()*. The value of *ndirs* should be between 1 and {OPEN_MAX}. Since the *ftw()* function uses at most one file descriptor for each level in the tree, this argument effectively limits the depth of a walk. A value of zero allows limitless recursion; a negative value is an error.

When *ftw()* returns, it closes any directory streams and file descriptors it uses, not counting any opened by the function pointed to by *fn*.

The *ftw()* function visits a directory before visiting any of its children.

RETURN VALUES

The *ftw()* function returns 0 when the tree is exhausted. If the function pointed to by *fn* returns a non-zero value or an error (other than [EACCES]), the *ftw()* function stops and returns that value. If *ftw()* detects an error, it returns -1 and sets *errno*.

ERROR VALUES

The *ftw()* function may fail for the following reasons:

[EACCES]	The process does not have permission to read or search a component of <i>path</i> .
[EINVAL]	The <i>ndirs</i> argument is invalid.
[ELOOP]	There were too many symbolic links. Since symbolic links are not supported on INTERIX, this error should not occur.
[ENAMETOOLONG]	

	The <i>path</i> is more than {PATH_MAX} characters long, or a path component is more than {NAME_MAX} characters long.
[ENOENT]	Either a component of <i>path</i> doesn't exist, or <i>path</i> is an empty string.
[ENOMEM]	Insufficient memory.
[ENOTDIR]	A component of <i>path</i> is not a directory.

SEE ALSO

longjmp(), *malloc()*, *nftw()*, *opendir()*, *siglongjmp()*, *stat()*

NAME

`fgetc()`, `getc()`, `getchar()` – get next character or word from input stream

SYNOPSIS

```
#include <stdio.h>

int fgetc (FILE *stream)
int getc (FILE *stream)
int getchar(void)
```

DESCRIPTION

The `fgetc()` function obtains the next input character (if present) from the stream pointed at by *stream*, or the next character pushed back on the stream via `ungetc`.

The `getc()` function acts essentially identically to `fgetc()`, but is a macro that expands in-line.

The `getchar()` function is equivalent to:

```
getc(stdin)
```

Once end-of-file is reached, the end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return EOF until the condition is cleared with `clearerr()`.

RETURN VALUES

If successful, these routines return the next requested object from the *stream*. If the stream is at end-of-file or a read error occurs, the routines return EOF. Since EOF is a valid integer value, you must use `feof()` and `ferror()` to distinguish between end-of-file and error. If an error occurs, the global variable *errno* is set to indicate the error.

ERRORS

These functions can set *errno* to the following values:

[EBADF]	<i>Stream</i> is not a valid file descriptor open for reading.
[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 <i>stream</i> was negative.
[EAGAIN]	The file was marked for non-blocking I/O, and no data were ready to be read.

SEE ALSO

`ferror()`, `fread()`, `fopen()`, `putc()`, `ungetc()`

NAME

getcwd() – get working directory pathname

SYNOPSIS

```
#include <unistd.h>
```

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

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*.

This routine has traditionally been used by programs to save the name of a working directory for the purpose of returning to it.

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.

If an error occurs, the contents of *buf* are not changed.

ERRORS

The *getcwd()* function will fail if:

[EFAULT]	The <i>buf</i> argument is NULL.
[EINVAL]	The <i>size</i> argument is zero.
[ENOENT]	A component of the pathname no longer exists.
[ENOMEM]	Insufficient memory is available.
[ERANGE]	The <i>size</i> argument is greater than zero but smaller than the length of the pathname plus 1.

SEE ALSO

chdir()

NAME

getenv(), putenv() – environment variable functions

SYNOPSIS

```
#include <stdlib.h>

char * getenv (const char *name)
int  putenv (const char *string)
```

DESCRIPTION

These functions set and fetch environment variables from the host. For compatibility with differing environment conventions, the given arguments *name* and *value* may be appended and prepended, respectively, with an equal sign =.

The *getenv()* function obtains the current value of the environment variable, *name*. If the variable *name* is not in the current environment, a null pointer is returned.

The *putenv()* function inserts or resets the environment variable *name* in the current environment list. If the variable *name* does not exist in the list, it is inserted with the given *value*. If it does exist, its value is replaced.

RETURN VALUES

Putenv() returns zero if successful; otherwise the global variable *errno* is set to indicate the error and a -1 is returned.

ERRORS

[ENOMEM] *Putenv()* was unable to allocate memory for the environment.

SEE ALSO

sh

exec()

NAME

getgid(), getegid() – get group process identification

SYNOPSIS

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

gid_t getgid (void)
gid_t getegid (void)
```

DESCRIPTION

The *getgid()* function returns the real group ID of the calling process, *getegid()* returns the effective group ID of the calling process.

The real group ID is the group of the user who invoked the program. The real group ID is specified at login time.

The effective group ID is the group of the process, if it has the set-group-ID mode set. Since the effective group ID gives the process additional permissions during execution, *getgid()* is used to determine the real-user-id of the calling process.

ERRORS

The *getgid()* and *getegid()* functions are always successful, and no return value is reserved to indicate an error.

SEE ALSO

getuid(), *setgid()*

NAME

getgrnam(), getgrgid() – user account database operations

SYNOPSIS

```
#include <grp.h>

struct group * getgrnam (const char *groupname)
struct group * getgrgid (gid_t gid)
```

DESCRIPTION

These functions operate on the user account database file. They return information in a **struct group** structure, defined in the include file **<grp.h>**:

```
struct group {
    char *    gr_name;    group name (null-terminated string)
    gid_t    gr_gid;     group ID
    char **   gr_mem;     group members (null-terminated array of pointers)
};
```

The functions *getgrnam()* and *getgrgid()* search the group database for the given group name pointed to by *groupname* or the group ID pointed to by *gid*.

The **gr_name** member is a null-terminated string. The **gr_mem** member is a null-terminated array of pointers to null-terminated strings. These strings identify all the user and group members in this group.

Refer to the *getpwnam()* reference page for a description of the domain scheme in Windows NT networking.

The return value points to an area that is reused on subsequent calls. The information must be copied elsewhere before the next call to these functions. The names returned in *gr_name* and *gr_mem* are in the form “domain+name”.

RETURN VALUES

The functions *getgrnam()* and *getgrgid()* return a pointer to the group entry if successful.

On failure, the functions return a null pointer and set the value of *errno*.

ERRORS

The *getgrgid()* and *getgrnam()* calls may fail for the following reasons:

- | | |
|----------|--|
| [EIO] | An I/O error occurred. |
| [EINTR] | A signal was caught during the call. |
| [EMFILE] | The calling process already had {OPEN_MAX} file descriptors open. |
| [ENFILE] | The system already had the maximum allowable number of files open. |

SEE ALSO

getgroups(), *getlogin()*, *getpwnam()*, *getpwuid()*

NAME

getgroups() – retrieve supplementary group ID list

SYNOPSIS

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

int getgroups (int gidsetsize, gid_t grouplist[])
```

DESCRIPTION

The *getgroups()* function reads through the group database and fills the *grouplist* array with the supplementary group IDs of the calling process.

The argument *gidsetsize* specifies the number of elements in the supplied array.

To determine the number of supplemental group IDs associated with the calling process, set *gidsetsize* to 0.

On Windows NT, groups and users share the same namespace in the user accounts database.

RETURN VALUES

The *getgroups()* function returns the number of supplementary group IDs filled in the *grouplist* array.

On error, it returns -1. The group array will be filled with as many groups as will fit.

ERRORS

The *getgroups()* function fails because:

[EINVAL]	The <i>gidsetsize</i> argument isn't equal to 0 and is less than the number of supplementary group IDs.
----------	---

SEE ALSO

getgrnam(), *getpwnam()*

NAME

gethostbyname(), gethostbyaddr(), sethostent(), endhostent(), perror() – get network host entry (and `h_errno` variable)

SYNOPSIS

```
#include <netdb.h>
extern int h_errno;
struct hostent *gethostbyname (char *name)
struct hostent *gethostbyaddr (char *addr, int len, int type)
sethostent (int stayopen)
endhostent (void)
perror (char *string)
```

DESCRIPTION

The *gethostbyname()* 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 the information obtained from the local machine's Winsock DLL configuration (however your Windows NT networking system is configured):

```
struct hostent {
    char *    h_name;           official name of host
    char **   h_aliases;       alias list
    int       h_addrtype;      host address type
    int       h_length;        length of address
    char **   h_addr_list;     list of addresses from name server
};
```

For backwards compatibility, the address is also defined as:

```
#define      h_addr    h_addr_list[0]
```

The members of this structure are:

h_name Official name of the host.

h_aliases

A zero terminated array of alternate names for the host.

h_addrtype

The type of address being returned; currently always AF_INET.

h_length

The length, in bytes, of the address.

h_addr_list

A zero 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.

The *endhostent()* and *sethostent()* functions are no-ops. They are made available for ease of portability.

DIAGNOSTICS

Error return status from *gethostbyname()* 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 *perror()* can be used to

print an error message describing the failure. If its argument *string* is non-NULL, *string* 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.

NOTES

The sockets mechanism in INTERIX is built around the Microsoft Winsock DLL. Configuring the network information for sockets depends upon your Winsock configuration; see the appropriate system documentation.

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Currently the functions only understand the Internet address format.

SEE ALSO

hosts, *hostname*

NAME

gethostname() – get name of current host

SYNOPSIS

```
#include <unistd.h>
```

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

DESCRIPTION

Gethostname() returns the standard host name for the current processor. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

RETURN VALUES

If the call succeeds, it returns 0. If the call fails, it returns –1 and sets *errno* to indicate the error.

ERRORS

The following errors may be returned by these calls:

[EFAULT] The *name* or *namelen* parameter gave an invalid address.

BUGS

Host names are limited to {MAXHOSTNAMELEN} (from <limits.h>) characters.

NAME

`_getInstallPath()`, `_prefixInstallPath()` – find root of INTERIX install path

SYNOPSIS

```
#include <opennt/opennt.h>

char * _getInstallPath(void)

char * _prefixInstallPath( const char *name, char *buf, int len )
```

DESCRIPTION

These functions retrieve the pathname for the directory where the INTERIX was installed. If you installed the INTERIX Commands and Utilities in the default location (**//C/Interix/**), then the pathname is **//C/Interix/**.

`_getInstallPath()` returns the pathname itself. It looks in the registry for the entry `SSI_REG_INSTALLROOT` and stores the value in a static buffer. It returns a pointer to that buffer. The pathname always ends in `/`.

The `_prefixInstallPath()` call prefixes the installed-root directory onto another path (specified by *name*) and returns a pointer to the concatenated name. If *buf* is NULL, then the concatenated name is stored in a static buffer; if *buf* is non-NULL, the concatenated name is stored there. If you specify *buf*, you must also specify a non-zero value for *len*, the number of bytes that *buf* will hold.

The `_prefixInstallPath()` call is intended to be used for porting software that relies on hard-coded pathnames. For example, the `rcsdiff` utility traditionally execs the **/bin/diff** program. Since INTERIX product can be installed anywhere on the system, **/bin/diff** may not exist.

Because this is a non-standard API, you must **#define** `_ALL_SOURCE` to use it.

RETURN CODES

The `_getInstallPath()` function returns a pointer to a static buffer, or NULL on failure. It can fail for one of these reasons:

- Insufficient memory,
- a component of the installed path exceeds `{NAME_MAX}`,
- or if the entire installed path is longer than `{PATH_MAX}`.

The `_prefixInstallPath()` function returns a pointer to the concatenated path, or NULL for one of the following errors:

- `_getInstallPath()` failed
- *len* was not specified and *buf* was not NULL;
- the concatenated pathname was longer than *len*.

EXAMPLE

Many traditional programs define paths as static strings. For example:

```
#define PAGER "/bin/more"
```

```
...
```

```
printf("Using %s as the pager\n", PAGER);
```

A trivial conversion which ignores the possibility of failure is:

```
#ifndef __OPENNT
```

```
# define PAGER "/bin/more"
```

```
#else
```

```
# define PAGER _prefixInstallPath("/bin/more", NULL, 0)
```

```
#endif /* __OPENNT */
```

```
...
```

```
printf("Using %s as the pager\n", PAGER);
```

SEE ALSO

confstr()

NAME

getlogin() – get login name

SYNOPSIS

```
#include <unistd.h>

char * getlogin(void)
```

DESCRIPTION

The *getlogin()* routine returns the login name of the user associated with the current process. The name is normally associated with a login shell at the time a session is created, and is inherited by all processes descended from the login shell.

Note that the login name is returned in the format

domain+logname

While all INTERIX APIs recognize the + character as a separator in a login name, Win32 programs may not. If you will be passing the returned string to a Win32 program, you may need to change the + character to a backslash (\).

RETURN

If a call to *getlogin()* succeeds, it returns a pointer to a null-terminated string in a static buffer. If the name has not been set, it returns NULL.

SEE ALSO

getpwnam(), *getpwuid()*

NAME

getopt() – get option character from command line argument list

SYNOPSIS

```
#include <unistd.h>
extern char *optarg;
extern int optind;
extern int optopt;
extern int opterr;
extern int optreset;

int getopt (int argc, char * const *argv, const char *optstring)
```

DESCRIPTION

The *getopt()* function incrementally parses a command line argument list *argv* and returns the next *known* option character. An option character is *known* if it has been specified in the string of accepted option characters, *optstring*.

The option string *optstring* may contain the following elements: individual characters, and characters followed by a colon to indicate an option argument is to follow. For example, an option string *x* recognizes an option “*-x*”, and an option string *x:* recognizes an option and argument “*-x argument*”. It does not matter to *getopt()* if a following argument has leading white space.

On return from *getopt()*, *optarg* points to an option argument, if it is anticipated, and the variable *optind* contains the index to the next *argv* argument for a subsequent call to *getopt()*. The variable *optopt* saves the last *known* option character returned by *getopt()*.

The variables *opterr* and *optind* are both initialized to 1. The *optind* variable may be set to another value before a set of calls to *getopt()* in order to skip over more or less *argv* entries.

In order to use *getopt()* to evaluate multiple sets of arguments, or to evaluate a single set of arguments multiple times, the variable *optreset* must be set to 1 before the second and each additional set of calls to *getopt()*, and the variable *optind* must be reinitialized.

The double dash argument (*--*) stops option processing and is skipped, so the argument following *--* is considered the first positional argument.

A single dash argument (*-*) stops option processing but isn’t skipped; *optind* is set so that *-* is the first positional argument.

RETURN VALUES

The *getopt()* function returns the next option character specified on the command line, and handles the following special cases:

It returns a colon (:) if there is a missing argument and the first character of *optstring* was a colon.

It returns a question mark (?) if it encounters an option character not in the *optstring* or there is a missing argument and the first character of *optstring* was not a colon.

Otherwise, *getopt()* returns -1 when all command line options have been parsed.

The interpretation of options in the argument list may be cancelled by the option *--* (double dash) which causes *getopt()* to signal the end of argument processing and return an EOF. When all options have been processed (i.e., up to the first non-option

argument), *getopt()* returns EOF.

DIAGNOSTICS

If the *getopt()* function encounters a character not found in the string *optarg* or detects a missing option argument it writes an error message and returns ? to the stderr. Setting *opterr* to a zero will disable these error messages. If *optstring* has a leading : then a missing option argument causes a : to be returned in addition to suppressing any error messages.

Option arguments are allowed to begin with “-”; this is reasonable but reduces the amount of error checking possible.

EXAMPLE

```
extern char *optarg;
extern int optind;
int bflag, ch, fd;
bflag = 0;
while ((ch = getopt(argc, argv, "bf:")) != EOF)
    switch(ch) {
        case 'b':
            bflag = 1;
            break;
        case 'f':
            if ((fd = open(optarg, O_RDONLY, 0)) < 0) {
                (void)fprintf(stderr,
                    "myname: %s: %s\n", optarg, strerror(errno));
                exit(1);
            }
            break;
        case '?':
        default:
            usage();
    }
argc -= optind;
argv += optind;
```

BUGS

A single dash “-” may be specified as an character in *optstring*, however it should *never* have an argument associated with it. This allows *getopt()* to be used with programs that expect “-” as an option flag. This practice is wrong, and should not be used in any current development. It is provided for backward compatibility *only*. By default, a single dash causes *getopt()* to return EOF. This is, we believe, compatible with System V.

It is also possible to handle digits as option letters. This allows *getopt()* to be used with programs that expect a number (“-3”) as an option. This practice is wrong, and should not be used in any current development. It is provided for backward compatibility *only*. The following code fragment works in most cases.

```
int length;
char *p;
while ((c = getopt(argc, argv, "0123456789")) != EOF)
    switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            p = argv[optind - 1];
            if (p[0] == '-' && p[1] == ch && !p[2])
                length = atoi(++p);
            else
                length = atoi(argv[optind] + 1);
            break;
    }
}
```


NAME

getpeername() – get name of connected peer

SYNOPSIS

```
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

If the call succeeds, it returns 0.

If the call fails, it returns -1 and sets *errno* to indicate the error.

ERRORS

The call succeeds unless:

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

SEE ALSO

accept(), *bind()*, *socket()*, *getsockname()*

NAME

getpgrp() – get process group

SYNOPSIS

```
#include <unistd.h>
```

```
pid_t getpgrp (void)
```

DESCRIPTION

The *getpgrp()* function returns the process group ID of the current process.

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

SEE ALSO

INTERIX Software Developer's Kit (Release 2.2).Xr 2 setsid ()

NAME

getpid(), getppid() – get parent or calling process identification

SYNOPSIS

```
#include <unistd.h>

pid_t getpid (void)
pid_t getppid (void)
```

DESCRIPTION

Getpid() returns the process ID of the calling process. The ID is guaranteed to be unique and is useful for constructing temporary file names.

Getppid() returns the process ID of the parent of the calling process.

ERRORS

The *getpid()* and *getppid()* functions are always successful, and no return value is reserved to indicate an error.

NAME

getprotobynumber(), getprotobyname(), setprotoent(), endprotoent() – get protocol entry

SYNOPSIS

```
#include <netdb.h>

struct protoent * getprotobyname (char *name)
struct protoent * getprotobynumber (int proto)
void setprotoent (int stayopen)
void endprotoent (void)
```

DESCRIPTION

The *getprotobyname()* and *getprotobynumber()* functions each return a pointer to an object with the following structure containing the information from a network protocol database.

```
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.

p_prot The protocol number.

The *setprotoent()* and *endprotoent()* functions are no-ops provided for portability.

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

The functions *getprotobyname()* and *getprotobynumber()* return pointers to **protoent** structures.

All of the function return NULL on failure.

NOTES

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 available.

The INTERIX sockets implementation is built around the Microsoft Winsock DLL; to configure the protocols appropriately, see the documentation for configuring your Windows NT network.

SEE ALSO

protocols

NAME

getpwnam(), getpwuid() – user accounts database operations

SYNOPSIS

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

struct passwd * getpwnam (const char *login)
struct passwd * getpwuid (uid_t uid)
```

DESCRIPTION

These functions operate on the user accounts database file. Each function returns information in a **struct passwd** structure, which contains the following members (defined in the header file **<pwd.h>**):

```
struct passwd {
    char *    pw_name;      user name
    uid_t    pw_uid;       user ID number
    gid_t    pw_gid;       user's group ID number
    char *    pw_dir;       initial working directory
    char *    pw_shell;     initial shell program
    char *    pw_passwd;    always "*"
    char *    pw_gecos;     user information; see text
    time_t    pw_change;    not used; for future development
    time_t    pw_expire;    not used; for future development
};
```

The functions *getpwnam()* and *getpwuid()* search the user accounts database for the given login name or user uid, respectively. Every Windows NT system has its own security database (controlled by a Security Account Manager or SAM), which is identified by its “Domain” name. Every user and group account defined on that system is stored in this database. User and group names are unique within a database because they share the same name address space.

Fully-qualified user and group names always include this domain name using the following notations:

```
domain/logname
domain\logname
domain+logname
logname
```

With the following meanings:

<i>domain/logname</i>	The user <i>logname</i> in the specified <i>domain</i> .
<i>domain\logname</i>	The user <i>logname</i> in the specified <i>domain</i> .
<i>domain+logname</i>	The user <i>logname</i> in the specified <i>domain</i> . This is the format returned by INTERIX calls.
<i>+logname</i>	The user <i>logname</i> in the same domain as the calling user.
<i>logname</i>	The user <i>logname</i> in the same domain as the calling user.

Some user and group names are “well-known” in Windows NT and therefore do not have (or require) a domain prefix. Examples include SYSTEM, Everyone and Administrators.

The value returned in **passwd.pw_name** is a fully-qualified name in the format “domain+name”. The value returned in **passwd.pw_shell** is always “/bin/sh”. The value returned in **passwd.pw_gecos** is the user description string in the user database (set using the Windows NT User Manager or through the Windows NT *net.exe* program).

RETURN VALUES

The functions *getpwnam()* and *getpwuid()* return a valid pointer to a **passwd** structure on success and a NULL pointer if end-of-file is reached or an error occurs.

The return values may point to static data that is overwritten on each call.

NOTES

There is a long-standing convention that the **pw_gecos** member contains a string of comma-separated fields, traditionally:

fullname, office#, workphone#, homephone#

It's not reasonable to assume that Windows NT administrators will follow this convention. Therefore, code such as this may not return the expected information:

```
char *cp = strtok(Sender->pw_gecos, ",;");
```

SEE ALSO

getgrnam(), getgroups(), getlogin(), stripdomainprefix()

NAME

getservbyport(), getservbyname(), setservernt(), endservernt() – get service entry

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent(void)
struct servent *getservbyname (char *name, char *proto)
struct servent *getservbyport (int port, char *proto)
void setservernt (int stayopen)
void endservernt (void)
```

DESCRIPTION

The *getservent()*, *getservbyname()* and *getservbyport()* functions each return a pointer to an object with the following structure containing the information from a network services database.

```
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.

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

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

The *getservent()* function reads the next line of the file, opening the file if necessary.

The *setservernt()* and *endservernt()* functions are no-ops, provided for portability.

The *getservbyname()* and *getservbyport()* functions return information from a matching protocol name or port number. If a protocol name is also supplied (*proto* is non-NULL), searches must also match the protocol.

NOTES

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it.

RETURN VALUES

Returns NULL pointer on EOF or error.

FILES

These functions make use of the following files:

%WINDIR%/system32/drivers/etc/services

The network services database, where **%WINDIR%** is the Windows NT installation directory.

SEE ALSO

getprotoent()

NAME

getsockname() – get socket name

SYNOPSIS

```
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

If the call succeeds, it returns 0. If it fails, it returns -1 and sets *errno* to indicate the error.

ERRORS

The call succeeds unless:

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

SEE ALSO

bind(), *socket()*

NAME

getsockopt(), 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 are always present at the uppermost “socket” level and they may also exist at other protocol levels.

When manipulating socket options the level at which the option resides and the name of the option must be specified. For all the *SO_** options with the *level* *SOL_SOCKET*, the *TCP_** uses level *IPPROTO_TCP*.

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 **<sys/socket.h>** 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).

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

<i>SO_ACCEPTCONN</i>	0	socket is <i>listen()</i> ing
<i>SO_BROADCAST</i>	0 (off)	enables permission to transmit broadcast messages
<i>SO_DEBUG</i>	0 (off)	enables recording of debugging information
<i>SO_DONTROUTE</i>	0 (off)	enables routing bypass for outgoing messages
<i>SO_ERROR</i>	N / A	get and clear error on the socket (get only)
<i>SO_KEEPALIVE</i>	0 (off)	enables keep connections alive
<i>SO_LINGER</i>	0 (off)	linger on close if data present
<i>SO_OOBINLINE</i>	0 (off)	enables reception of out-of-band data in band
<i>SO_RCVBUF</i>	?	set buffer size for input

SO_REUSEADDR	0 (off)	enables local address reuse
SO_SNDBUF	?	set buffer size for output
SO_TYPE	N / A	get the type of the socket (get only)
TCP_NODELAY	0 (off)	disables the Nagle algorithm

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 limit on these values.

`SO_DEBUG` enables debugging in the underlying protocol modules.

`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_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_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 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.

`SO_REUSEADDR` indicates that the rules used in validating addresses supplied in a `bind()` call should allow reuse of local addresses.

The `TCP_NODELAY` option disables the Nagle algorithm for minimizing network congestion. Briefly, the algorithm requires that TCP segments not be sent until all outstanding data is acknowledged, or there's a maximum segment size packet to send. The stack is allowed to condense data into multibyte segments. Most applications are not affected by the Nagle algorithm and therefore should not set `TCP_NODELAY`. Some applications are affected:

- Applications that do two or more small data sends and that expect immediate responses from each.
- Applications that need a steady flow of data.

A mouse- or pointer-using application such as an X Server is an application that should disable the Nagle algorithm; it needs to track and respond to the mouse movements.

RETURN VALUES

`Getsockopt()` returns 0 if it succeeds. If it fails, it returns -1 and sets `errno` to indicate the error.

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()*

NAME

gettimeofday() – get date and time

SYNOPSIS

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

int gettimeofday (struct timeval *tp, void *tzp)
```

DESCRIPTION

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

The **timeval** structure is defined in **<sys/time.h>** as:

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

The *tzp* argument is no longer supported; any value passed in is ignored.

RETURN VALUE

The function returns 0 on success. On failure, they return -1 and set the global variable *errno* to indicate the error.

ERRORS

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

- [EFAULT] An argument address referenced invalid memory.
- [EPERM] A user without appropriate privileges attempted to set the time.

NOTES

The *tzp* argument used to be a **struct timezone**:

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

If non-zero, the **tz_dsttime** indicated that Daylight Savings time applied locally during the appropriate time of the year.

SEE ALSO

date
ctime()

NAME

gettzenv() – retrieve Win32 system TZ environment variable

SYNOPSIS

```
#include <opennt/opennt.h>

char *gettzenv(void)
```

DESCRIPTION

The *gettzenv()* call retrieves the system *TZ* environment variable instead of the user's. The call returns a pointer to a static buffer containing the system environment variable. The variable is in the form “TZ=*value*”.

If the system *TZ* variable has not been set, the call returns “TZ=UTC0”. This is usually an indication of greater problems.

This is not a standard API; you must define `_ALL_SOURCE` (for example, compile with `-D_ALL_SOURCE`).

RETURN VALUE

The *gettzenv()* function returns a pointer to a static buffer. On success, this buffer contains the environment variable. On failure, the buffer contains “TZ=UTC0”.

NAME

getuid(), geteuid() – get user identification

SYNOPSIS

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

uid_t getuid (void)
uid_t geteuid (void)
```

DESCRIPTION

The *getuid()* function returns the real user ID of the calling process. The *geteuid()* function returns the effective user ID of the calling process.

The real user ID is that of the user who has invoked the program.

The effective user ID is the group of the process, if it has the set-user-ID mode set. Since the effective user ID gives the process additional permissions during execution, *getuid()* is used to determine the real user ID of the calling process.

ERRORS

The *getuid()* and *geteuid()* functions are always successful, and no return value is reserved to indicate an error.

SEE ALSO

getgid()

NAME

glob(), globfree() – generate pathnames matching a pattern

SYNOPSIS

```
#include <glob.h>

int glob (const char *pattern, int flags,
          const int (*errfunc)(const char *, int), glob_t *pglob)
void globfree (glob_t *pglob)
```

DESCRIPTION

The *glob()* function is a pathname generator that implements the rules for file name pattern matching used by the shell.

The include file **<glob.h>** defines the structure type *glob_t*, which contains at least the following fields:

```
typedef struct {
    int gl_pathc;           /* count of total paths so far */
    int gl_matchc;          /* count of paths matching pattern */
    int gl_offs;            /* reserved at beginning of gl_pathv */
    int gl_flags;           /* returned flags */
    char **gl_pathv;        /* list of paths matching pattern */
} glob_t;
```

The argument *pattern* is a pointer to a pathname pattern to be expanded. The *glob()* argument matches all accessible pathnames against the pattern and creates a list of the pathnames that match. In order to have access to a pathname, *glob()* requires search permission on every component of a path except the last and read permission on each directory of any filename component of *pattern* that contains any of the special characters *, ? or [.

The *glob()* argument stores the number of matched pathnames into the *gl_pathc* field, and a pointer to a list of pointers to pathnames into the *gl_pathv* field. The first pointer after the last pathname is NULL. If the pattern does not match any pathnames, the returned number of matched paths is set to zero.

It is the caller's responsibility to create the structure pointed to by *pglob*. The *glob()* function allocates other space as needed, including the memory pointed to by *gl_pathv*.

The argument *flags* is used to modify the behavior of *glob()*. The value of *flags* is the bitwise inclusive OR of any of the following values defined in **<glob.h>**:

GLOB_APPEND

Append pathnames generated to the ones from a previous call (or calls) to *glob()*. The value of *gl_pathc* will be the total matches found by this call and the previous call(s). The pathnames are appended to, not merged with the pathnames returned by the previous call(s). Between calls, the caller must not change the setting of the GLOB_DOOFFS flag, nor change the value of *gl_offs* when GLOB_DOOFFS is set, nor (obviously) call *globfree()* for *pglob*.

GLOB_DOOFFS

Make use of the *gl_offs* field. If this flag is set, *gl_offs* is used to specify how many NULL pointers to prepend to the beginning of the *gl_pathv* field. In other words, *gl_pathv* will point to *gl_offs* NULL pointers, followed by *gl_pathc*

pathname pointers, followed by a NULL pointer.

GLOB_ERR

Causes *glob()* to return when it encounters a directory that it cannot open or read. Ordinarily, *glob()* continues to find matches.

GLOB_MARK

Each pathname that is a directory that matches *pattern* has a slash appended.

GLOB_NOCHECK

If *pattern* does not match any pathname, then *glob()* returns a list consisting of only *pattern*, with the number of total pathnames is set to 1, and the number of matched pathnames set to 0. If GLOB_QUOTE is set, its effect is present in the pattern returned.

GLOB_NOSORT

By default, the pathnames are sorted in ascending ASCII order; this flag prevents that sorting (speeding up *glob()*).

The following values may also be included in *flags*, however, they are non-standard extensions to POSIX.1-1988 standard.

GLOB_ALTDIRFUNC

The following additional fields in the *pglob* structure have been initialized with alternate functions for *glob* to use to open, read, and close directories and to get stat information on names found in those directories.

```
void (*gl_opendir)(const char * name);
struct dirent (*gl_readdir)(void *);
void (*gl_closedir)(void *);
int (*gl_lstat)(const char *name, struct stat *st);
int (*gl_stat)(const char *name, struct stat *st);
```

This extension is provided to allow programs such as *restore()* to provide globbing from directories stored on tape.

GLOB_BRACE

Pre-process the pattern string to expand {pat,pat,...} strings like *csh1*. The {} is left unexpanded for historical reasons (*Csh()* does the same thing to ease typing of *find()* patterns).

GLOB_MAGCHAR

Set by the *glob()* function if the pattern included globbing characters. See the description of the usage of the *gl_matchc* structure member for more details.

GLOB_NOMAGIC

Is the same as GLOB_NOCHECK but it only appends the *pattern* if it does not contain any of the special characters “*”, “?” or “[”. GLOB_NOMAGIC is provided to simplify implementing the historic *csh()* globbing behavior and should probably not be used anywhere else.

GLOB_QUOTE

Use the backslash (\) character for quoting; every occurrence of a backslash followed by a character in the pattern is replaced by that character, avoiding any special interpretation of the character.

GLOB_TILDE

Expand patterns that start with ~ to user name home directories.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is non-NULL, *glob()* calls *(*errfunc)(path,errno)*. This may be unintuitive: a pattern like **/Makefile* will try to *stat()* *foo/Makefile* even if *foo* is not a directory, resulting in a call to *errfunc*. The error routine can suppress this action by testing for *ENOENT* and *ENOTDIR*; however, the *GLOB_ERR* flag will still cause an immediate return when this happens.

If *errfunc* returns non-zero, *glob()* stops the scan and returns *GLOB_ABEND* after setting *gl_pathc* and *gl_pathv* to reflect any paths already matched. This also happens if an error is encountered and *GLOB_ERR* is set in *flags*, regardless of the return value of *errfunc*, if called. If *GLOB_ERR* is not set and either *errfunc* is NULL or *errfunc* returns zero, the error is ignored.

The *globfree()* function frees any space associated with *pglob* from a previous call(s) to *glob()*.

RETURN VALUES

On successful completion, *glob()* returns zero. In addition the fields of *pglob* contain the values described below:

gl_pathc

contains the total number of matched pathnames so far. This includes other matches from previous invocations of *glob()* if *GLOB_APPEND* was specified.

gl_matchc

contains the number of matched pathnames in the current invocation of *glob()*.

gl_flags

contains a copy of the *flags* parameter with the bit *GLOB_MAGCHAR* set if *pattern* contained any of the special characters *"*"*, *"?"* or *"["*, cleared if not.

gl_pathv

contains a pointer to a NULL-terminated list of matched pathnames. However, if *gl_pathc* is zero, the contents of *gl_pathv* are undefined.

If *glob()* terminates due to an error, it sets *errno* and returns one of the following non-zero constants, which are defined in the include file **<glob.h>**:

GLOB_NOSPACE

An attempt to allocate memory failed.

GLOB_ABEND

The scan was stopped because an error was encountered and either *GLOB_ERR* was set or *(*errfunc)()* returned non-zero.

The arguments *pglob->gl_pathc* and *pglob->gl_pathv* are still set as specified above.

EXAMPLE

A rough equivalent of *ls -l *.c *.h* can be obtained with the following code:

```
glob_t g;
g.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &g);
glob("*.h", GLOB_DOOFFS | GLOB_APPEND, NULL, &g);
g.gl_pathv[0] = "ls";
g.gl_pathv[1] = "-l";
execvp("ls", g.gl_pathv);
```

SEE ALSO

sh

fnmatch(), *regexec()*

BUGS

Patterns longer than {MAXPATHLEN} may cause unchecked errors.

The *glob()* argument may fail and set *errno* for any of the errors specified for the library routines *stat()*, *closedir()*, *opendir()*, *readdir()*, *malloc()*, and *free()*.

NAME

gmtime() – transform binary date and time value to ASCII UTC form

SYNOPSIS

```
#include <time.h>

struct tm * gmtime (const time_t *clock)
```

DESCRIPTION

The function *gmtime()* takes as an argument a time value representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970; see *time()*). It converts the time value pointed at by *clock*, and returns a pointer to a “**struct tm**” (described below) which contains the broken-out time information for the value without adjusting for time zones or other factors such as Daylight Saving Time). It then returns a pointer to a **tm** structure (described below).

```
Thu Nov 24 18:22:48 1986\n\0
```

All the fields have constant width.

External declarations as well as the **tm** structure definition are in the **<time.h>** include file. The **tm** structure includes at least the following fields:

int	<i>tm_sec;</i>	seconds (0 – 60)
int	<i>tm_min;</i>	minutes (0 – 59)
int	<i>tm_hour;</i>	hours (0 – 23)
int	<i>tm_mday;</i>	day of month (1 – 31)
int	<i>tm_mon;</i>	month of year (0 – 11)
int	<i>tm_year;</i>	year – 1900
int	<i>tm_wday;</i>	day of week (Sunday = 0)
int	<i>tm_yday;</i>	day of year (0 – 365)
int	<i>tm_isdst;</i>	is summer time in effect?

The field *tm_isdst* is non-zero if Daylight Saving Time is in effect.

SEE ALSO

date

asctime(), *ctime()*, *localtime()*, *getenv()*, *time()*, *tzset()*

NAME

grantpt(), unlockpt() – Manipulate pty descriptors

SYNOPSIS

```
#include <stdlib.h>

int grantpt(int fildev)

int unlockpt(int fildev)
```

DESCRIPTION

The *grantpt()* function takes as an argument a master pseudo-terminal device, *fildev*, and changes the mode and ownership of the corresponding slave pseudo-terminal device. The slave's user ID is set to the calling process' real user ID, and the slave's permissions are set to readable and writable by the owner, and writable by the group.

Note: This implementation catches errors but does not actually change the mode and ownership.

The *unlockpt()* function unlocks the slave pseudo-terminal device associated with a master; the master is indicated by *fildev*.

RETURN VALUE

Both *grantpt()* and *lockpt()* return 0 on success, or -1 on failure. If one of these functions fails, it sets *errno* to indicate the error.

ERRORS

Grantpt() and *unlockpt()* may fail for the following reasons:

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

[EINVAL] The descriptor *fildev* doesn't actually indicate a master pseudo-terminal device.

Grantpt() may also fail for the following reason:

[EACCES] The corresponding slave pseudo-terminal couldn't be accessed.

NOTES

These implementations will catch errors, but they do not actually change the mode and ownership (for *grantpt()*) or unlock the slave (for *unlockpt()*).

SEE ALSO

ioctl(), *open()*, *ptsname()*
pty, *tty*

NAME

`hcreate()`, `hcreate_r()`, `hdestroy()`, `hdestroy_r()`, `hsearch()`, `hsearch_r()`, – manage hash search table

SYNOPSIS

```
#include <search.h>

int hcreate (size_t nel)
void hdestroy(void)
ENTRY *hsearch (ENTRY item, ACTION action)

void *hcreate_r(size_t nel)
void hdestroy_r (void *h)
ENTRY *hsearch_r(void *h, ENTRY item), ACTION action)
```

DESCRIPTION

The `hcreate()`, `hdestroy()`, and `hsearch()` functions manage hash search tables. They create, destroy, and search a given table.

First the table must be created using `hcreate()`. The argument `nel` is an estimate of the maximum number of elements the table will hold. (For better performance, the function may actually create a table with more than `nel` elements.)

To destroy the table, call `hdestroy()`.

To search and add items to the hash table, call `hsearch()`. It returns a pointer into the hash table. The function takes an **ENTRY** structure and an **ACTION** enumerated type; both are defined in `<search.h>`. The **ENTRY** defines the item being searched for and the **ACTION** defines the action to take if the item is not found.

An **ENTRY** is defined as:

```
typedef struct {
    char *key;
    char *data;
} ENTRY;
```

The `item.key` member points to a comparison key and the `item.data` member points to any other data associated with that key. For comparison, `hsearch()` uses the `strcmp()` function.

The enumeration type **ACTION** is defined as:

```
enum { FIND, ENTER, REMOVE } ACTION;
```

(The type REMOVE is an INTERIX extension.)

If the `action` argument to `hsearch()` is FIND, the function returns the null pointer. If the `action` argument is ENTER, the function inserts the item into the hash table and returns a pointer to the item. If the `action` is REMOVE, the `hsearch()` function looks from the indicated item and removes it from the hash table.

The `hcreate_r()`, `hdestroy_r()`, and `hsearch_r()` functions are reentrant versions of the standard functions. The `context` pointer refers to a structure as returned by `hcreate_r()`. These functions are only available if `_ALL_SOURCE` is defined.

Both of the `hsearch()` functions make copies of the ENTERed keys in a series of string tables. The application does not need to keep storing the keys after the keys return.

Because of this, the strings in the ENTRY structures by those functions are private and cannot be overwritten.

RETURN VALUES

The *hcreate()* function returns non-zero on success and 0 if it cannot create the hash table.

On success, the *hsearch()* function returns a pointer to the item; on failure it returns a null pointer. It can fail if the action is FIND and the item could not be found, or if the action is ENTER and the table is full.

The *hdestroy()* and *hdestroy_r()* functions don't return a value.

On success, the *hcreate_r()* function returns a pointer to a *context* structure. On failure, it returns **(void *)0** and sets the global variable *errno*.

On success, the *hsearch_r()* with the FIND or ENTER function returns a pointer to the item; with the argument REMOVE, it returns the dataportion of the removed element cast from a **void *** to an **ENTRY ***. On failure it returns **(void *)0** and sets the global variable *errno*.

ERRORS

The *hcreate()* function may fail for these reasons:

- [EBUSY] *Hcreate()* has already been called and the resultant hash table has not yet been destroyed.
- [ENOMEM] Not enough memory available.

The *hsearch()* and *hsearch_r()* functions may fail for these reasons:

- [ENOENT] If called with either FIND or REMOVE, the requested key was not found in the hash table.

The *hcreate_r()* function can fail for these reasons:

- [ENOMEM]

EXAMPLES

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

struct object_struct {
    char *name;
    size_t len;

    ...
};

/*
 * Place the supplied object into a hash table based upon its name.
 */
static int
hash_insert(struct object_struct *obj)
{
    ENTRY e, *r;
```

```

        e.key = obj->name;
        e.data = (void *)obj;

        r = hsearch(e, ENTER);

        if ((ENTRY *)0 == r)
            return 0;

        return 1;
    }

    /*s
     * Retrieve an object from the hash table based upon it's name.
     */
    static struct object_struct *
    hash_get(char *name)
    {
        ENTRY e, *r;

        if ((char *)0 == name || '\0' == *name)
            return (struct object_struct *)0;

        e.key = name;
        e.data = (void *)0;

        r = hsearch(e, FIND);

        if ((ENTRY *)0 == r)
            return (struct object_struct *)0;

        return (struct object_struct *)r->data;
    }

    int
    main(int argc, char *argv[])
    {
        struct object_struct *new, *obj;
        char *name;

        ...

        /*l
         * Instantiate the hash table.
         */
        hcreate(1024);

        ...

        /*l

```



```

    * Store a newly defined object.
    */
    if (!hash_insert(new))
        fprintf(stderr, "Could not insert object into the hash table!\n");

    ...

    /*l
    * Retrieve an object by name.
    */
    if ((struct object_struct *)0 == (obj=hash_get(name)))
        fprintf(stderr,
            "Could not find object '%s' in the hash table!\n",
            name);

    ...

    /*l
    * Release the hash table's resources.
    */
    hdestroy();

    ...
}

```

SEE ALSO

bsearch(), *lsearch()*, *tsearch()*

NAME

htonl(), htons(), ntohl(), ntohs() – convert values between host and network byte order

SYNOPSIS

```
#include <arpa/inet.h>

in_addr_t htonl (unsigned long hostlong)
in_port_t htons (unsigned short hostshort)
in_addr_t ntohl (unsigned long netlong)
in_port_t ntohs (unsigned short netshort)
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order.

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

NOTES

The *htons()* and *ntohs()* functions *must* have the correct prototype. If not declared, they are assumed to return an **int**, and there is no guarantee that the high word will be zeroed.

SEE ALSO

gethostbyname()

NAME

hypot(), cabs(), z_abs() – euclidean distance and complex absolute value functions

SYNOPSIS

```
#include <math.h>

double hypot (double x, double y)

struct complex {double x, y;} z;
double cabs (z)
double z_abs (struct z *z)
```

DESCRIPTION

The *hypot()*, *cabs()*, and *z_abs()* functions compute the $\sqrt{x^2 + y^2}$ in such a way that underflow will not happen, and overflow occurs only if the final result deserves it.

$\text{hypot}(\infty, v) = \text{hypot}(v, \infty) = +\infty$ for all v , including *NaN*.

The *z_abs()* function is similar to *cabs()* except that it takes a pointer to a **struct z** rather than the structure itself.

ERROR (due to Roundoff, etc.)

Below 0.97 *ulps*. Consequently $\text{hypot}(5.0, 12.0) = 13.0$ exactly; in general, *hypot* and *cabs* return an integer whenever an integer might be expected.

The same cannot be said for the shorter and faster version of *hypot* and *cabs* that is provided in the comments in *cabs.c*; its error can exceed 1.2 *ulps*.

NOTES

As might be expected, $\text{hypot}(v, \text{NaN})$ and $\text{hypot}(\text{NaN}, v)$ are *NaN* for all *finite* v ; with "reserved operand" in place of "NaN", the same is true on a VAX. But programmers on machines other than a VAX (if has no ∞) might be surprised at first to discover that $\text{hypot}(\pm\infty, \text{NaN}) = +\infty$. This is intentional; it happens because $\text{hypot}(\infty, v) = +\infty$ for *all* v , finite or infinite. Hence $\text{hypot}(\infty, v)$ is independent of v . Unlike the reserved operand fault on a VAX, the IEEE *NaN* is designed to disappear when it turns out to be irrelevant, as it does in $\text{hypot}(\infty, \text{NaN})$.

Given the following:

```
#include <math.h>

double a, b, r;
complex z;

a = 2.0;
b = 3.0;

z.x = 2.0;
z.y = 3.0;
```

The following give identical values for "r":

```
r = hypot(a, b);
r = cabs(z);
r = z_abs(&z);
```

RETURN VALUE

On success, *hypot()*, *cabs()*, and *z_abs()* return the length of the hypotenuse of a right-angled triangle with sides of length *x* and *y*.

If the result would cause overflow, they return *NaN* and set *errno*. ∞ .Gv *errno* . If *x* or *y* is *NaN*, they return *NaN* and set *errno*.

ERRORS

These functions can fail for the following reasons:

- | | |
|----------|--|
| [EDOM] | One of the arguments is <i>NaN</i> . |
| [ERANGE] | The result would cause overflow (in which case HUGE_VAL is returned) or underflow (in which case 0 is returned). |

SEE ALSO

math, *sqrt()*

NAME

`copysign()`, `drem()`, `finite()`, `ilogb()`, `nextafter()`, `remainder()`, `scalbn()` – functions for IEEE arithmetic

SYNOPSIS

```
#include <math.h>

double copysign (double x, double y)
double drem (double x, double y)
int finite (double x)
int ilogb (double x)
double nextafter (double x, double y)
double remainder (double x, double y)
double scalbn (double x, int n)
```

DESCRIPTION

`copysign()` returns x with its sign changed to y 's.

The `drem()` function returns the remainder $r := x - n*y$ where n is the integer nearest the exact value of x/y ; moreover if $|n - x/y| = 1/2$ then n is even. Consequently the remainder is computed exactly and $|r| \leq |y|/2$. But `drem(x,0)` is exceptional. See below under “RETURN VALUE”.)

`finite()` returns the value 1 just when $-\infty < x < +\infty$; otherwise a zero is returned (when $|x| = \infty$ or x is *NaN*).

`ilogb()` returns x 's exponent n , in integer format. `ilogb(±f)` returns `INT_MAX` and `ilogb(0)` returns `-INT_MIN`.

`nextafter()` returns the next machine representable number from x in direction y . The `nextafter()` function sets `errno` on error, so to check for errors, set `errno` to 0 before calling `nextafter()` and check its value after the call.

`remainder()` returns the remainder $r := x - n*y$ where n is the integer nearest the exact value of x/y ; moreover if $|n| - |x/y| = 1/2$ then n is even. Consequently the remainder is computed exactly and $|r| < |y|/2$. But `remainder(x,0)` and `remainder(∞,0)` are invalid operations that produce a *NaN*.

`scalbn()` returns $x*(2**n)$ computed by exponent manipulation.

RETURN VALUE

The `copysign()` function returns the value of x with the same sign as y .

`drem(x,0)` and `drem(∞,y)` are invalid operations that produce a *NaN*.

The `finite()` function returns 1 if the argument x is neither $\pm\infty$. Otherwise it returns 0.

The `ilogb()` function returns x 's exponent n , in integer format.

The `nextafter()` function returns the next representable double-precision floating point number after x in the direction of y .

The `remainder()` function returns

The `scalbn()` function returns

ERRORS

The `nextafter()` function can fail for the following reasons:

[EDOM] One of the arguments is *NaN*.

[ERANGE] The correct value would overflow.

The *remainder()* function can fail for the following reasons:

[EDOM] The *x* argument is 0 or the *y* argument is $\pm\infty$, or either *x* or *y* is *NaN*.

SEE ALSO

math

NAME

logb(), scalb(), significand() – IEEE test functions

SYNOPSIS

```
#include <math.h>

double logb (double x)
double scalb (double x, double n)
double significand (double x)
```

DESCRIPTION

These functions allow users to test conformance to the IEEE 754 standard.

logb() returns the exponent n of the argument x , n is a signed integer converted to double-precision floating-point. Specifically, it's the integral part of $\log_r |x|$, where r is the radix of the machine's floating point arithmetic. $\log_b(\pm\infty) = +\infty$; $\log_b(0) = -\infty$ with a division by zero exception.

The *scalb(x,n)* function returns $x(r^{**}n)$, where r is the radix of the machine's floating point arithmetic. (For $r=2$, this is identical to *ldexp()*.) The value is computed by exponent manipulation without first computing $r^{**}n$.

significand() returns *sig*, where the argument $x := sig * 2^{**}n$ with $1 \leq sig < 2$. The *significand()* value are not defined when x is 0, $\pm\infty$, or *NaN*.

RETURN VALUES

The *logb()* function returns the exponent of x . If x is zero, it returns `-HUGE_VAL` and sets *errno*. If x is $\pm\infty$, it returns ∞ . If x is *NaN*, it returns *NaN* and sets *errno*.

The *scalb()* function returns the value $x(r^{**}n)$ where r is the radix of the machine's floating point arithmetic.

ERRORS

The *logb()* function can fail for the following reasons:

[EDOM] Either the x argument is zero, or it is *NaN*.

The *scalb()* function can fail for the following reasons:

[EDOM] Either x or n is *NaN*.

[ERANGE] The correct value will overflow or underflow.

NOTES

Note that the function *ldexp()* is slightly more portable than *scalb()*, and that *frexp()* does the same jobs as *scalb()* and *significand()*, in a slightly more portable manner. Both *ldexp()* and *frexp()* use the radix as base for the exponent, rather than 2.

SEE ALSO

frexp(), *ieee()*, *ldexp()*, *math*

NAME

index() – locate character in string

SYNOPSIS

```
#include <strings.h>
```

```
char * index (const char *s, int c)
```

DESCRIPTION

The *index()* function locates the first character matching *c* (converted to a **char**) in the null-terminated string *s*.

RETURN VALUES

A pointer to the character is returned if it is found; otherwise NULL is returned. If *c* is `'\0'`, *index()* locates the terminating `'\0'`.

SEE ALSO

memchr(), *rindex()*, *strchr()*, *strcspn()*, *strpbrk()*, *strrchr()*, *strsep()*, *strspn()*, *strstr()*, *strtok()*

NAME

inet_aton(), inet_addr(), inet_network(), inet_ntoa(), inet_makeaddr(), inet_lnaof(),
inet_netof() – Internet address manipulation routines

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton (char *cp, struct in_addr *pin)
unsigned long inet_addr (char *cp)
unsigned long inet_network (char *cp)
char * inet_ntoa (struct in_addr in)
struct in_addr inet_makeaddr (int net, int 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.

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 interpreted as decimal).

DIAGNOSTICS

The constant INADDR_NONE is returned by *inet_addr()* and *inet_network()* for malformed requests.

NOTES

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**.

SEE ALSO

gethostbyname()

NAME

`infnan()` – signals invalid floating-point operations on a VAX (temporary)

SYNOPSIS

```
#include <math.h>

double infnan (int iarg)
```

DESCRIPTION

The Invalid, Overflow and Divide-by-Zero exceptions of the IEEE standard are approximated on INTERIX by calls to a procedure `infnan()` in appropriate places in *libm*(3). This is a portability decision. As exception-handling is improved on various systems, only `infnan()` among the codes in *libm* will have to be changed. And users of *libm* can design their own `infnan()` now to insulate themselves from future changes.

Whenever an elementary function code in *libm* has to simulate one of the aforementioned IEEE exceptions, it calls `infnan(iarg)` with an appropriate value of *iarg*. Then a reserved operand fault stops computation. But `infnan()` could be replaced by a function with the same name that returns some plausible value, assigns an apt value to the global variable *errno*, and allows computation to resume. Alternatively, the Reserved Operand Fault Handler could be changed to respond by returning that plausible value, etc. instead of aborting.

In the table below, the first two columns show various exceptions signaled by the IEEE standard, and the default result it prescribes. The third column shows what value is given to *iarg* by functions in *libm* when they invoke `infnan(iarg)` under analogous circumstances on a VAX. Currently `infnan()` stops computation under all those circumstances. The last two columns offer an alternative; they suggest a setting for *errno* and a value for a revised `infnan()` to return. And a C program to implement that suggestion follows.

IEEE Signal	IEEE Default	VAX Default	Suggested errno	Suggested return
Invalid	∞	EDOM	EDOM	0
Overflow	<i>NaN</i>	EDOM	EDOM	0
Overflow	∞	ERANGE	ERANGE	0
Div-by-0	∞	ERANGE	ERANGE or EDOM	+HUGE

(HUGE = 1.7e38 ... nearly 2.0^{*127})

ALTERNATIVE `infnan()`:

```
#include <math.h>
#include <errno.h>
extern int  errno ;
double infnan(iarg)
int  iarg ;
{
    switch(iarg) {
    case  ERANGE:      errno = ERANGE; return(HUGE);
    case  -ERANGE:     errno = EDOM; return(-HUGE);
    default:           errno = EDOM; return(0);
    }
}
```

ERRORS

The *infnan()* function can generate the following errors:

[EDOM]

[ERANGE]

SEE ALSO

math, *errno*, *signal()*.

NAME

ioctl() – control a socket or file

SYNOPSIS

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

```
ioctl (int fd, int command, void *arg);
```

DESCRIPTION

The *ioctl()* function manipulates the underlying device parameters. In particular, many operating characteristics of terminals and sockets may be controlled with *ioctl()* requests. The argument *fd* must be an open file descriptor.

Encoded in an *ioctl()* request is whether the argument is an “in” parameter or an “out” parameter, and the size of the argument *arg* in bytes.

The possible values for *command* are defined in **<sys/ioctl.h>** and include:

- | | |
|------------|---|
| FIONBIO | Sets or clears on-blocking I/O. The <i>arg</i> should point to an int . If the int is non-zero, the flag is set and non-blocking mode is enabled; if it is zero, the flag is cleared, and non-blocking mode is disabled. |
| FIONREAD | Returns in <i>arg</i> the number of bytes available to read from <i>fd</i> ; <i>arg</i> should point to an int . |
| SIOCATMARK | For sockets; this boolean returns true if the socket’s read pointer is at the out-of-band mark. |
| TIOCEXT | For pseudo terminals only; when this flag is non-zero (on), external processing mode is enabled. When zero, external processing mode is disabled. External processing mode places the terminal in a mode where characters are never echoed, special characters are never processed, and no characters are erased. This mode is often used in programs like <i>telnetd</i> . |
| TIOCSIG | For pseudo terminals only; the <i>arg</i> should point to an int . Specifies a signal that can be applied to the terminal as if the signal were to be keyboard-generated. Valid signals are SIGTSTP, SIGINT, and SIGQUIT. |

The call

```
ioctl(d, FIONBIO, NULL)
```

is equivalent to a call to

```
fcntl(d, F_SETFL, O_NONBLOCK)
```

except that for each *fcntl()* call, all of the flag values must set, while the *ioctl()* call allows you to set only one flag.

RETURN VALUE

The return value of the *ioctl()* depends upon the *command*. If an error occurs, it returns -1 and sets *errno* to indicate the error.

ERRORS

Ioctl() will fail if:

- | | |
|---------|--|
| [EBADF] | <i>Fd</i> is not a valid open file descriptor. |
| [EINTR] | The function was interrupted by a signal. |

[EINVAL] the data to which *arg* points is not valid.

NOTES

Traditional implementations of *ioctl()* also handle the terminal interface. For terminal manipulation (traditionally handled with TIOCGETD, TIOCSETD, TIOCGETP, and TIOCSETP or with *stty()* and *gtty()*), use the POSIX calls *tcgetattr()*, *tcsetattr()*, *cfgetispeed()*, *cfsetispeed()*, *cfgetospeed()*, and *cfsetospeed()* to manipulate the **termios** structure, instead.

SEE ALSO

execve(), *fcntl()*

NAME

isalnum() – alphanumeric character test

SYNOPSIS

```
#include <ctype.h>

int isalnum (int c)
```

DESCRIPTION

The *isalnum()* call tests for any character for which *isalpha()* or *isdigit()* is true.

RETURN VALUES

The *isalnum()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalpha(), *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

isalpha() – alphabetic character test

SYNOPSIS

```
#include <ctype.h>

int isalpha (int c)
```

DESCRIPTION

In the POSIX / C locale, the *isalpha()* function tests for any character for which *isupper()* or *islower()* is true. More generally, it tests for any character for which none of *iscntrl()*, *isdigit()*, *ispunct()*, or *isspace()* is true.

RETURN VALUES

The *isalpha()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(), *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

isascii – test for ASCII character

SYNOPSIS

```
#include <ctype.h>

int isascii (int c)
```

DESCRIPTION

The *isascii()* function tests for an ASCII character, which is any character with a value less than or equal to 0177.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

isblank() – white-space character test

SYNOPSIS

```
#include <ctype.h>
```

```
int isblank (int c)
```

DESCRIPTION

The *isblank()* function tests whether *c* is a character of the class “blank” in the program’s current locale. For the C and POSIX locales, this is the characters “space” and “horizontal tab”.

RETURN VALUES

The *isblank()* function returns non-zero if the character is a blank character and returns zero if the character is not.

SEE ALSO

isalnum(), *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

iscntrl() – control character test

SYNOPSIS

```
#include <ctype.h>
```

```
int iscntrl (int c)
```

DESCRIPTION

The *iscntrl()* function tests for any control character.

RETURN VALUES

The *iscntrl()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

isdigit() – decimal-digit character test

SYNOPSIS

```
#include <ctype.h>
```

```
int isdigit (int c)
```

DESCRIPTION

The *isdigit()* function tests for any decimal-digit character. In the POSIX/C locale, this is the characters:

0 1 2 3 4 5 6 7 8 9

RETURN VALUES

The *isdigit()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

isgraph() – printing character test (space character exclusive)

SYNOPSIS

```
#include <ctype.h>

int isgraph (int c)
```

DESCRIPTION

The *isgraph()* function tests for any printing character except space.

RETURN VALUES

The *isgraph()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

isinf() – test number for infinite value

SYNOPSIS

```
#include <math.h>

isinf (double x)
```

DESCRIPTION

The *isinf()* function tests the argument *x* to see if it is effectively infinite. Both positive and negative infinities are tested.

RETURN VALUES

The *isinf()* function returns 1 if the argument *x* is positive or negative infinity. It returns 0 otherwise.

SEE ALSO

isnan(), *math*

NAME

islower() – lower-case character test

SYNOPSIS

```
#include <ctype.h>
```

```
int islower (int c)
```

DESCRIPTION

The *islower()* function tests for any lower-case letters. In the POSIX/C locale, this includes the following characters:

“a”	“b”	“c”	“d”	“e”
“f”	“g”	“h”	“i”	“j”
“k”	“l”	“m”	“n”	“o”
“p”	“q”	“r”	“s”	“t”
“u”	“v”	“w”	“x”	“y”
“z”				

RETURN VALUES

The *islower()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

isnan() – test argument for Not A Number

SYNOPSIS

```
#include <math.h>

isnan (double x)
```

DESCRIPTION

The *isnan()* function tests the argument *x* to see if it is not a number.

RETURN VALUES

The *isnan()* function returns 1 if the argument *x* is *NaN*. It returns 0 otherwise.

SEE ALSO

isnan(), *math*

NAME

isprint() – printing character test (space character inclusive)

SYNOPSIS

```
#include <ctype.h>

int isprint (int c)
```

DESCRIPTION

The *isprint()* function tests for any printing character including space (' ').

RETURN VALUES

The *isprint()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

ispunct() – punctuation character test

SYNOPSIS

```
#include <ctype.h>
```

```
int ispunct (int c)
```

DESCRIPTION

The *ispunct()* function tests for any printing character except for space (' ') or a character for which *isalnum()* is true.

RETURN VALUES

The *ispunct()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

isspace() – white-space character test

SYNOPSIS

```
#include <ctype.h>
```

```
int isspace (int c)
```

DESCRIPTION

The *isspace()* function tests for the standard white-space characters. This is the obverse of *isalnum()*.

RETURN VALUES

The *isspace()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

isupper() – upper-case character test

SYNOPSIS

```
#include <ctype.h>
```

```
int isupper (int c)
```

DESCRIPTION

The *isupper()* function tests for any upper-case letter.

RETURN VALUES

The *isupper()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

isxdigit() – hexadecimal-digit character test

SYNOPSIS

```
#include <ctype.h>

int isxdigit (int c)
```

DESCRIPTION

The *isxdigit()* function tests for any hexadecimal-digit character. In the POSIX/C locale, this is the characters:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

RETURN VALUES

The *isxdigit()* function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *tolower()*, *toupper()*

NAME

j0(), j1(), jn(), y0(), y1(), yn() – bessel functions of first and second kind

SYNOPSIS

```
#include <math.h>

double j0 (double x)
double j1 (double x)
double jn (int n, double x)
double y0 (double x)
double y1 (double x)
double yn (int n, double x)
```

DESCRIPTION

The functions *j0()* and *j1()* compute the *Bessel* function of the first kind of the order 0 and the *order* 1, respectively, for the real value *x*; the function *jn()* computes the *Bessel* function of the first kind of the integer order *n* for the real value *x*.

The functions *y0()* and *y1()* compute the linearly independent *Bessel* function of the second kind of the order 0 and the *order* 1, respectively, for the positive *integer* value *x* (expressed as a **double**); the function *yn()* computes the *Bessel* function of the second kind *n* for the positive *integer* value *x* (expressed as a double).

RETURN VALUES

If these functions are successful, the computed value is returned.

SEE ALSO

math, *infnan()*

NAME

killpg() – send signal to a process group

SYNOPSIS

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

int killpg (pid_t pgrp, int signum)
```

DESCRIPTION

Killpg() sends the signal *signum* to the process group *pgrp*. See *sigaction()* for a list of signals. If *pgrp* is 0, *killpg()* sends the signal to the sending process's process group.

The *killpg()* call is successful if the sending process has permission to send *signum* to any process in the process group list. To be successful, the sending process must have the same user ID as the process group member. As a single special case the continue signal SIGCONT may be sent to any process that is a descendant of the current process.

RETURN VALUES

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

ERRORS

Killpg() will fail and no signal will be sent if:

[EINVAL]	<i>Signum</i> is not a valid signal number.
[ESRCH]	No process can be found in the process group specified by <i>pgrp</i> .
[ESRCH]	The process group was given as 0 but the sending process does not have a process group.
[EPERM]	The sending process is not the super-user and one or more of the target processes has an effective user ID different from that of the sending process.

SEE ALSO

kill(), *getpgrp()*, *sigaction()*, *signal()*

NAME

kill() – send signal to a process

SYNOPSIS

```
#include <signal.h>

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

DESCRIPTION

The *kill()* function sends the signal given by *sig* to *pid*, a process or a group of processes. *Sig* may be one of the signals specified for *sigaction()* or it may be 0. If *sig* is 0, error checking is performed but no signal is actually sent. (This can be used to check the validity of *pid*.)

For a process to have permission to send a signal to the process *pid*, the real or effective user ID of the receiving process must match the real or effective user ID of the sending process or the user must have appropriate privileges. These user ID tests are not applied when sending SIGCONT to a process that is a member of the same session as the sending process.

- If *pid* > 0: *Sig* is sent to the process whose ID is equal to *pid*.
- If *pid* = 0: *Sig* is sent to all processes whose group ID is equal to the process group ID of the sender, and for which the process has permission.
- If *pid* = -1: *Sig* is sent to all processes for which the process has permission to send that signal.
- If *pid* < -1: *Sig* is sent to all processes whose group ID is equal to the absolute value of *pid*, and for which the process has permission.

The *kill()* function only delivers signals to processes in the POSIX subsystem (that is, POSIX processes), not to Win32 processes. If you attempt to *kill()* a process which has *exec()*ed a Win32 process, the signal isn't delivered to the Win32 process.

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

Kill() will fail and no signal will be sent if:

- [EINVAL] *Sig* is not a valid signal number.
- [ESRCH] No process can be found corresponding to that specified by *pid*.
- [EPERM] The effective user ID of the sending process does not match the effective user-id of the receiving process. When signaling a process group, this error is returned if any members of the group could not be signaled.

SEE ALSO

getpid(), *getpgid()*, *sigaction()*

NAME

`labs()` – return the absolute value of a long integer

SYNOPSIS

```
#include <stdlib.h>

long labs (long j)
```

DESCRIPTION

The `labs()` function returns the absolute value of the **long integer** *j*.

RETURN VALUE

`Labs()` returns a **long integer**.

SEE ALSO

`abs()`, `cabs()`, `floor()`

NAME

`ldexp()` – multiply floating-point number by integral power of 2

SYNOPSIS

```
#include <math.h>

double ldexp (double x, int exp)
```

DESCRIPTION

The `ldexp()` function multiplies a floating-point number by an integral power of 2.

RETURN VALUES

The `ldexp()` function returns the value of *x* times 2 raised to the power *exp*.

If the resultant value would cause an overflow, the global variable *errno* is set and the value {HUGE_VAL} is returned.

ERRORS

`ldexp()` can fail because:

[ERANGE] The result would cause an overflow.

SEE ALSO

`frexp()`, `modf()`

NAME

ldiv() – return quotient and remainder from division

SYNOPSIS

```
#include <stdlib.h>
```

```
ldiv_t ldiv (int num, int denom)
```

DESCRIPTION

The *ldiv()* function computes the value *num* / *denom* and returns the quotient and remainder in a structure named **ldiv_t** that contains two **long integer** members named *quot* and *rem*.

SEE ALSO

div()

NAME

lgamma(), gamma(), lgamma_r(), gamma_r() – log gamma function, gamma function, signgam variable

SYNOPSIS

```
#include <math.h>
extern int signgam;

double lgamma (double x)
double gamma (double x)
double lgamma_r (double x, int *signgamp)
double gamma_r (double x, int *signgamp)
```

DESCRIPTION

Gamma(x) returns $\Gamma(x)$, with no effect on *signgam*.

Depending on the value of the variable *_LIB_VERSION*, *lgamma()* may be the same as *gamma()* (as specified by the XPG) or it may return $\ln |\Gamma(x)|$ where

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt \text{ for } x > 0 \text{ and}$$

$$\Gamma(x) = \pi / (\Gamma(1-x) \sin(\pi x)) \text{ for } x < 1.$$

See “NOTES”

The external integer *signgam* returns the sign of $\Gamma(x)$.

The functions *gamma_r()* and *lgamma_r()* are reentrant versions of *gamma()* and *lgamma()* (where *lgamma_r()* always returns the logarithm of the gamma function). The argument *signgamp* is a pointer to a user-provided variable which will contain the sign of the *x* after the call returns.

IDIOSYNCRASIES

Do not use the expression “*signgam*exp(lgamma(x))*” to compute $g := \Gamma(x)$. Instead use a program like this (in C):

```
lg = lgamma(x); g = signgam*exp(lg);
```

Only after *lgamma()* has returned can *signgam* be correct.

For arguments in its range, *gamma()* is preferred, as for positive arguments it is accurate to within one unit in the last place. Exponentiation of *lgamma()* will lose up to 10 significant bits.

RETURN VALUES

Gamma() and *lgamma()* return appropriate values unless an argument is out of range. Overflow will occur for sufficiently large positive values. For large non-integer negative values, *gamma()* will underflow.

ERRORS

The *lgamma()* function can fail for the following reasons:

- | | |
|----------|---|
| [EDOM] | The argument <i>x</i> is a non-positive integer or <i>NaN</i> . |
| [ERANGE] | The result would have caused an underflow or overflow. |

NOTES

The name *gamma()* was originally dedicated to the *lgamma()* function, so some old code

may no longer be compatible. For this reason, the XPG specifications make *gamma()* and *lgamma()* identical. If you want *lgamma()* to actually return the logarithm, set the value of the global variable `_LIB_VERSION` to the value `_BSD44_` (the default value is `_XOPEN_`).

The variable `_LIB_VERSION` dictates the conformance aspects of the math APIs, primarily errors and boundary conditions. The code has only been tested for conformance with the value `_XOPEN_`.

To avoid possible conflicts in behavior, set `_LIB_VERSION` to `_BSD44_` immediately before the call to *lgamma()* and set it back to `_XOPEN_` immediately after.

SEE ALSO

math

NAME

link() – make a hard file link

SYNOPSIS

```
#include <unistd.h>
```

```
int link (const char *oldname, const char *newname)
```

DESCRIPTION

The *link()* function call atomically creates the specified directory entry (hard link) *newname* with the attributes of the underlying object pointed at by *oldname*. If the link is successful: the link count of the underlying object is incremented; *oldname* and *newname* share equal access and rights to the underlying object.

If *oldname* is removed, the file *newname* is not deleted and the link count of the underlying object is decremented.

Oldname must exist for the hard link to succeed and both *oldname* and *newname* must be in the same file system. *Oldname* may not be a directory; NTFS does not support hard-linking a directory.

You cannot use *link()* on a FAT file system.

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. |
| [EINVAL] | Either pathname contains a character with the high-order bit set. |
| [ENAMETOOLONG] | A component of either pathname exceeded {NAME_MAX} characters, or entire length of either pathname exceeded {PATH_MAX} characters. |
| [ENOENT] | A component of either path prefix 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. |
| [ENOENT] | The file named by <i>oldname</i> does not exist. |
| [EEXIST] | The link named by <i>newname</i> does exist. |
| [EPERM] | The file named by <i>oldname</i> is a directory. |
| [EXDEV] | The link named by <i>newname</i> and the file named by <i>oldname</i> are on different file systems. |
| [ENOSPC] | The directory in which the entry for the new link is being placed cannot be extended because there is no space left on the file system. |

containing the directory.

[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

unlink()

NAME

`listen()` – listen for connections on a socket

SYNOPSIS

```
#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 `[ECONNREFUSED,]` or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

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]` 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()`

BUGS

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

NAME

localeconv() – Get locale-specific rules for formatting numeric quantities

SYNOPSIS

```
#include <locale.h>

struct lconv *localconv (void)
```

DESCRIPTION

The *localeconv()* function returns a pointer to a **struct lconv** which describes the current locale. This is used to make a program specific to a particular language, cultural convention and codeset.

Currently, the POSIX subsystem only supports the C and POSIX locales (which are identical).

The **lconv** structure is described in the file **<locale.h>**. It contains:

Member	Default	Description
char * <i>decimal_point</i>	"."	Character used for non-monetary quantities
char * <i>thousands_sep</i>	"	Character used to separate groups of digits for non-monetary quantities
<i>char</i> * <i>grouping</i>	"	String indicating size of each group of digits for non-monetary quantities 0 — repeat previous element 1..{CHAR_MAX}-1 — number of digits in current group {CHAR_MAX} — no further grouping
char * <i>int_curr_symbol</i>	"	International currency symbol
char * <i>currency_symbol</i>	"	Local currency symbol
char * <i>mon_decimal_point</i>	"	Character used for monetary quantities
char * <i>mon_thousands_sep</i>	"	Character used to separate groups of digits for monetary quantities
char * <i>mon_grouping</i>	"	Character used to separate groups of digits for monetary quantities
char * <i>positive_sign</i>	"	String to indicate monetary quantity with non-negative value
char * <i>negative_sign</i>	"	String used to indicate monetary quantity with negative value
char <i>int_frac_digits</i>	{CHAR_MAX}	Number of digits after decimal point for internationally-formatted monetary quantities
char <i>frac_digits</i>	{CHAR_MAX}	Number of digits after the decimal point for monetary quantities
char <i>p_cs_precedes</i>	{CHAR_MAX}	1 if currency symbol precedes positive monetary values, 0 if it follows
char <i>p_sep_by_space</i>	{CHAR_MAX}	1 if a space goes between currency symbol and digits for positive monetary values, 0 if no space

Member	Default	Description
char <i>n_cs_precedes</i>	{CHAR_MAX}	1 if currency symbol precedes negative monetary values, 0 if it follows
char <i>n_sep_by_space</i>	{CHAR_MAX}	1 if a space goes between currency symbol and digits for negative monetary values, 0 if no space
char <i>p_sign_posn</i>	{CHAR_MAX}	Position of positive sign in monetary values: 0 — surround with () 1 — sign string precedes quantity and currency symbol 2 — sign string succeeds quantity and currency symbol 3 — sign string precedes currency symbol 4 — sign string immediately after currency symbol
char <i>n_sign_posn</i>	{CHAR_MAX}	Position of negative sign in monetary values: 0 — surround with () 1 — sign string precedes quantity and currency symbol 2 — sign string succeeds quantity and currency symbol 3 — sign string precedes currency symbol 4 — sign string immediately after currency symbol

RETURN VALUE

The *localeconv()* function returns a pointer to a **struct lconv** which describes the current locale.

SEE ALSO

setlocale()

NAME

localtime() – transform timer data into timer structure in local time

SYNOPSIS

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

```
struct tm * localtime (const time_t *clock)
```

DESCRIPTION

The *localtime()* function takes as an argument a time value representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970; see *time()*). It converts the time value pointed at by *clock*, and returns a pointer to a **struct tm** (described below) which contains the broken-out time information for the value after adjusting for the current time zone (and any other factors such as Daylight Saving Time). Time zone adjustments are performed as specified by the *TZ* environmental variable (see *tzset()*). The function *localtime()* uses *tzset* to initialize time conversion information if *tzset* has not already been called by the process.

After filling in the *tm* structure, *localtime()* sets the *tm_isdst*'th element of *tzname* to a pointer to an ASCII string that's the time zone abbreviation to be used with *localtime()*'s return value.

External declarations as well as the **tm** structure definition are in the **<time.h>** include file. The **tm** structure includes at least the following fields:

int	<i>tm_sec;</i>	seconds (0 – 60)
int	<i>tm_min;</i>	minutes (0 – 59)
int	<i>tm_hour;</i>	hours (0 – 23)
int	<i>tm_mday;</i>	day of month (1 – 31)
int	<i>tm_mon;</i>	month of year (0 – 11)
int	<i>tm_year;</i>	year – 1900
int	<i>tm_wday;</i>	day of week (Sunday = 0)
int	<i>tm_yday;</i>	day of year (0 – 365)
int	<i>tm_isdst;</i>	is summer time in effect?

The field *tm_isdst* is non-zero if summer time is in effect.

SEE ALSO

date

asctime(), *ctime()*, *gmtime()*, *mktime()*, *difftime()*, *getenv()*, *time()*, *tzset()*

NAME

`lsearch()`, `lfind()` – linear searching routines

SYNOPSIS

```
char * lsearch (const void *key, const void *base, size_t *nelp,  
               size_t width, int (*compar)(void *, void *))  
char * lfind (const void *key, const void *base, size_t *nelp,  
             size_t width, int (*compar)(void *, void *))
```

DESCRIPTION

The functions `lsearch()` and `lfind()` provide basic linear searching functionality.

Base is the pointer to the beginning of an array. The argument *nelp* is the current number of elements in the array, where each element is *width* bytes long. The *compar* function is a comparison routine which is used to compare two elements. It takes two arguments which point to the *key* object and to an array member, in that order, and must return an integer less than, equivalent to, or greater than zero if the *key* object is considered, respectively, to be less than, equal to, or greater than the array member.

The `lsearch()` and `lfind()` functions return a pointer into the array referenced by *base* where *key* is located. If *key* does not exist, `lfind()` will return a null pointer and `lsearch()` will add it to the array. When an element is added to the array by `lsearch()` the location referenced by the argument *nelp* is incremented by one.

SEE ALSO

`bsearch()`, `db()`

NAME

`lseek()` – reposition read / write file offset

SYNOPSIS

```
#include <unistd.h>
```

```
off_t lseek (int fd, off_t offset, int whence)
```

DESCRIPTION

The `lseek()` function repositions the offset of the file descriptor *fd* to the argument *offset* according to the directive *whence*. The argument *fd* must be an open file descriptor.

`Lseek()` repositions the file pointer *fd* 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 pointer will remain unchanged if:

- | | |
|----------|---|
| [EBADF] | <i>Fd</i> is not an open file descriptor. |
| [ESPIPE] | <i>Fd</i> is associated with a pipe, socket, or FIFO. |
| [EINVAL] | <i>Whence</i> is not a proper value. |

SEE ALSO

`dup()`, `fseek()`, `open()`

NAME

malloc() – general memory allocation function

SYNOPSIS

```
#include <stdlib.h>

void * malloc (size_t size)
```

DESCRIPTION

The *malloc()* function allocates uninitialized space for an object whose size is specified by *size*. The *malloc()* function maintains multiple lists of free blocks according to size, allocating space from the appropriate list.

RETURN VALUES

The *malloc()* function returns a pointer to the allocated space if successful; otherwise it returns NULL.

SEE ALSO

calloc(), *free()*, *realloc()*

NAME

math – introduction to mathematical library functions

DESCRIPTION

These functions constitute the C math library, *libm*. The link editor searches this library under the **-lm** option. Declarations for these functions may be obtained from the include file **<math.h>**.

LIST OF FUNCTIONS

Name	Description	Error Bound (ULPs)
acos	inverse trigonometric function	3
acosh	inverse hyperbolic function	3
asin	inverse trigonometric function	3
asinh	inverse hyperbolic function	3
atan	inverse trigonometric function	1
atanh	inverse hyperbolic function	3
atan2	inverse trigonometric function	2
cabs	complex absolute value	1
cbrt	cube root	1
ceil	integer no less than	0
copysign	copy sign bit	0
cos	trigonometric function	1
cosh	hyperbolic function	3
drem	remainder	0
erf	error function	
erfc	complementary error function	
exp	exponential	1
expm1	exp(x)-1	1
fabs	absolute value	0
floor	integer no greater than	0
hypot	Euclidean distance	1
infnan	signals exceptions	
j0	bessel function	
j1	bessel function	
jn	bessel function	
lgamma	log gamma function; (formerly gamma.3)	
log	natural logarithm	1
logb	exponent extraction	0
log10	logarithm to base 10	3
log1p	log(1+x)	1
pow	exponential x**y	60–500
rint	round to nearest integer	0
scalb	exponent adjustment	0

sin	trigonometric function	1
sinh	hyperbolic function	3
sqrt	square root	1
tan	trigonometric function	3
tanh	hyperbolic function	3
y0	bessel function	
y1	bessel function	
yn	bessel function	

NOTES

In older systems, most of these functions were provided in two versions, one for the double-precision "D" format in the DEC VAX-11 family of computers, another for double-precision arithmetic conforming to the IEEE Standard 754 for Binary Floating-Point Arithmetic. This system provides only the IEEE format, and it is described here.

Properties of IEEE 754 Double-Precision:

Wordsize: 64 bits, 8 bytes. Radix: Binary.

Precision: 53 significant bits, roughly like 16 significant decimals.

If x and x' are consecutive positive Double-Precision numbers (they differ by 1 *ulp*), then

$$1.1\text{e-}16 < 0.5^{**53} < (x'-x)/x \leq 0.5^{**52} < 2.3\text{e-}16.$$

Range:

Overflow threshold = $2.0^{**1024} = 1.8\text{e}308$

Underflow threshold = $0.5^{**1022} = 2.2\text{e-}308$

Overflow goes by default to a signed ∞ .

Underflow is *Gradual*, rounding to the nearest integer multiple of $0.5^{**1074} = 4.9\text{e-}324$.

Zero is represented ambiguously as +0 or -0.

Its sign transforms correctly through multiplication or division, and is preserved by addition of zeros with like signs; but $x-x$ yields +0 for every finite x . The only operations that reveal zero's sign are division by zero and *copysign*($x, \pm 0$). In particular, comparison ($x > y$, $x \geq y$, etc.) cannot be affected by the sign of zero; but if finite $x = y$ then $\infty = 1/(x-y) \neq -1/(y-x) = -\infty$.

∞ is signed.

It persists when added to itself or to any finite number. Its sign transforms correctly through multiplication and division, and $(\text{finite})/\pm\infty = \pm 0$ (nonzero)/0 = $\pm\infty$. But $\infty-\infty$, $\infty*0$ and ∞/∞ are, like 0/0 and $\text{sqrt}(-3)$, invalid operations that produce *NaN*:

Reserved operands:

There are $2^{**53}-2$ of them, all called *NaN* (Not a Number). Some, called Signaling *NaNs*, trap any floating-point operation performed upon them; they are used to mark missing or uninitialized values, or nonexistent elements of arrays. The rest are Quiet *NaNs*; they are the default results of Invalid Operations, and propagate through subsequent arithmetic operations. If $x \neq x$ then x is *NaN*; every other predicate ($x > y$, $x = y$, $x < y$, ...) is FALSE if *NaN* is involved.

NOTE: Trichotomy is violated by *NaN*.

Besides being FALSE, predicates that entail ordered comparison, rather than mere (in)equality, signal Invalid Operation when *NaN* is involved.

Rounding:

Every algebraic operation (+, −, *, /, √) is rounded by default to within half an *ulp*, and when the rounding error is exactly half an *ulp* then the rounded value's least significant bit is zero. This kind of rounding is usually the best kind, sometimes provably so; for instance, for every $x = 1.0, 2.0, 3.0, 4.0, \dots, 2.0^{**52}$, we find $(x/3.0)*3.0 == x$ and $(x/10.0)*10.0 == x$ and ... despite that both the quotients and the products have been rounded. Only rounding like IEEE 754 can do that. But no single kind of rounding can be proved best for every circumstance, so IEEE 754 provides rounding towards zero or towards $+\infty$ or towards $-\infty$ at the programmer's option. And the same kinds of rounding are specified for Binary–Decimal Conversions, at least for magnitudes between roughly $1.0e-10$ and $1.0e37$.

Exceptions:

IEEE 754 recognizes five kinds of floating–point exceptions, listed below in declining order of probable importance.

Exception	Default Result
Invalid Operation	<i>NaN</i> , or FALSE
Overflow	$\pm\infty$
Divide by Zero	$\pm\infty$
Underflow	Gradual Underflow
Inexact	Rounded value

NOTE: An Exception is not an Error unless handled badly. What makes a class of exceptions exceptional is that no single default response can be satisfactory in every instance. On the other hand, if a default response will serve most instances satisfactorily, the unsatisfactory instances cannot justify aborting computation every time the exception occurs.

For each kind of floating–point exception, IEEE 754 provides a Flag that is raised each time its exception is signaled, and stays raised until the program resets it. Programs may also test, save and restore a flag. Thus, IEEE 754 provides three ways by which programs may cope with exceptions for which the default result might be unsatisfactory:

1. Test for a condition that might cause an exception later, and branch to avoid the exception.
2. Test a flag to see whether an exception has occurred since the program last reset its flag.
3. Test a result to see whether it is a value that only an exception could have produced.

CAUTION: The only reliable ways to discover whether Underflow has occurred are to test whether products or quotients lie closer to zero than the underflow threshold, or to test the Underflow flag. (Sums and differences cannot underflow in IEEE 754; if $x \neq y$ then $x-y$ is correct to full precision and certainly nonzero regardless of how tiny it may be.) Products and quotients

that underflow gradually can lose accuracy gradually without vanishing, so comparing them with zero (as one might on a VAX) will not reveal the loss. Fortunately, if a gradually underflowed value is destined to be added to something bigger than the underflow threshold, as is almost always the case, digits lost to gradual underflow will not be missed because they would have been rounded off anyway. So gradual underflows are usually *provably* ignorable. The same cannot be said of underflows flushed to 0.

At the option of an implementor conforming to IEEE 754, other ways to cope with exceptions may be provided:

- 4) **ABORT.** This mechanism classifies an exception in advance as an incident to be handled by means traditionally associated with error-handling statements like "ON ERROR GO TO ...". Different languages offer different forms of this statement, but most share the following characteristics:
 - No means is provided to substitute a value for the offending operation's result and resume computation from what may be the middle of an expression. An exceptional result is abandoned.
 - In a subprogram that lacks an error-handling statement, an exception causes the subprogram to abort within whatever program called it, and so on back up the chain of calling subprograms until an error-handling statement is encountered or the whole task is aborted and memory is dumped.
- 5) **STOP.** This mechanism, requiring an interactive debugging environment, is more for the programmer than the program. It classifies an exception in advance as a symptom of a programmer's error; the exception suspends execution as near as it can to the offending operation so that the programmer can look around to see how it happened. Quite often the first several exceptions turn out to be quite unexceptionable, so the programmer ought ideally to be able to resume execution after each one as if execution had not been stopped.
- 6) Other ways lie beyond the scope of this document.

The crucial problem for exception handling is the problem of Scope, and the problem's solution is understood, but not enough manpower was available to implement it fully in time to be distributed in 4.3 BSD's *libm*, upon which much of the INTERIX math library is based. Ideally, each elementary function should act as if it were indivisible, or atomic, in the sense that:

- No exception should be signaled that is not deserved by the data supplied to that function.
- Any exception signaled should be identified with that function rather than with one of its subroutines.
- The internal behavior of an atomic function should not be disrupted when a calling program changes from one to another of the five or so ways of handling exceptions listed above, although the definition of the function may be correlated intentionally with exception handling.

Ideally, every programmer should be able *conveniently* to turn a debugged subprogram into one that appears atomic to its users. But simulating all three

characteristics of an atomic function is still a tedious affair, entailing hosts of tests and saves–restores; work is under way to ameliorate the inconvenience.

Meanwhile, the functions in *libm* are only approximately atomic. They signal no inappropriate exception except possibly:

Over / Underflow

when a result, if properly computed, might have lain barely within range, and

Inexact in *cabs*, *cbrt*, *hypot*, *log10* and *pow*

when it happens to be exact, thanks to fortuitous cancellation of errors.

Otherwise:

Invalid Operation is signaled only when

any result but *NaN* would probably be misleading.

Overflow is signaled only when

the exact result would be finite but beyond the overflow threshold.

Divide–by–Zero is signaled only when

a function takes exactly infinite values at finite operands.

Underflow is signaled only when

the exact result would be nonzero but tinier than the underflow threshold.

Inexact is signaled only when

greater range or precision would be needed to represent the exact result.

BUGS

When signals are appropriate, they are emitted by certain operations within the codes, so a subroutine–trace may be needed to identify the function with its signal in case method 5) above is in use. And the codes all take the IEEE 754 defaults for granted; this means that a decision to trap all divisions by zero could disrupt a code that would otherwise get correct results despite division by zero.

SEE ALSO

An explanation of IEEE 754 and its proposed extension p854 was published in the IEEE magazine MICRO in August 1984 under the title "A Proposed Radix– and Word–length–independent Standard for Floating–point Arithmetic" by W. J. Cody et al. The manuals for Pascal, C and BASIC on the Apple Macintosh document the features of IEEE 754 pretty well. Articles in the IEEE magazine COMPUTER vol. 14 no. 3 (Mar. 1981), and in the ACM SIGNUM Newsletter Special Issue of Oct. 1979, may be helpful although they pertain to superseded drafts of the standard.

NAME

memcpy – copy string until character found

SYNOPSIS

```
#include <string.h>
```

```
void * memcpy (void *dst, const void *src, int c, size_t len)
```

DESCRIPTION

The *memcpy()* function copies bytes from string *src* to string *dst*. If the character *c* (as converted to an **unsigned char**) occurs in the string *src*, the copy stops and a pointer to the byte after the copy of *c* in the string *dst* is returned. Otherwise, *len* bytes are copied, and a NULL pointer is returned.

SEE ALSO

bcopy(), *memcpy()*, *memmove()*, *strcpy()*

NAME

memchr() – locate byte in byte string

SYNOPSIS

```
#include <string.h>
```

```
void * memchr (const void *b, int c, size_t len)
```

DESCRIPTION

The *memchr()* function converts *c* to an **unsigned char** and searches for *c* in the first *len* characters of string *b*.

RETURN VALUES

The *memchr()* function returns a pointer to the byte located, or NULL if no such byte exists within *len* bytes.

SEE ALSO

strchr(), *strcspn()*, *strpbrk()*, *strrchr()*, *strsep()*, *strspn()*, *strstr()*, *strtok()*

NAME

memcmp() – compare byte string

SYNOPSIS

```
#include <string.h>
```

```
int memcmp (const void *b1, const void *b2, size_t len)
```

DESCRIPTION

The *memcmp()* function compares the first *len* bytes of string *b1* against the first *len* bytes of string *b2*.

RETURN VALUES

The *memcmp()* function returns 0 if the the two strings are identical, otherwise returns the difference between the first two differing bytes (treated as unsigned char values, so that \200 is greater than \0, for example). Zero-length strings are always identical.

SEE ALSO

strcasecmp(), *strcmp()*, *strcoll()*, *strxfrm()*

NAME

memcpy() – copy byte string

SYNOPSIS

```
#include <string.h>
```

```
void * memcpy (void *dst, const void *src, size_t len)
```

DESCRIPTION

The *memcpy()* function copies *len* bytes from string *src* to string *dst*.

RETURN VALUES

The *memcpy()* function returns the original value of *dst*.

SEE ALSO

bcopy(), *memmove()*, *strcpy()*

NAME

memmove() – copy byte string

SYNOPSIS

```
#include <string.h>
```

```
void * memmove (void *dst, const void *src, size_t len)
```

DESCRIPTION

The *memmove()* function copies *len* bytes from string *src* to string *dst*. The two strings may overlap; the copy is always done in a non-destructive manner.

RETURN VALUES

The *memmove()* function returns the original value of *dst*.

SEE ALSO

bcopy(), *memcpy()*, *strcpy()*

NAME

memset() – write a byte to byte string

SYNOPSIS

```
#include <string.h>
```

```
void * memset (void *b, int c, size_t len)
```

DESCRIPTION

The *memset()* function writes *len* bytes of value *c* to the string *b*.

RETURNS

The *memset()* function returns its first argument, *b*.

NAME

mkdir() – make a directory file

SYNOPSIS

```
#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 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.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EPERM]	The <i>path</i> 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 file serial numbers on the file system on which the directory is being created.
[EIO]	An I/O error occurred while making the directory entry or allocating the file serial number.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EFAULT]	<i>Path</i> points outside the process's allocated address space.

SEE ALSO

chmod(), *stat()*, *umask()*

NAME

mkfifo() – make a fifo file

SYNOPSIS

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

int mkfifo (const char *path, mode_t mode)
```

DESCRIPTION

Mkfifo() creates a new fifo file with name *path*. The access permissions are specified by *mode* and restricted by the *umask()* of the calling process.

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

RETURN VALUES

mkfifo() returns 0 on success; it returns -1 for an error, and sets *errno*.

ERRORS

Mkfifo() will fail and no fifo will be created if:

[ENOTSUPP]	The kernel has not been configured to support fifo's.
[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 {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EPERM]	The <i>path</i> 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 directory in which the entry for the new fifo is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOSPC]	There are no free file serial numbers on the file system on which the fifo is being created.
[EIO]	An I/O error occurred while making the directory entry or allocating the file serial number.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EFAULT]	<i>Path</i> points outside the process's allocated address space.

SEE ALSO

chmod(), *stat()*, *umask()*

NAME

mktemp() – make temporary file name (unique)

SYNOPSIS

```
#include <stdlib.h>
```

```
char * mktemp (char *template)
```

```
int mkstemp (char *template)
```

DESCRIPTION

The *mktemp()* function takes the given file name template and overwrites a portion of it to create a file name. This file name is unique and suitable for use by the application. The template may be any file name with some number of *X*s appended to it, for example **/tmp/temp.XXXX**. The trailing *X*s are replaced with the current process number and/or a unique letter combination. The number of unique file names *mktemp()* can return depends on the number of *X*s provided; six *X*s will result in *mktemp()* testing roughly 26**6 combinations.

The *mkstemp()* function makes the same replacement to the template and creates the template file, mode 0600, returning a file descriptor opened for reading and writing. This avoids the race between testing for a file's existence and opening it for use.

RETURN VALUES

The *mktemp()* function returns a pointer to the template on success and NULL on failure. The *mkstemp()* function returns -1 if no suitable file could be created. If either call fails an error code is placed in the global variable *errno*.

ERRORS

The *mktemp()* and *mkstemp()* functions may set *errno* to one of the following values:

[ENOTDIR] The pathname portion of the template is not an existing directory.

The *mktemp()* and *mkstemp()* functions may also set *errno* to any value specified by the *stat()* function.

The *mkstemp()* function may also set *errno* to any value specified by the *open()* function.

SEE ALSO

chmod(), *getpid()*, *open()*, *stat()*

NAME

mktime() – transform format of time value

SYNOPSIS

```
#include <time.h>

time_t mktime (struct tm *tm)
```

DESCRIPTION

The function *mktime()* converts the local time, expressed in the structure pointed to by *tm* into a **time_t** time value with the same encoding as that of the values returned by the *time()* function, that is, seconds from the Epoch, UTC.

The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges. (A positive or zero value for *tm_isdst* causes *mktime()* to presume initially that summer time (for example, Daylight Saving Time) is or is not in effect for the specified time, respectively. A negative value for *tm_isdst* causes the *mktime()* function to attempt to divine whether summer time is in effect for the specified time.)

On successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to their normal ranges; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined. *Mktime()* returns the specified calendar time; if the calendar time cannot be represented, it returns -1.

External declarations as well as the **tm** structure definition are in the **<time.h>** include file. The **tm** structure includes at least the following fields:

int	<i>tm_sec;</i>	seconds (0 – 60)
int	<i>tm_min;</i>	minutes (0 – 59)
int	<i>tm_hour;</i>	hours (0 – 23)
int	<i>tm_mday;</i>	day of month (1 – 31)
int	<i>tm_mon;</i>	month of year (0 – 11)
int	<i>tm_year;</i>	year – 1900
int	<i>tm_wday;</i>	day of week (Sunday = 0)
int	<i>tm_yday;</i>	day of year (0 – 365)
int	<i>tm_isdst;</i>	is summer time in effect?

The field *tm_isdst* is non-zero if summer time is in effect.

SEE ALSO

date

asctime(), *ctime()*, *gmtime()*, *localtime()*, *difftime()*, *getenv()*, *time()*, *tzset()*

NAME

mmap() – map a file into memory

SYNOPSIS

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

void *mmap(void *addr, size_t len, int prot,
            int flags, int fd, off_t off);
void *munmap(void *addr, size_t len, int prot,
              int flag, int fd, off_t off)
```

DESCRIPTION

The *mmap()* function maps a file on disk into a buffer in memory so that when bytes are retrieved from the buffer, they're read from the file. The file has the open file descriptor *fd* and the function maps *len* bytes of the file. The portion of the file to be mapped to the buffer is *off* bytes from the beginning of the file. The function returns a pointer (*pa*) such that the range [*pa*, *pa* + *len*) is the same as the range [*fd*, *fd* + *len*).

addr The point in the address space where the mapping should be placed. Normally this is set to zero, which indicates that the system may place the mapping anywhere. If it's a positive non-zero value, then it is considered a hint to the system for placing the buffer. This can be changed by setting MAP_FIXED in the *flags* argument; if set, the function must return the value of *addr* as the address of the buffer. If it cannot, the function fails. The value of *addr* should be a multiple of the system's virtual memory page size, returned by *sysconf()* with the argument _SC_PAGESIZE (or _SC_PAGE_SIZE).

len The length of the mapped space, in bytes. It's rounded up to the nearest multiple of *sysconf(_SC_PAGE_SIZE)*.

prot The set of accesses permitted to the pages being mapped. Traditionally these should be the same as the open mode of the file (that is, you can't write to the memory if the file was opened read-only). These access values are defined in **<sys/mman.h>**:

PROT_READ

Read permission.

PROT_WRITE

Write permission.

PROT_EXEC

Execute permission.

PROT_NONE

No access allowed.

flags Other information about how the mapped file is to be handled. Possible values are:

MAP_SHARED

Changes can be shared; cannot be specified with MAP_PRIVATE. When the calling process writes to the memory region, the file may be changed. These changes are visible in all MAP_SHARED mappings of the same portion of the file, by any process.

MAP_PRIVATE

Changes are private; cannot be specified with MAP_SHARED. When the calling process writes to the memory region, the file isn't changed, they affect a copy of the file. No other process

mapping the same portion of the file can see these changes.

MAP_FIXED

The pointer returned by the function must be equal to *addr*. Normally, the returned pointer is derived from the *addr* but doesn't have to be identical. This is the least portable way of coding with the function.

- This information is retained across a *fork()*, but not across an *exec()*.
- fd* The file descriptor of the file to be mapped. The file should have already been opened.
- off* The offset to the part of the file where mapping should start. The value of *off* should be a multiple of the system's virtual memory page size, returned by *sysconf(_SC_PAGESIZE)*.

When the process terminates, the mapped region is unmapped. It does *not* unmap when *fd* is closed.

You can also explicitly unmap the region by calling the *unmap()* function.

RETURN VALUE

On success, *mmap()* returns the address where the mapping was placed. On failure, it returns *-1* and sets *errno* to indicate the error.

ERRORS

The *mmap()* function can fail if:

- | | |
|----------|--|
| [EBADF] | The file descriptor <i>fd</i> was not valid. |
| [EACCES] | The file indicated by <i>fd</i> wasn't open to be read; or <i>fd</i> wasn't open for writing and PROT_WRITE was specified for a MAP_SHARED mapping. |
| [ENXIO] | The addresses [<i>off</i> , <i>off+len</i>) were invalid for <i>fd</i> . |
| [EINVAL] | The <i>off</i> or the <i>addr</i> argument, if MAP_FIXED was specified, was not a multiple of the page size (as returned by <i>sysconf()</i>) or was otherwise invalid; or <i>flags</i> was invalid (neither MAP_PRIVATE nor MAP_SHARED was set). |
| [EMFILE] | Fulfilling the request would exceed the system limit on the number of mapped regions. |
| [ENODEV] | The <i>fd</i> refers to a file of an unsupported type. |
| [ENOMEM] | If MAP_FIXED was specified, then the range [<i>addr</i> , <i>addr+len</i>) is greater than allowed for a process' address space; if MAP_FIXED was not specified, there is insufficient room in the address space to perform the mapping. |

SEE ALSO

exec(), *fcntl()*, *fork()*, *munmap()*, *shmat()*, *sysconf()*

NAME

`modf()` – extract signed integral and fractional values from floating-point number

SYNOPSIS

```
#include <math.h>
```

```
double modf (double value, double *iptr)
```

DESCRIPTION

The `modf()` function breaks the argument *value* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a **double** in the object pointed to by *iptr*.

RETURN VALUES

The `modf()` function returns the signed fractional part of *value*.

SEE ALSO

`frexp()`, `ldexp()`

NAME

msgctl() – control message operations

SYNOPSIS

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

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

DESCRIPTION

The *msgctl()* function provides a way to control message queue operations. The arguments are:

msqid An identifier indicating a particular message queue, returned by *msgget()*. The queue has a data structure associated with it; the visible elements of the data structure can be stored in a structure of type **msqid_ds** (see the description of *buf*).

cmd The command to be carried out; *msgctl()* supports three commands (defined in **<sys/msg.h>**):

IPC_STAT Store the current data structure information in *buf*.

IPC_SET Set the values of the message queue's data structure to those specified in *buf*. Only the *uid*, *gid*, and *mode* members of **msqid_ds.msg_perm** can be modified. The process must have the appropriate privileges or an effective user ID equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with the queue.

IPC_RMID Remove *msqid* and the associated data structure from the system. The process must have the appropriate privileges or an effective user ID equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with the queue.

buf A pointer to a data structure of type **msqid_ds** which contains information about the message queue. It is defined in **<sys/msg.h>**; *msgctl()* only allows you to alter the following members:

struct ipc_perm	msg_perm	operation permission structure
msglen_t	msg_qbytes	max. # of bytes allowed on queue

See *msgget()* for a description of all of the visible members of **msqid_ds** and their default values. The type **msglen_t** is **typedefed** in **<sys/msg.h>**. The **ipc_perm** structure is defined in **<sys/ipc.h>**, which is automatically included when you include **<sys/msg.h>**.

RETURN VALUE

Msgctl() returns 0 on success; on failure, it returns -1 and sets *errno* to indicate the error.

ERRORS

The *msgctl()* function can fail for these reasons:

[EACCES]	The <i>cmd</i> is IPC_STAT but the calling process doesn't have read permission.
[EINVAL]	Either <i>msqid</i> or <i>cmd</i> isn't valid.
[ENOSPC]	Increasing the message queue would exceed system resources. The message queue will not be damaged if this error is returned.
[EPERM]	If the <i>cmd</i> is IPC_RMID or IPC_SET, the process' effective user ID (which is the same as the user ID) is not equal to the value of msg_perm.cuid or

msg_perm.uid in the data structure for the specified *msqid*.

SEE ALSO

msgget(), *msgrcv()*, *msgsnd()*

NAME

msgget() – get message queue

SYNOPSIS

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

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

DESCRIPTION

The *msgget()* function returns a message queue identifier associated with *key*.

The *key* can be an agreed-upon integer or it can have the value `IPC_PRIVATE`.

The *msgflg* controls the mode and permissions of the message queue. The following bits (defined in `<sys/ipc.h>`) can be OR'ed into *msgflg*:

`IPC_CREAT` Create the entry if the key doesn't exist.

`IPC_EXCL` Fail if the key already exists and its creation was requested.

`IPC_NOWAIT`

Return an error if the request must wait. This request controls the behavior of *msgrcv()* and *msgsnd()*, while the `IPC_CREAT` and `IPC_EXCL` requests control how an IPC handle is obtained.

If the *key* already has a message queue identifier associated with it, then *msgget()* returns that value.

The *msgget()* function creates the message queue if:

- The *key* is `IPC_PRIVATE`; or
- the *key* doesn't already have a message queue identifier associated with it and (*msgflg* & `IPC_CREAT`) is non-zero.

When a message queue is created, the data structure associated with the identifier is initialized as follows:

Member	Meaning	Initialized
<code>msg_perm.uid</code>	effective user ID	of calling process
<code>msg_perm.gid</code>	effective group ID	of calling process
<code>msg_perm.cuid</code>	effective user ID	of calling process
<code>msg_perm.cgid</code>	effective group ID	of calling process
<code>msg_perm.mode</code>	Permissions	low-order 9 bits are set to low-order 9 bits of <i>msgflg</i>
<code>msg_ctime</code>	time of last change	current time
<code>msg_lrpid</code>	process ID of last <i>msgrcv()</i>	0
<code>msg_lspid</code>	process ID of last <i>msgsnd()</i>	0
<code>msg_qbytes</code>	max num. of bytes allowed on queue	system default
<code>msg_qnum</code>	number of messages on queue	0
<code>msg_rtime</code>	Time of last <i>msgrcv()</i>	0
<code>msg_stime</code>	Time of last <i>msgsnd()</i>	0

Note that there is no distinction between user and effective user IDs. The limit for *msg_qbytes* is available memory.

The header file **<sys/ipc.h>** is automatically included when you include the file **<sys/msg.h>**.

RETURN VALUE

On success, *msgget()* returns a message queue identifier, a non-negative integer . On failure, it returns **-1** and sets *errno* to indicate the error.

ERRORS

The *msgget()* function can fail for the following reasons:

- [EACCES] A message queue identifier already exists for *key*, but permissions cannot be granted.
- [EEXIST] A message queue identifier already exists for *key*, but
 ((*msgflg*&IPC_CREAT) && (*msgflg*&IPC_EXCL)) is non-zero.
- [ENOENT] No message queue identifier exists for *key* and creation of one was not requested (that is, *msgflg*&IPC_CREAT is zero).
- [ENOSPC] Creating the message queue identifier would exceed the system-wide limit on the number of message queue identifiers.

SEE ALSO

msgctl(), *msgrcv()*, *msgsnd()*

NAME

msgrcv() – read message from a message queue

SYNOPSIS

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

int msgrcv (int msqid, void *msgp, size_t msgsz,
            long int msgtyp, int msgflg);
```

DESCRIPTION

Msgrcv() retrieves a message from the queue associated with the identifier *msqid*. The message is stored in a user-defined structure indicated by *msgp* (see *msgsnd()*) and is of the size specified by *msgsz*.

The type of message to be received is specified by the argument *msgtyp*. The *msgrcv()* function searches through the queue for messages of that type; if *msgtyp* is 0, the function returns the first message in the queue. If *msgtyp* is <0, the function returns the first message whose type is less than or equal to the absolute value of *msgtyp*.

The *msgflg* describes what is to be done if there is no message of the appropriate type in the queue:

(*msgflg*&IPC_NOWAIT) != 0

The calling process does not wait for a message but returns immediately; the function returns a value of -1 and sets *errno* to [ENOMSG].

(*msgflg*&IPC_NOWAIT) = 0

The calling process suspends execution until:

- A message of the right type appears in the queue.
- The message queue is removed from the system; the function returns -1 and sets *errno* to [EIDRM].
- The calling process catches a signal.

Once the function has received a message, the function changes the data structure associated with *msqid* in these ways:

- **msg_qnum** is reduced by 1.
- **msg_lrp** is set to the process ID of the calling process.
- **msg_rtime** is set to the current time.

RETURN VALUE

On success, the *msgrcv()* function returns the number of bytes placed into the data buffer of the *msgp* structure. On failure it returns -1 and sets *errno* to indicate the error.

ERRORS

[E2BIG]	The message text is larger than <i>msgsz</i> and the MSG_NOERROR flag has not been set (that is, (<i>msgflg</i> &MSG_NOERROR) is zero).
[EACCES]	The calling process does not have permission to perform that operation.
[EIDRM]	The message queue identifier <i>msqid</i> has been removed from the system.

- | | |
|----------|---|
| [EINTR] | The function was interrupted by a signal. |
| [EINVAL] | The <i>msqid</i> is not a valid message queue identifier or <i>msgsz</i> is less than 0. |
| [ENOMSG] | There is no message of the desired type in the queue and the IPC_NOWAIT bit is set (that is, (<i>msgflg</i> &IPC_NOWAIT) is non-zero). |

SEE ALSO

msgctl(), *msgget()*, *msgsnd()*

NAME

msgsnd() – send message to message queue

SYNOPSIS

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

```
int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg)
```

DESCRIPTION

The *msgsnd()* function sends a message to the queue associated with the identifier *msqid*. The message is held in the data structure pointed to by *msgp*; the size of the message in bytes is indicated by *msgsz*, and can be any value from 0 to the system limit. The *msgflg* argument specifies what is to be done if the queue already holds the maximum number of bytes or if the total number of messages on all queues on the system meets the system-imposed limit. The important instruction for *msgflg* is *IPC_NOWAIT*.

When the *msgsnd()* function succeeds, the data structure associated with *msqid* is modified:

- **msg_qnum** is increased by 1
- **msg_lspid** is set to the process ID of the calling process
- **msg_stime** is set to the current time.

The *msgp* structure is defined by the user, but the first member must be a non-zero positive **long int** which holds a value used for selecting messages. The second member must hold the data text. The length must be the same as *msgsz*.

If the message queue is full, or if the system-limit on queued messages is reached, the behavior of *msgsnd()* depends upon the value of *msgflg*:

- If (*msgflg*&*IPC_NOWAIT*) is non-zero, the message isn't sent and the calling process returns immediately.
- If (*msgflg*&*IPC_NOWAIT*) is zero, the calling process suspends execution until one of:
 - the condition is relieved (messages are removed from the message queue, the total number of queued messages shrinks), and the message is sent;
 - the identifier for this message queue is removed from the system, and *msgsnd()* returns -1 and sets *errno* to [EIDRM].
 - The calling process is interrupted by a signal that is to be caught, in which case the signal is handled and the message is not sent.

RETURN VALUE

The *msgsnd()* function returns 0 on success. On failure, it returns -1, doesn't send the message, and sets *errno* to indicate the error.

ERRORS

- | | |
|----------|--|
| [EACCES] | The calling process does not have permission to perform that operation. |
| [EAGAIN] | The operation failed for some reason and The non-wait bit is set (that is, (<i>msgflg</i> & <i>IPC_NOWAIT</i>) is non-zero). |

[EIDRM]	The message queue identifier <i>msqid</i> has been removed from the system.
[EINTR]	The function was interrupted by a signal.
[EINVAL]	The <i>msqid</i> is not a valid message queue identifier, or the message size value is negative or too large, or the value of the message type is less than 1.

EXAMPLES

A simple structure for data which sends four-character strings as messages might be:

```
struct a_msg {  
    long int  msg_type;      message type  
    char      msg_text[5];  message text  
};
```

For this structure, the value of *msgsz* would be 5.

SEE ALSO

msgctl(), *msgget()*, *msgrcv()*

NAME

`munmap()` – remove mappings for files in memory

SYNOPSIS

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

void *munmap (void *addr, size_t len)
```

DESCRIPTION

The `munmap()` function removes a mapping for a file from the range `[addr, addr + len)`.

`Addr` should be a multiple of the page size as given by `sysconf()`. The `len` argument is rounded up to the next multiple of the page size.

Mapped files are automatically unmapped when the process terminates.

RETURN VALUE

On success, `munmap()` returns 0. On failure, it returns `-1` and sets `errno` to indicate the error.

ERRORS

The `munmap()` function can fail if:

- | | |
|----------|---|
| [EINVAL] | The <code>addr</code> is not a multiple of the page size. |
| | Addresses in the range <code>[addr, addr + len)</code> are not in the valid range for a process' address space. |
| | <code>Len</code> is 0. |

SEE ALSO

`exec()`, `fcntl()`, `fork()`, `mmap()`, `shmat()`, `sysconf()`

NAME

nftw() – walk a file tree

SYNOPSIS

```
#include <ftw.h>

struct FTW {
    int base;
    int level;
};

int nftw (const char *path, int (*fn)(const char *,
    const struct stat *, int, struct FTW *), int depth,
    int flag)
```

DESCRIPTION

The *nftw()* call recursively descends the directory hierarchy rooted in *path*, much like *ftw()*. However, *nftw()* takes an additional argument *flag*, which is a bitwise inclusive OR of zero or more of these flags:

FTW_CHDIR

Changes the current working directory to each directory as it reports the files in that directory. If not set, *nftw()* does not change the current working directory.

FTW_DEPTH

Perform a depth-wise search. If this is set, *nftw()* reports files in a directory before reporting the directory itself. Otherwise, *nftw()* reports the directory before reporting the files in that directory (the same behavior as *ftw()*).

FTW_MOUNT

Only report files in the same filesystem as *path*. If this bit is clear, *nftw()* reports all files encountered in the walk.

FTW_PHYS

Do a physical walk, ignoring symbolic links. If this bit is clear, *nftw()* follows symbolic links instead of reporting them, and does not report the same file twice.

The *depth* argument is similar to the *ftw()* *ndirs* argument; it sets the maximum number of file descriptors used by *nftw()* while traversing the tree. At most one is used per directory level. If *depth* is 0, limitless recursion is allowed.

For each object it encounters, *nftw()* calls the function pointed to by *fn* with four arguments:

- The object's pathname.
- A pointer to the *stat* buffer, containing information on the object.
- An integer containing other information. The integer is one of:

FTW_D The object is a directory.

FTW_DNR

The object is a directory that cannot be read. In this case, children of the directory are not read.

FTW_DP

The object is a directory and subdirectories have been visited. (This is only reported if the **FTW_DEPTH** bit is set in the *flag*.)

FTW_F The object is a file.

FTW_NS

The object was not successfully *stat()*'ed. In this case, the **stat** function will contain undefined values. For example, a file with read but not execute/search permission can cause this value.

FTW_SL

The object is a symbolic link. (This is only reported if the FTW_PHYS bit is set in the *flag*.) Since symbolic links are not supported under INTERIX, this is unlikely to occur.

FTW_SLN

The object is a symbolic link and does not point to an existing file. (This is only reported if the FTW_PHYS bit is *not* set in the *flag*.) Since symbolic links are not supported under INTERIX, this is unlikely to occur.

- A pointer to an **FTW** structure: The **base** value is the offset of the object's filename in the pathname passed as the first argument to *fn()*. The **level** indicates the depth relative to the root of the walk; the root level is 0. For example, if *path* is "/", the file "/a/b/c" has a base (offset) of 5 and a level of 3.

The *nftw()* function uses at most one file descriptor for each level in the tree.

RETURN VALUES

The *ftw()* function returns 0 when the tree is exhausted. If the function pointed to by *fn* returns a non-zero value or an error (other than [EACCES]), the *ftw()* function stops and returns that value. If *ftw()* detects an error, it returns -1 and sets *errno*.

ERROR VALUES

The *ftw()* function may fail for the following reasons:

[EACCES]	The process does not have permission to read or search a component of <i>path</i> .
[EINVAL]	The <i>ndirs</i> argument is invalid.
[ELOOP]	There were too many symbolic links. Since symbolic links are not supported on INTERIX, this error should not occur.
[ENAMETOOLONG]	The <i>path</i> is more than {PATH_MAX} characters long, or a path component is more than {NAME_MAX} characters long.
[ENOENT]	Either a component of <i>path</i> doesn't exist, or <i>path</i> is an empty string.
[ENOMEM]	There is not enough memory.
[ENOTDIR]	A component of <i>path</i> is not a directory.

SEE ALSO

ftw(), *opendir()*, *readdir()*, *stat()*

NAME

_ntpath2posix() – convert Win32-style pathname to POSIX format

SYNOPSIS

```
#include <opennt/opennt.h>
```

```
_ntpath2posix(char *source, char *dest)
```

DESCRIPTION

The *_ntpath2posix()* routine converts a Win32-style pathname (such as “c:\Interix”) into a POSIX-style pathname (such as “//C/Interix”). The drive specifier is converted to upper case, prefixed by //, and all backslashes are converted to slashes.

The routine converts the Win32 pathname pointed to by *source* and stores it in *dest*. Other than that, the routine does no significant error checking; source strings such as “:\users” become “:/users”.

If *dest* is NULL, the path is converted in place and the result is stored in *source*. Remember that a fully-qualified POSIX-style pathname will always be one character longer than the equivalent Win32-style pathname.

NOTES

The routine has no idea about current working directories on other drives, and interprets all pathnames with drive specifiers as rooted pathnames. For example, if the current working directory on drive D: is **D:\WIN32APP**, then the Win32 pathname **D:ACROBAT3** normally refers to the directory **D:\WIN32APP\ACROBAT3**. The *_ntpath2posix()* routine converts **D:ACROBAT3** into **//D/ACROBAT3**.

RETURN VALUES

The *_ntpath2posix()* routine returns a pointer to the converted pathname.

SEE ALSO

chgpath

NAME

`open()` – open or create a file for reading or writing

SYNOPSIS

```
#include <fcntl.h>
```

```
int open (const char *path, int flags, ...)
```

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), and modified by the process' `umask()` value (see `umask()`).

The flags specified are formed by OR'ing the following values

Symbol	Meaning
<code>O_APPEND</code>	append on each write
<code>O_CREAT</code>	create file if it does not exist
<code>O_EXCL</code>	error if create and file exists
<code>O_NOCTTY</code>	do not acquire as a controlling terminal
<code>O_NONBLOCK</code>	do not block on open
<code>O_RDONLY</code>	open for reading only
<code>O_RDWR</code>	open for reading and writing
<code>O_WRONLY</code>	open for writing only
<code>O_TRUNC</code>	truncate size to 0

When the file is opened with the `O_CREAT` flag set and it doesn't already exist, `open()` requires a third argument, which it treats as a *mode* of type **mode_t**. This argument specifies the mode of the newly-created file. (For a description of a *mode* see the reference page for `chmod()`.)

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).

If successful, `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 `execve` system calls; see `close()` and `fcntl()`.

The system imposes a limit on the number of file descriptors open simultaneously by one process.

RETURN VALUES

On success, the *open()* call returns an **integer** file descriptor which refers to the file. On failure, the call returns **-1** and sets *errno*.

ERRORS

The named file is opened unless:

[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.
[EEXIST]	O_CREAT and O_EXCL were specified and the file exists.
[EFAULT]	<i>Path</i> points outside the process's allocated address space.
[EINTR]	The <i>open()</i> operation was interrupted by a signal.
[EIO]	An I/O error occurred while making the directory entry or allocating the file serial number for O_CREAT.
[EISDIR]	The named file is a directory, and the arguments specify it is to be opened for writing.
[EMFILE]	The process has already reached its limit for open file descriptors.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
[ENFILE]	The system has already reached its limit for open file descriptors.
[ENOENT]	O_CREAT is not set and the named file does not exist.
[ENOENT]	A component of the pathname that must exist does not exist.
[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 file serial numbers on the file system on which the file is being created.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[EROFS]	The named file resides on a read-only file system, and the file is to be modified.

SEE ALSO

chmod(), *close()*, *creat()*, *dup()*, *fcntl()*, *lseek()*, *read()*, *write()*, *umask()*

NAME

pathconf(), fpathconf() – get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>

long pathconf (const char *path, int name)
long fpathconf (int fd, int name)
```

DESCRIPTION

The *pathconf()* and *fpconf()* functions provides a method for applications to determine the current value of a configurable system limit or option variable associated with a pathname or file descriptor.

For *pathconf()*, the *path* argument is the name of a file or directory. For *fpconf()*, the *fd* argument is an open file descriptor. The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file **<unistd.h>**.

The available values are as follows:

- _PC_LINK_MAX**
The maximum file link count.
- _PC_MAX_CANON**
The maximum number of bytes in terminal canonical input line.
- _PC_MAX_INPUT**
The minimum maximum number of bytes for which space is available in a terminal input queue.
- _PC_NAME_MAX**
The maximum number of bytes in a file name.
- _PC_PATH_MAX**
The maximum number of bytes in a pathname.
- _PC_PIPE_BUF**
The maximum number of bytes which will be written atomically to a pipe.
- _PC_CHOWN_RESTRICTED**
Return 1 if appropriate privileges are required for the *chown()* system call, otherwise 0.
- _PC_NO_TRUNC**
Return 1 if file names longer than {KERN_NAME_MAX} are truncated.
- _PC_VDISABLE**
Returns the terminal character disabling value.

RETURN VALUES

If the call to *pathconf()* or *fpconf()* is not successful, -1 is returned and *errno* is set appropriately. Otherwise, if the variable is associated with functionality that does not have a limit in the system, -1 is returned and *errno* is not modified. Otherwise, the current variable value is returned.

ERRORS

If any of the following conditions occur, the *pathconf()* and *fpathconf()* functions shall return `-1` and set *errno* to the corresponding value.

[EINVAL] The value of the *name* argument is invalid.

[EINVAL] The implementation does not support an association of the variable name with the associated file.

Pathconf() will fail if:

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

[ENAMETOOLONG]

A component of a pathname exceeded `{NAME_MAX}` characters, or an entire pathname exceeded `{PATH_MAX}` characters.

[ENOENT] The named file does not exist.

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

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

Fpathconf() will fail if:

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

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

NAME

pause() – stop until signal

SYNOPSIS

```
#include <unistd.h>

int pause (void)
```

DESCRIPTION

The *pause()* function forces a process to pause until a signal is received from either the *kill()* function or an interval timer. (See *alarm()*.) Upon termination of a signal handler started during a *pause()*, the *pause()* call will return.

RETURN VALUES

Always returns -1.

ERRORS

The *pause()* function always returns:

[EINTR] The call was interrupted.

SEE ALSO

alarm(), *kill()*, *select()*, *sigsuspend()*

NAME

pipe() – create descriptor pair for interprocess communication

SYNOPSIS

```
#include <unistd.h>

int pipe (int *fd)
```

DESCRIPTION

The *pipe()* function creates a *pipe*, which is an object allowing unidirectional data flow, and allocates a pair of file descriptors. The first descriptor connects to the *read end* of the pipe, and the second connects to the *write end*, so that data written to *fd[1]* appears on (i.e., can be read from) *fd[0]*. This allows the output of one program to be sent to another program: the source's standard output is set up to be the write end of the pipe, and the sink's standard input is set up to be the read end of the pipe. The pipe itself persists until all its associated descriptors are closed.

A pipe whose read or write end has been closed is considered *widowed*. Writing on such a pipe causes the writing process to receive a SIGPIPE signal. Widowing a pipe is the only way to deliver end-of-file to a reader: after the reader consumes any buffered data, reading a widowed pipe returns a zero count.

RETURN VALUES

On successful creation of the pipe, zero is returned. Otherwise, a value of *-1* is returned and the variable *errno* set to indicate the error.

ERRORS

The *pipe()* call will fail if:

- | | |
|----------|--|
| [EMFILE] | Too many descriptors are active. |
| [ENFILE] | The system file table is full. |
| [EFAULT] | The <i>fd</i> buffer is in an invalid area of the process's address space. |

SEE ALSO

sh
read(), *write()*, *fork()*

NAME

popen(), pclose() – process I/O with pipes

SYNOPSIS

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

```
FILE * popen (const char *command, const char *type)  
int pclose (FILE *stream)
```

DESCRIPTION

The *popen()* function “opens” a process by creating a pipe, forking, and invoking the shell. Since a pipe is by definition unidirectional, the *type* argument may specify only reading or writing, not both; the resulting stream is correspondingly read-only or write-only.

The *command* argument is a pointer to a null-terminated string containing a shell command line. This command is passed to **\$OPENNT_ROOT/bin/sh** using the **-c** flag; interpretation, if any, is performed by the shell. The *mode* argument is a pointer to a null-terminated string which must be either “r” for reading or “w” for writing.

The return value from *popen()* is a normal standard I/O stream in all respects save that it must be closed with *pclose()* rather than *fclose()*. Writing to such a stream writes to the standard input of the command; the command’s standard output is the same as that of the process that called *popen()* unless this is altered by the command itself. Conversely, reading from a “popened” stream reads the command’s standard output, and the command’s standard input is the same as that of the process that called *popen()*.

Note that output *popen()* streams are fully buffered by default.

The *pclose()* function waits for the associated process to terminate and returns the exit status of the command.

RETURN VALUE

The *popen()* function returns NULL if the *fork()* or *pipe()* calls fail, or if it cannot allocate memory.

On success, the *pclose()* function the exit status of the terminating command. On failure, *pclose()* returns -1 if *stream* is not associated with a *popen()*’ed command, or if *stream* already *pclose()*’d

ERRORS

The *popen()* function does not reliably set *errno*.

SEE ALSO

sh

fork(), *pipe()*, *fflush()*, *fclose()*, *fopen()*, *system()*

NAME

printf(), fprintf(), sprintf(), snprintf(), vprintf(), vfprintf(), vsprintf(), vsnprintf() – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf (const char *format ...)
int fprintf (FILE *stream, const char *format ...)
int sprintf (char *str, const char *format ...)
int snprintf (char *str, size_t size, const char *format ...)

#include <stdarg.h>

int vprintf (const char *format, va_list ap)
int vfprintf (FILE *stream, const char *format, va_list ap)
int vsprintf (char *str, const char *format, va_list ap)
int vsnprintf (char *str, size_t size, const char *format, va_list ap)
```

DESCRIPTION

The functions in the *printf()* family produce output according to a *format*, as described below. *Printf()* and *vprintf()* write output to stdout, the standard output stream; *fprintf()* and *vfprintf()* write output to the specified output *stream*; *sprintf()*, *snprintf()*, *vsprintf()*, and *vsnprintf()* write to the character string *str*.

These functions return the number of characters printed (not including the trailing \0 used to end output to strings). *Sprintf()* and *vsprintf()* effectively assume an infinite *size*.

Snprintf() and *vsnprintf()* write at most *size*–1 of the characters printed into the output string (the *size*'th character then get the terminating \0); if the return value is greater than or equal to the *size*, the string was too short and some of the printed characters were discarded. *Sprintf()* and *vsprintf()* effectively assume an infinite *size*.

The output is written according to the *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of <stdarg.h>) are converted for output.

The format string contains zero or more directives; a directive is either an ordinary character (any character but %), which is copied unchanged to the output stream. The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; or a conversion specification. A conversion specification fetches zero or more subsequent arguments. Each conversion specification is introduced by the character %. The arguments must correspond properly (after type promotion) with the conversion specifier. The syntax of a conversion specifier is:

```
%[#][0][-][ ][+][width][.[precision]][h|l|L]type
```

Only the *type* is required.

In sequence, these flags are:

- # A # character specifies that the value should be converted to an “alternate form”. This affects only o, x, X, e, E, f, g, and G conversions.
 - For o conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero).

- For x and X conversions, a non-zero result has the string 0x (or 0X for X conversions) prepended to it.
 - For e, E, f, conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows).
 - For g and G conversions, the result will always contain a decimal point, even if no digits follow it, and trailing zeros are not removed from the result as they would otherwise be.
- 0 A zero character specifies that padding on the left is done with zeros rather than blanks. For all conversions except n, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (d, i, o, u, i, x, and X), the 0 flag is ignored.
- A negative field width flag – indicates the converted value is to be left adjusted on the field boundary. Except for n conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A – overrides a 0 if both are given.
- A space specifies that a blank should be left before a positive number produced by a signed conversion (d, e, E, f, g, G, or i).
- + A + character specifies that a sign always be placed before a number produced by a signed conversion. A + overrides a space if both are used.
- width* This optional decimal digit string specifies a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width.
- A field width may be indicated by an asterisk * instead of a digit string. In this case, an **int** argument supplies the field width. A negative field width is treated as a left adjustment flag followed by a positive field width.
- precision* An optional precision, in the form of a period . followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for d, i, o, u, x, and X conversions, the number of digits to appear after the decimal-point for e, E, and f conversions, the maximum number of significant digits for g and G conversions, or the maximum number of characters to be printed from a string for s conversions.
- A precision may be indicated by an asterisk * instead of a digit string. In this case, an **int** argument supplies the precision. A negative precision is treated as though it were missing.
- h | l | L One of the optional characters h, l (ell), or L, depending on the conversion *type*. The values are:
- h a following d, i, o, u, x, or X conversion corresponds to a **short int** or **unsigned short int** argument, or that a following n conversion corresponds to a pointer to a **short int** argument.
- l (ell) A following d, i, o, u, x, or X conversion applies to a pointer to a **long int** or **unsigned long int** argument, or that a following n conversion corresponds to a pointer to a **long int** argument.
- L A following e, E, f, g, or G conversion corresponds to a **long double** argument.
- q A following d, i, o, u, x, or X conversion applies to a pointer to a

quad_t or **u_quad_t** (**unsigned long int**) argument, or that a following *n* conversion corresponds to a pointer to a **quad_t** argument. The types **quad_t** (extended precisions) and **u_quad_t** (unsigned extended precision) are defined in `<sys/types.h>`. Two *l* (ell) conversion option characters are equivalent to a single *q*.

type A character that specifies the type of conversion to be applied.

Type	Argument	Conversion
%		Prints literal % character
c	int	Prints a character
d	int	Signed decimal
D	long int	Signed decimal; equivalent to ld
e	double	Decimal in form [-]d.ddde[mdd]
E	double	Decimal in form [-]d.dddE[mdd]
f	double	Decimal in form [-]ddd.ddd
g	double	The more compact of f or e format
i	int	Signed decimal
n	int *	Number of character printed so far stored in argument
o	int	Unsigned octal
O	long int	Unsigned octal; equivalent to lo
p	void *	Hexadecimal value of pointer, as for %#x or %#lx
s	char *	String
u	int	Unsigned decimal
U	long int	Unsigned decimal; equivalent to lu
X	int	Unsigned hexadecimal using letters ABCDEF
x	int	Unsigned hexadecimal using letters abcdef

Note:

- f The number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.
- g The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style e is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.
- s The **char *** argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating NUL character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NUL character.
- n The number of characters written so far is stored into the integer indicated by the **int *** (or variant) pointer argument. No argument is converted.
- % A % is written. No argument is converted. The complete conversion specification is %%.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

EXAMPLES

To print a date and time in the form 'Sunday, July 3, 10:02', where *weekday* and *month* are pointers to strings:

```
#include <stdio.h>
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
        weekday, month, day, hour, min);
```

To print to five decimal places:

```
#include <math.h>
#include <stdio.h>
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

To allocate a 128-byte string and print into it:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
char *nefmt(const char *fmt, ...)
{
    char *p;
    va_list ap;

    if ((p = malloc(128)) == NULL)
        return (NULL);
    va_start(ap, fmt);
    (void) vsnprintf(p, 128, fmt, ap);
    va_end(ap);
    return (p);
}
```

SEE ALSO

printf

scanf()

NAME

ptsname() – get name of slave pseudo-terminal device

SYNOPSIS

```
#include <stdlib.h>

char * ptsname(int fildes)
```

DESCRIPTION

The *ptsname()* call returns the name of a slave pty that corresponds the master pseudo-terminal device indicated by *fildes*. This call is normally used after opening **/dev/*ptmx***.

The format of the slave name is **/dev/*tty*[*pqrs*]**0–9a–f****.

RETURN VALUE

On success, *ptsname()* returns a pointer to a string which is the name of the slave pseudo terminal. On failure, it returns NULL. (It fails if *fildes* can't be associated with the master side of a valid pseudo terminal.)

SEE ALSO

grantpt(), *open()*, *ttyname()*, *unlockpt()*
pty, *termios*, *tty*

NAME

fputc(), *putc()*, *putchar()* – output a character or word to a stream

SYNOPSIS

```
#include <stdio.h>

int fputc (int c, FILE *stream)
int putc (int c, FILE *stream)
int putchar (int c)
```

DESCRIPTION

The *fputc()* function writes the character *c* (converted to an **unsigned char**) to the output stream pointed to by *stream*.

Putc() acts essentially identically to *fputc()*, but is a macro that expands in-line. It may evaluate *stream* more than once, so arguments given to *putc()* should not be expressions with potential side effects.

Putchar() is identical to *putc()* with an output stream of *stdout*.

The *putw()* function writes the specified **int** to the named output *stream*.

RETURN VALUES

The functions, *fputc()*, *putc()* and *putchar()* return the character written. If an error occurs, the value EOF is returned.

SEE ALSO

feof(), *ferror()*, *fopen()*, *getc()*

NAME

qsort() – sort function

SYNOPSIS

```
#include <stdlib.h>

void qsort (void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *))
```

DESCRIPTION

The *qsort()* function is a modified partition-exchange sort, or quicksort.

The *qsort()* function sorts an array of *nmemb* objects, the initial member of which is pointed to by *base*. The size of each object is specified by *size*.

The contents of the array *base* are sorted in ascending order according to a comparison function pointed to by *compar*, which requires two arguments pointing to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

Qsort() is *not* stable, that is, if two members compare as equal, their order in the sorted array is undefined. While *qsort()* does not allocate memory, it is implemented using recursion.

RETURN VALUES

The *qsort()* function returns no value.

SEE ALSO

sort

NAME

radixsort() – radix sort

SYNOPSIS

```
#include <limits.h>
#include <stdlib.h>

int radixsort (u_char **base, int nmemb, u_char *table,
               u_int endbyte)
int sradixsort (u_char **base, int nmemb, u_char *table,
               u_int endbyte)
```

DESCRIPTION

The *radixsort()* and *sradixsort()* functions are implementations of radix sort.

These functions sort an array of pointers to byte strings, the initial member of which is referenced by *base*. The byte strings may contain any values; the end of each string is denoted by the user-specified value *endbyte*.

Applications may specify a sort order by providing the *table* argument. If *table* is not NULL, it must reference an array of {UCHAR_MAX}+1 bytes which contains the sort weight of each possible byte value. The end-of-string byte must have a sort weight of 0 or 255 (for sorting in reverse order). More than one byte may have the same sort weight. The *table* argument is useful for applications which wish to sort different characters equally, for example, providing a table with the same weights for A-Z as for a-z will result in a case-insensitive sort. If *table* is NULL, the contents of the array are sorted in ascending order according to the ASCII order of the byte strings they reference and *endbyte* has a sorting weight of 0.

The *sradixsort()* function is stable, that is, if two elements compare as equal, their order in the sorted array is unchanged. The *sradixsort()* function uses additional memory sufficient to hold *nmemb* pointers.

The *radixsort()* function is not stable, but uses no additional memory.

These functions are variants of most-significant-byte radix sorting; in particular, see D.E. Knuth's Algorithm R and section 5.2.5, exercise 10. They take linear time relative to the number of bytes in the strings.

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The value of the *endbyte* element of *table* is not 0 or 255.

Additionally, the *sradixsort()* function may fail and set *errno* for any of the errors specified for the library routine *malloc()*.

SEE ALSO

sort

qsort()

NAME

raise() – send a signal to the current process

SYNOPSIS

```
#include <signal.h>

int raise (int signum)
```

DESCRIPTION

The *raise()* function sends the signal *signum* to the current process.

This is equivalent to

```
kill(getpid(), signum)
```

RETURN VALUES

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

ERRORS

The *raise()* function may fail and set *errno* for any of the errors specified for the library functions *getpid()* and *kill()*.

SEE ALSO

kill()

NAME

drand48(), erand48(), lrand48(), nrand48(), mrand48(), jrand48(), srand48(), seed48(),
lcong48() – pseudo random number generators and initialization routines

SYNOPSIS

```
#include <stdlib.h>

double drand48 (void)
double erand48 (unsigned short xseed[3])
long lrand48 (void)
long nrand48 (unsigned short xseed[3])
long mrand48 (void)
long jrand48 (unsigned short xseed[3])
void srand48 (long seed)
unsigned short * seed48 (unsigned short xseed[3])
void lcong48 (unsigned short p[7])
```

DESCRIPTION

The *rand48()* family of functions generates pseudo-random numbers using a linear congruential algorithm working on integers 48 bits in size. The particular formula employed is

$$r(n+1) = (a * r(n) + c) \bmod m$$

where the default values are for the multiplicand $a = 0\text{xfdeec66d} = 25214903917$ and the addend $c = 0\text{xb} = 11$. The modulo is always fixed at $m = 2^{**}48$. $r(n)$ is called the seed of the random number generator.

For all the six generator routines described next, the first computational step is to perform a single iteration of the algorithm.

drand48() and *erand48()* return values of type **double**. The full 48 bits of $r(n+1)$ are loaded into the mantissa of the returned value, with the exponent set such that the values produced lie in the interval $[0.0, 1.0)$.

lrand48() and *nrand48()* return values of type **long** in the range $[0, 2^{**}31-1]$. The high-order (31) bits of $r(n+1)$ are loaded into the lower bits of the returned value, with the topmost (sign) bit set to zero.

mrnd48() and *jrand48()* return values of type **long** in the range $[-2^{**}31, 2^{**}31-1]$. The high-order (32) bits of $r(n+1)$ are loaded into the returned value.

drand48(), *lrand48()*, and *mrnd48()* use an internal buffer to store $r(n)$. For these functions the initial value of $r(0) = 0\text{x1234abcd330e} = 20017429951246$.

On the other hand, *erand48()*, *nrand48()*, and *jrand48()* use a user-supplied buffer to store the seed $r(n)$, which consists of an array of 3 shorts, where the zeroth member holds the least significant bits.

All functions share the same multiplicand and addend.

srand48() is used to initialize the internal buffer $r(n)$ of *drand48()*, *lrand48()*, and *mrnd48()* such that the 32 bits of the seed value are copied into the upper 32 bits of $r(n)$, with the lower 16 bits of $r(n)$ arbitrarily being set to 0x330e . Additionally, the constant multiplicand and addend of the algorithm are reset to the default values given above.

seed48() also initializes the internal buffer *r(n)* of *drand48()*, *lrand48()* and *mrand48()* but here all 48 bits of the seed can be specified in an array of 3 shorts, where the zeroth member specifies the lowest bits. Again, the constant multiplicand and addend of the algorithm are reset to the default values given above. *seed48()* returns a pointer to an array of 3 shorts which contains the old seed. This array is statically allocated, thus its contents are lost after each new call to *seed48()*.

Finally, *lcg48()* allows full control over the multiplicand and addend used in *drand48()*, *erand48()*, *lrand48()*, *nrand48()*, *mrand48()*, and *jrand48()*, and the seed used in *drand48()*, *lrand48()*, and *mrand48()*. An array of 7 shorts is passed as parameter; the first three shorts are used to initialize the seed; the second three are used to initialize the multiplicand; and the last short is used to initialize the addend. It is thus not possible to use values greater than 0xffff as the addend.

Note that all three methods of seeding the random number generator always also set the multiplicand and addend for any of the six generator calls.

For a more powerful random number generator, see *random()*

AUTHOR

Martin Birgmeier

SEE ALSO

rand(), *random()*

NAME

`random()`, `srandom()`, `initstate()`, `setstate()` – better random number generator; routines for changing generators

SYNOPSIS

```
#include <stdlib.h>

long random (void)

void srandom (unsigned seed)

char * initstate (unsigned seed, char *state, int n)

char * setstate (char *state)
```

DESCRIPTION

The `random()` function uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random number generator is very large, approximately $16 \times (2^{31}-1)$.

`Srandom()` has almost the same calling sequence and initialization properties as `srand()`. The difference is that `rand()` produces a much less random sequence — in fact, the low dozen bits generated by `() rand` go through a cyclic pattern. All the bits generated by `random()` are usable. For example, `random() & 01` will produce a random binary value.

Unlike `srand()`, `srandom()` does not return the old seed; the reason for this is that the amount of state information used is much more than a single word. (Two other routines are provided to deal with restarting / changing random number generators). Like `rand()`, however, `random()` will by default produce a sequence of numbers that can be duplicated by calling `srandom()` with 1 as the seed.

The `initstate()` routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by `initstate()` to decide how sophisticated a random number generator it should use — the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error.) The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. The `initstate()` function returns a pointer to the previous state information array.

Once a state has been initialized, the `setstate()` routine provides for rapid switching between states. The `setstate()` function returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to `initstate()` or `setstate()`.

Once a state array has been initialized, it may be restarted at a different point either by calling `initstate()` (with the desired seed, the state array, and its size) or by calling both `setstate()` (with the state array) and `srandom()` (with the desired seed). The advantage of calling both `setstate()` and `srandom()` is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69} , which should be sufficient for most purposes.

AUTHOR

Earl T. Cohen

RETURN VALUES

If *initstate()* is called with less than 8 bytes of state information, or if *setstate()* detects that the state information has been garbled, error messages are printed on the standard error output.

BUGS

About 2/3 the speed of *rand()*.

SEE ALSO

rand()

NAME

rand(), srand() – system random number generator

SYNOPSIS

```
#include <stdlib.h>

void srand (unsigned seed)
int rand (void)
```

DESCRIPTION

The *rand()* function computes a sequence of pseudo-random integers in the range of 0 to {RAND_MAX} (as defined in the header file **<stdlib.h>**).

The *srand()* function sets its argument as the seed for a new sequence of pseudo-random numbers to be returned by *rand()*. These sequences are repeatable by calling *srand()* with the same seed value.

If no seed value is provided, the functions are automatically seeded with a value of 1.

To seed the random number generator with the time (a popular source), you can use:

```
srand((unsigned int)time(NULL));
```

SEE ALSO

random()

NAME

`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 {
    void *iov_base;
    size_t iov_len;
};
```

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

On objects capable of seeking, the `read()` starts at a position given by the pointer associated with *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.

When the *nbytes* argument exceeds {SSIZE_MAX}, the return value is truncated to type **ssize_t**.

ERRORS

`Read()` and `readv()` will succeed unless:

- | | |
|--------------|--|
| [EAGAIN] | The file was marked for non-blocking I/O, and no data were ready to be read. |
| [EBADF] | <i>D</i> is not a valid file descriptor open for reading. |
| [ECONNRESET] | The connection was forcibly closed by one of the peers; this is normally an indication of a timeout or a reboot. |

[EFAULT]	<i>Buf</i> points outside the allocated address space.
[EINTR]	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
[EINVAL]	The pointer associated with <i>d</i> was negative.
[EIO]	An I/O error occurred while reading from the file system.

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

[EINVAL]	<i>Iovcnt</i> was less than or equal to 0, or greater than 16.
[EINVAL]	One of the <i>iov_len</i> values in the iov array was negative.
[EINVAL]	The sum of the <i>iov_len</i> 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()*

NAME

realloc() – reallocate memory

SYNOPSIS

```
#include <stdlib.h>
```

```
void * realloc (void *ptr, size_t size)
```

DESCRIPTION

The *realloc()* function changes the size of the object pointed to by *ptr* to the size specified by *size*. The contents of the object are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If *ptr* is a null pointer, the *realloc()* function behaves like the *malloc()* function for the specified size. Otherwise, if *ptr* does not match a pointer earlier returned by the *calloc()*, *malloc()*, or *realloc()* function, or if the space has been deallocated by a call to the *free* or *realloc()* function, unpredictable and usually detrimental behavior will occur. If the space cannot be allocated, the object pointed to by *ptr* is unchanged. If *size* is zero and *ptr* is not a null pointer, the object it points to is freed.

The *realloc()* function returns either a null pointer or a pointer to the possibly moved allocated space.

SEE ALSO

calloc(), *free()*, *malloc()*

NAME

recv(), recvfrom() – receive a message from a socket

SYNOPSIS

```
#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)
```

DESCRIPTION

Recvfrom() is 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

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.

RETURN VALUES

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

ERRORS

The calls fail if:

[EBADF]	The argument <i>s</i> is an invalid descriptor.
[ENOTCONN]	The socket is associated with a connection-oriented protocol and has not been connected (see <i>connect()</i> and <i>accept()</i>).
[ENOTSOCK]	The argument <i>s</i> 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.
[EINTR]	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(), *read()*, *select()*, *getsockopt()*, *socket()*

NAME

regcomp(), regexexec(), regerror(), regfree() – regular-expression library

SYNOPSIS

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

int regcomp (regex_t *preg, const char *pattern, int cflags);
int regexexec (const regex_t *preg, const char *string,
               size_t nmatch, regmatch_t pmatch[], int eflags);
size_t regerror (int errcode, const regex_t *preg,
                char *errbuf, size_t errbuf_size);
void regfree(regex_t *preg);
```

DESCRIPTION

These routines implement POSIX 1003.2 regular expressions (“RE”s); see *re_format*. *Regcomp()* compiles an RE written as a string into an internal form, *regexexec()* matches that internal form against a string and reports results, *regerror()* transforms error codes from either into human-readable messages, and *regfree()* frees any dynamically-allocated storage used by the internal form of an RE.

The header **<regex.h>** declares two structure types, **regex_t** and **regmatch_t**, the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type **regoff_t**, and a number of constants with names starting with “REG_”.

Regcomp() compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags*, and places the results in the **regex_t** structure pointed to by *preg*. *Cflags* is the bitwise OR of zero or more of the following flags:

REG_EXTENDED

Compile modern (“extended”) REs, rather than the obsolete (“basic”) REs that are the default.

REG_BASIC

This is a synonym for 0, provided as a counterpart to REG_EXTENDED to improve readability.

REG_NOSPEC

Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the “RE” is a literal string. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. REG_EXTENDED and REG_NOSPEC may not be used in the same call to *regcomp()*.

REG_ICASE

Compile for matching that ignores upper/lower case distinctions. See *re_format*.

REG_NOSUB

Compile for matching that need only report success or failure, not what was matched.

REG_NEWLINE

Compile for newline-sensitive matching. By default, newline is a completely

ordinary character with no special meaning in either REs or strings. With this flag, '[' bracket expressions and '.' never match newline, a '^' anchor matches the null string after any newline in the string in addition to its normal function, and the '\$' anchor matches the null string before any newline in the string in addition to its normal function.

REG_PEND

The regular expression ends, not at the first NUL, but just before the character pointed to by the *re_endp* member of the structure pointed to by *preg*. The *re_endp* member is of type **const char ***. This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

RETURN VALUES

When successful, *regcomp()* returns 0 and fills in the structure pointed to by *preg*. One member of that structure (other than *re_endp*) is publicized: *re_nsub*, of type **size_t**, contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If *regcomp()* fails, it returns a non-zero error code; see ERRORS.

Regexec() matches the compiled RE pointed to by *preg* against the *string*, subject to the flags in *eflags*, and reports results using *nmatch*, *pmatch*, and the returned value. The RE must have been compiled by a previous invocation of *regcomp()*. The compiled form is not altered during execution of *regexec()*, so a single compiled RE can be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by *string* is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of zero or more of the following flags:

REG_NOTBOL

The first character of the string is not the beginning of a line, so the '^' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_NOTEOL

The NUL terminating the string does not end a line, so the '\$' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_STARTEND

The string is considered to start at *string + pmatch[0].rm_so* and to have a terminating NUL located at *string + pmatch[0].rm_eo* (there need not actually be a NUL at that location), regardless of the value of *nmatch*. See below for the definition of *pmatch* and *nmatch*. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Note that a non-zero *rm_so* does not imply REG_NOTBOL; REG_STARTEND affects only the location of the string, not how it is matched.

See *re_format* for a discussion of what is matched in situations where an RE or a portion thereof could match any of several substrings of *string*.

Normally, *regexec()* returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations; see ERRORS.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, *regexec()* ignores the *pmatch* argument (but see below for the case where REG_STARTEND is specified). Otherwise, *pmatch* points to an array of *nmatch* structures of type *regmatch_t*(). Such a structure has at least the members *rm_so* and *rm_eo*, both of type *regoff_t*() (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to *regexec()*. An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the *pmatch* array is filled in to indicate what substring of *string* was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i*, with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array—corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, *i* > *preg->re_nsub*)—have both *rm_so* and *rm_eo* set to -1. If a subexpression participated in the match several times, the reported substring is the last one it matched. (Note, as an example in particular, that when the RE '(b*)+' matches 'bbb', the parenthesized subexpression matches each of the three 'b's and then an infinite number of empty strings following the last 'b', so the reported substring is one of the empties.)

If REG_STARTEND is specified, *pmatch* must point to at least one *regmatch_t*() (even if *nmatch* is 0 or REG_NOSUB was specified), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by *nmatch*; if *nmatch* is 0 or REG_NOSUB was specified, the value of *pmatch*[0] will not be changed by a successful *regexec()*.

Regerror() maps a non-zero *errcode* from either *regcomp()* or *regexec()* to a human-readable, printable message. If *preg* is non-NULL, the error code should have arisen from use of the *regex_t*() pointed to by *preg*, and if the error code came from *regcomp()*, it should have been the result from the most recent *regcomp()* using that *regex_t*(). (*Regerror()* may be able to supply a more detailed message using information from the *regex_t*.) *Regerror()* places the NUL-terminated message into the buffer pointed to by *errbuf*, limiting the length (including the NUL) to at most *errbuf_size* bytes. If the whole message won't fit, as much of it as will fit before the terminating NUL is supplied. In any case, the returned value is the size of buffer needed to hold the whole message (including terminating NUL). If *errbuf_size* is 0, *errbuf* is ignored but the return value is still correct.

If the *errcode* given to *regerror()* is first ORed with REG_ITOA, the “message” that results is the printable name of the error code, e.g. “REG_NOMATCH”, rather than an explanation thereof. If *errcode* is REG_ATOI, then *preg* shall be non-NULL and the *re_endp* member of the structure it points to must point to the printable name of an error code; in this case, the result in *errbuf* is the decimal digits of the numeric value of the error code (0 if the name is not recognized).

REG_ITOA and REG_ATOI are intended primarily as debugging facilities; they are extensions, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

Regfree() frees any dynamically-allocated storage associated with the compiled RE pointed to by *preg*. The remaining *regex_t*() is no longer a valid compiled RE and the effect of supplying it to *regexec()* or *regerror()* is undefined.

None of these functions references global variables except for tables of constants; all are safe for use from multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that 1003.2 leaves up to the implementor, either by explicitly saying “undefined” or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See *re_format* for a discussion of the definition of case-independent matching.

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by 1003.2 (such magic meanings occur only in obsolete [“basic”] REs) is taken as an ordinary character.

Any unmatched [is a REG_EBRACK error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

RE_DUP_MAX, the limit on repetition counts in bounded repetitions, is 255.

A repetition operator (?, *, +, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow ‘^’ or ‘|’.

‘|’ cannot appear first or last in a (sub)expression or after another ‘|’, i.e. an operand of ‘|’ cannot be an empty subexpression. An empty parenthesized subexpression, ‘()’, is legal and matches an empty (sub)string. An empty string is not a legal RE.

A ‘{’ followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A ‘{’ *not* followed by a digit is considered an ordinary character.

‘^’ and ‘\$’ beginning and ending subexpressions in obsolete (“basic”) REs are anchors, not ordinary characters.

NOTE

This implementation is based on the work of Henry Spencer.

ERRORS

Non-zero error codes from *regcomp()* and *regexexec()* include the following:

REG_NOMATCH

regexexec() failed to match

REG_BADPAT invalid regular expression

REG_ECOLLATE

invalid collating element

REG_ECTYPE invalid character class

REG_EESCAPE backslash (\) applied to unescapable character

REG_ESUBREG

invalid backreference number

REG_EBRACK brackets [] not balanced

REG_EPAREN	parentheses () not balanced
REG_EBRACE	braces { } not balanced
REG_BADBR	invalid repetition count(s) in { }
REG_ERANGE	invalid character range in []
REG_ESPACE	ran out of memory
REG_BADRPT	?, *, or + operand invalid
REG_EMPTY	empty (sub)expression
REG_ASSERT	“can’t happen”—you found a bug
REG_INVARG	invalid argument, e.g. negative-length string

SEE ALSO*grep**re_format*

POSIX 1003.2, sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

NAME

remove() – remove file from a directory

SYNOPSIS

```
#include <unistd.h>

int remove (const char *path)
```

DESCRIPTION

The *remove()* 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.

Calling *remove()* with a directory is not supported.

This function is identical to *unlink()*.

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 *remove()* succeeds unless:

[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.
[EBUSY]	The entry to be removed is the mount point for a mounted file system.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.
[EPERM]	The named file is a directory.
[EIO]	An I/O error occurred while deleting the directory entry or deallocating the file serial number.
[EROFS]	The named file resides on a read-only file system.

SEE ALSO

close(), *link()*, *unlink()*, *rmdir()*

NAME

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.

The process doesn't require write access for the *from* or *to* directories in order for the call to succeed.

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:

[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.
[EFAULT]	<i>Path</i> points outside the process's allocated address space.
[EINVAL]	Either pathname contains a character with the high-order bit set.
[EINVAL]	<i>From</i> is a parent directory of <i>to</i> , or an attempt is made to rename . or ...
[EIO]	An I/O error occurred while making or updating a directory entry.
[EISDIR]	<i>To</i> is a directory, but <i>from</i> is not a directory.
[ENAMETOOLONG]	A component of either pathname exceeded {NAME_MAX} characters, or the entire length of either pathname exceeded {PATH_MAX} characters.
[ENOENT]	A component of the <i>from</i> path does not exist, or a path prefix of <i>to</i> does not exist.
[ENOSPC]	The directory in which the entry for the new name is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOTDIR]	A component of either path prefix is not a directory.
[ENOTDIR]	<i>from</i> is a directory, but <i>to</i> is not a directory.
[ENOTEMPTY]	<i>To</i> is a directory and is not empty.

[EPERM]	The directory containing <i>from</i> is marked sticky, and neither the containing directory nor <i>from</i> are owned by the effective user ID.
[EPERM]	The file cannot be moved or renamed to different file system (for example, //C/foo to //D/foo).
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EXDEV]	The link named by <i>to</i> and the file named by <i>from</i> are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.

SEE ALSO*open()*

NAME

renamewtmpx() – rename current wtmpx file and start new one

SYNOPSIS

```
renamewtmpx (const char *newname)
```

DESCRIPTION

The *renamewtmpx()* function renames the file **\$OPENNT_ROOT/var/adm/wtmpx** to *newname*.

This ensures that the current **wtmpx** file is renamed and that a new **wtmpx** file is started in its place, and that no data is lost during the renaming process.

Failure to use this interface means that data continues to be sent to the original file, even under a new name.

RETURN VALUES

The *renamewtmpx()* function returns 0 on success; on failure, it returns -1 and sets the global variable *errno*.

ERRORS

[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.

[EINVAL]

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

[EISDIR]

Newname is a directory.

[ENAMETOOLONG]

The *newname* exceeds {NAME_MAX} characters.

[ENOENT]

A path prefix of *newname* does not exist.

[ENOTDIR]

A component of *newname*'s path prefix is not a directory.

[EPERM]

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

[EPERM]

The file cannot be moved or renamed to different file system (for example, **//C/foo** to **//D/foo**).

[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.

[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]

Newname points outside the process's allocated address space.

[EEXIST]

The file *newname* already exists.

SEE ALSO

mvwtmptx

rename

NAME

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(3)*, 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(exec, tcp)
```

(see *getservent*) will return a pointer to a structure, which contains the necessary port. The protocol for connection is described in detail in *rexecd*.

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 standard 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, *rexecd*

NAME

rindex() – locate character in string

SYNOPSIS

```
#include <strings.h>
```

```
char * rindex (const char *s, int c)
```

DESCRIPTION

The *rindex()* function locates the last character matching *c* (converted to a **char**) in the null-terminated string *s*.

RETURN VALUES

A pointer to the character is returned if it is found; otherwise NULL is returned. If *c* is `\0`, *rindex()* locates the terminating `\0`.

SEE ALSO

index(), *memchr()*, *strchr()*, *strcspn()*, *strpbrk()*, *strrchr()*, *strsep()*, *strspn()*, *strstr()*, *strtok()*

NAME

rint() – round-to-closest integer functions

SYNOPSIS

```
#include <math.h>

double rint (double x)
```

DESCRIPTION

The *rint()* function finds the integer (represented as a double precision number) nearest to *x* in the direction of the prevailing rounding mode.

NOTES

On a VAX, *rint(x)* is equivalent to adding half to the magnitude and then rounding towards zero.

In the default rounding mode, to nearest, on a machine that conforms to IEEE 754, *rint(x)* is the integer nearest *x* with the additional stipulation that if $|rint(x) - x|$ is even. Other rounding modes can make *rint()* act like *floor()*, or like *ceil()*, or round towards zero.

Another way to obtain an integer near *x* is to declare (in C)

```
double x;    int k;    k = x;
```

Most C compilers round *x* towards 0 to get the integer *k*, but some do otherwise. If in doubt, use *floor()*, *ceil()*, or *rint()* first, whichever you intend. Also note that, if *x* is larger than *k* can accommodate, the value of *k* and the presence or absence of an integer overflow are hard to predict.

SEE ALSO

abs(), *fabs()*, *ceil()*, *floor()*, *ieee()*, *math()*

NAME

rmdir() – remove a directory

SYNOPSIS

```
#include <unistd.h>

int rmdir (const char *path)
```

DESCRIPTION

Rmdir() removes a directory whose name is given by *path*. The directory must not have any entries other than dot (.) and dot-dot (..).

Rmdir() will fail if *path* specifies the root directory.

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 directory is removed unless:

[EBUSY]	The directory named by <i>path</i> was either the root directory or it was the working directory of a process.
[ENOTDIR]	A component of the path is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
[ENOENT]	The named directory does not exist.
[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 file serial number.
[EROFS]	The directory entry to be removed resides on a read-only file system.
[EFAULT]	<i>Path</i> points outside the process's allocated address space.

SEE ALSO

mkdir(), *unlink()*

NAME

scanf(), fscanf(), sscanf(), vscanf(), vsscanf(), vfscanf() – input format conversion

SYNOPSIS

```
#include <stdio.h>

int scanf (const char *format ...)
int fscanf (FILE *stream, const char *format ...)
int sscanf (const char *str, const char *format ...)

#include <stdarg.h>

int vscanf (const char *format, va_list ap)
int vsscanf (const char *str, const char *format, va_list ap)
int vfscanf (FILE *stream, const char *format, va_list ap)
```

DESCRIPTION

The *scanf()* family of functions scans input according to a *format* as described below. This format may contain *conversion specifiers*; the results from such conversions, if any, are stored through the *pointer* arguments. The *scanf()* function reads input from the standard input stream *stdin*, *fscanf()* reads input from the stream pointer *stream*, and *sscanf()* reads its input from the character string pointed to by *str*. The *vfscanf()* function is analogous to *vfprintf()* and reads input from the stream pointer *stream* using a variable argument list of pointers (see <stdarg.h>). The *vscanf()* function scans a variable argument list from the standard input and the *vsscanf()* function scans it from a string; these are analogous to the *vprintf()* and *vsprintf()* functions respectively. Each successive pointer argument must correspond properly with each successive conversion specifier (but see ‘suppression’ below). All conversions are introduced by the % (percent sign) character. The *format* string may also contain other characters. White space (such as blanks, tabs, or newlines) in the *format* string match any amount of white space, including none, in the input. Everything else matches only itself. Scanning stops when an input character does not match such a format character. Scanning also stops when an input conversion cannot be made (see below).

CONVERSIONS

Following the % character introducing a conversion there may be a number of *flag* characters, as follows:

%[*][*fieldwidth*][*size*][*conversion*]

- * Suppresses assignment. The conversion that follows occurs as usual, but no pointer is used; the result of the conversion is simply discarded.

fieldwidth

There may be an optional maximum field width, expressed as a decimal integer. If no width is given, a default of ‘infinity’ is used (with one exception, below); otherwise at most this many characters are scanned in processing the conversion. Before conversion begins, most conversions skip white space; this white space is not counted against the field width.

size The characters h, l, and L indicate the size of the receiving object:

- h Indicates that the corresponding pointer is a **short** data type: **hd**, **hi**, and **hn** indicate a **short int**; **ho**, **hu**, and **hx** indicate an **unsigned short int**.

- l (The letter ell.) Indicates that the corresponding pointer is a **long** or **double** data type: **ld**, **li**, and **ln** indicate a **long int**; **lo**, **lu**, and **lx** indicate an **unsigned long int**; **le**, **lf**, and **lg** indicate a **double**.
- L Indicates that the corresponding pointer is a **long double** type: **Le**, **Lf**, and **Lg** indicate a **long double**.

The following *conversions* are available:

- % Matches a literal '%'. That is, '%%' in the format string matches a single input '%' character. No conversion is done, and assignment does not occur.
- c Matches a sequence of *width* count characters (default 1); the next pointer must be a pointer to **char**, and there must be enough room for all the characters (no terminating NUL is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.
- d Matches an optionally signed decimal integer; the next pointer must be a pointer to **int**.
- E, e Equivalent to f.
- F, f Matches an optionally signed floating-point number; the next pointer must be a pointer to **float**.
- g Equivalent to f.
- i Matches an optionally signed integer; the next pointer must be a pointer to **int**. The integer is read in base 16 if it begins with 0x or 0X, in base 8 if it begins with 0, and in base 10 otherwise. Only characters that correspond to the base are used.
- o Matches an octal integer; the next pointer must be a pointer to **int**.
- u Matches an optionally signed decimal integer; the next pointer must be a pointer to **unsigned int**.
- x Matches an optionally signed hexadecimal integer; the next pointer must be a pointer to **unsigned int**.
- s Matches a sequence of non-white-space characters; the next pointer must be a pointer to **char**, and the array must be large enough to accept all the sequence and the terminating NUL character. The input string stops at white space or at the maximum field width, whichever occurs first.
- [Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to **char**, and there must be enough room for all the characters in the string, plus a terminating NUL character. To match a string that is not delimited by white space, use this to specify a set of characters within square brackets ([]). This is used when the s conversion specifier is not appropriate. The usual skip of leading white space is suppressed.

If the first character after the open bracket is a circumflex (^), the set of matching characters *excludes* the ones in brackets.

To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set.

The hyphen character – is also special; when placed between two other characters, it adds all intervening characters to the set. To include a hyphen, make it the last character before the final close bracket. For instance, `[^]0-9-]` means the set ‘everything except close bracket, zero through nine, and hyphen’. The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out.

- p Matches a pointer value (as printed by `%p` in *printf()*); the next pointer must be a pointer to **void**.
- n Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to **int**. This is *not* a conversion, although it can be suppressed with the `*` flag.

RETURN VALUES

These functions return the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, while there was input available, no conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a `%d` conversion. The value EOF is returned if an input failure occurs before any conversion such as an end-of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.

SEE ALSO

getc(), *printf()*, *strtod()*, *strtol()*, *strtoul()*

NAME

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*-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 (&fdsetx)
    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 the maximum number of descriptors supported by the system or greater. In this implementation, FD_SETSIZE is 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.

SEE ALSO

accept(), *connect()*, *gettimeofday()*, *read()*, *recv()*, *send()*, *write()*

BUGS

User programs cannot be written independent of the kernel limit on the number of open files, and the dimension of a sufficiently large bit field for select remains a problem. The default size `FD_SETSIZE` is the current maximum number of open files (`{OPEN_MAX}`). In this implementation, you cannot open more files than `{OPEN_MAX}` with *select()*, so there is no reason to change the value of `FD_SETSIZE`.

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.

NAME

semctl() – semaphore control operations

SYNOPSIS

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

```
int semctl(int semid, int semnum, int cmd, ...)
```

DESCRIPTION

The *semctl()* function provides control operations over semaphores in the semaphore set identified by *semid*. The control operation is defined by the *cmd* argument. (Values for *cmd* are defined in **<sys/sem.h>**.) Some control operations act on a single semaphore in the set, others act on the entire set identified by *semid*.

Semaphore identifiers are obtained with the *semget()* call; semaphore operations are performed with the *semop()* call.

Operations on a Single Semaphore

The following values for *cmd* act on the semaphore identified by both *semid* and *semnum*.

<i>cmd</i>	Command
GETPID	Returns value of <i>sempid</i> . Requires read permission.
GETNCNT	Returns value of <i>semncnt</i> . Requires read permission.
GETZCNT	Returns value of <i>semzcnt</i> . Requires read permission.
GETVAL	Returns value of <i>semval</i> member. Requires read permission.
SETVAL	Sets value of <i>semval</i> member. Requires alter permission.
IPC_RMID	Removes <i>semid</i> from the system and destroy all associated structures and semaphores. Requires permission to be same as the value of <i>sem_perm.cuid</i> or <i>sem_perm.uid</i> in the semid_ds structure associated with <i>semid</i> .

See the *semget()* reference page for a description of the structure members. The IPC_RMID macro is defined in **<sys/ipc.h>**, which is included by **<sys/sem.h>**.

Operations on a Semaphore Set

These values of *cmd* operate on each semaphore in the set. They require a fourth argument to the *semctl()* function. The fourth argument contains an array which is used to hold values for the semaphores; it's described in more detail under “The Fourth Argument”.

<i>cmd</i>	Command
GETALL	For each semaphore in the set, place the value of <i>semval</i> in the array pointed to by <i>arg.array</i> . Requires read permission.
SETALL	For each semaphore in the set, set the value of <i>semval</i> to the corresponding value in the array pointed to by <i>arg.array</i> . When successful, this operation clears the <i>semaadj</i> values

<i>cmd</i>	Command
	corresponding to each semaphore. Requires alter permission.

Setting, Reading, and Removing Semaphores

The following values of *cmd* deal with *stat*'ing, setting, and removing semaphores:

<i>cmd</i>	Command
IPC_STAT	Copy current value of each member in the data structure associated with <i>semid</i> into the data structure pointed at by <i>arg.buf</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> . Requires read permission.
IPC_SET	Set the value of UID, GID, and mode permission members in the semid_ds data structure associated with <i>semid</i> to the corresponding value in the data structure point at by <i>arg.buf</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> . The calling process needs an effective UID that matches the semaphore's <i>sem_perm.cuid</i> or <i>sem_perm.uid</i> values.
IPC_RMID	Remove the semaphore specified by <i>semid</i> from the system and destroy the set of semaphores and datastructure associated. The calling process needs an effective UID that matches the semaphore's <i>sem_perm.cuid</i> or <i>sem_perm.uid</i> values.

The Fourth Argument

For these values of *cmd*, a fourth argument is required:

SETVAL, GETALL, SETALL, IPC_STAT, IPC_SET, IPC_RMID

This argument is of type **union semun** and the application program must explicitly declare it:

```
union semun {
    int                val;
    struct semid_ds *  buf;
    unsigned short *   array;
};
```

RETURN VALUE

The return value for a successful call to *semctl()* depends upon the *cmd*:

GETVAL
The value of *semval*.

GETPID
The value of *sempid*.

GETNCNT
The value of *semmcnt*.

GETZCNT

The value of *semnzcnt*.

All others

The value 0.

On failure, *semctl()* returns -1 and sets *errno* to indicate the error.

ERRORS

The *semctl()* function can fail for the following reasons:

- | | |
|----------|---|
| [EACCES] | The calling process does not have permission to perform the operation. |
| [EINVAL] | <i>Semid</i> is not a valid semaphore identifier, or <i>semnum</i> is outside of the range 0–(<i>semnum</i> –1), or <i>cmd</i> is not a valid command. |
| [EPERM] | <i>Cmd</i> was IPC_RMID or IPC_SET but either calling process did not have the appropriate privileges or the effective user ID of the calling process wasn't equal to the user ID of the semaphore. |
| [ERANGE] | <i>Cmd</i> is SETVAL or SETALL but the value given for <i>semval</i> is greater than the system maximum. |
| [ERANGE] | <i>Cmd</i> is SETVAL or SETALL, but the intended value for <i>semval</i> is greater than the system maximum. |

SEE ALSO

semget(), *semop()*

NAME

semget() – get or create a set of semaphores

SYNOPSIS

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

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

DESCRIPTION

The *semget()* function returns the semaphore identifier associated with *key*. If the following are true, *semget()* creates the semaphore identifier, its associated **semid_ds** data structure and the associated set of semaphores (the number of semaphores in the set is determined by *nsems*):

- The *key* is `IPC_PRIVATE`.
- The *key* doesn't already have a semaphore associated with it, and the value of (*semflg* & `IPC_CREAT`) is not zero.

The **semid_ds** data structure has the following members:

```
struct semid_ds {
    struct ipc_perm    sem_perm    operation permission structure
    unsigned short int sem_nsems   number of semaphores in set
    time_t             sem_otime   time of last semop()
    time_t             sem_ctime   time of last change by semctl()
}
```

The new **semid_ds** data structure is initialized to:

- The IDs in the operation permissions structure are set to those of the effective IDs of the calling process. That is, the members *sem_perm.cuid*, *sem_perm.uid*, *sem_perm.cgid*, and *sem_perm.gid* are set to the value of the calling process' effective user ID and effective group ID. (On INTERIX systems, these are the user ID and the group ID, respectively.)
- The low-order 9 bits of *sem_perm.mode* are set to the low-order 9 bits of *semflg*.
- The value of *sem_nsems* is set to the value of *nsems*.
- The value of *sem_otime* is set to 0.
- The value of *sem_ctime* is set to the current time.

Note: The data structure associated with each semaphore in the set is *not* initialized. Use the *semctl()* function with the command `SETVAL` or `SETALL` to initialize each semaphore.

A semaphore contains the following members (the data structure is anonymous):

```
unsigned short int  semval    semaphore value
pid_t              sempid    process ID of last operation
                                number of processes
```

		waiting for <i>semval</i> to become greater than current value
unsigned short int	semncnt	
unsigned short int	semzcnt	number of processes waiting for <i>semval</i> to become 0

The header file `<sys/sem.h>` includes `<sys/ipc.h>`.

RETURN VALUE

On success, *semget()* returns the semaphore identifier, a non-negative integer; on failure, it returns `-1` and sets *errno* to indicate the error.

ERRORS

The *semget()* function can fail for these reasons:

- | | |
|----------|--|
| [EACCES] | Although a semaphore identifier exists for <i>key</i> , its permissions are not compatible with the low-order 9 bits of <i>semflg</i> . |
| [EEXIST] | Although a semaphore identifier exists for <i>key</i> , the value of <code>((semflg & IPC_CREAT) && (semflg & IPC_EXCL))</code> is non-zero. |
| [EINVAL] | The value of <i>nsems</i> is outside the system boundaries (less than or equal to 0 or greater than the limit), or an identifier exists for <i>key</i> but the number of semaphores in the set associated with it is less than <i>nsems</i> and <i>nsems</i> isn't equal to 0. |
| [ENOENT] | No semaphore identifier exists for <i>key</i> , and <code>(semflg & IPC_CREAT)</code> equals 0. |
| [ENOSPC] | Creating the identifier would exceed the system-wide maximum number of semaphores allowed. |

SEE ALSO

semctl(), *semop()*

NAME

semop() – operate on a set of semaphores

SYNOPSIS

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

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

DESCRIPTION

The *semop()* function performs atomic actions on the set of semaphores associated with the semaphore associated with *semid*. The operations are specified in *sops*, a user-defined array of semaphore operation structures (**sembuf** structures). The argument *nsops* specifies the number of structures in the array.

The structure **sembuf** contains the following members:

```
struct sembuf {
    unsigned short int    sem_num    semaphore number
    short int             sem_op     semaphore operation
    short int             sem_flg     operation flags
};
```

There are really only three semaphore operations (specified by *sem_op*), although the conditions under which each occurs can be difficult to understand. The three semaphore operations are:

- Reduce the value of *semval* by the absolute value of *semop*.
- Return immediately rather than wait for a synchronization event.
- Suspend execution of the process until a synchronization event. What constitutes a *synchronization event* depends upon whether *sem_op* is positive or negative.

The actual operation performed depends upon the permissions of the calling process, the value of *sem_op*, the value of *semval* (in the **sembuf** structure), and the value of *sem_flg*.

1. *semop* is **negative**, and the calling process has **alter permission**:

- If *semval* (from the **sembuf** structure) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is **subtracted** from *semval*.

The calling process' *semadj* value may be incremented by the absolute value of *sem_op*.

- If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is non-zero, then *semop()* **returns immediately**.
- If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is 0, *semop()* increments the *semncnt* associated with the semaphore and waits for one of these synchronization events:
 - The value of *semval* becomes greater than or equal to the absolute value of *sem_op*. Then the value of *semncnt* associated with the semaphore is decremented, the absolute value of *sem_op* is subtracted from *semval*, and the calling process' *semadj* value may be increased by the absolute value of *sem_op*.

- The *semid* is removed from the system. This is an error; *errno* is set to [EIDRM] and the function return -1 .
 - The calling process catches a signal. The value of *semncnt* associated with the semaphore is decremented and the calling process resumes execution as described by *sigaction()*.
2. *Semop* is **positive**, and the calling process has **alter permission**:
 - *Sem_op* is added to *semval*.
 - The calling process' *semadj* value may be decreased by the value of *sem_op*.
 3. *Semop* is **zero** and the calling process has **read permission**:
 - If *semval* is 0, *semval()* returns immediately.
 - If *semval* is non-zero, and (*sem_flg* & IPC_NOWAIT) is non-zero, *semop()* returns immediately.
 - If *semval* is non-zero and (*sem_flg* & IPC_NOWAIT) is 0, *semop()* increments the *semzcnt* associated with the semaphore and waits for one of these synchronization events:
 - The value of *semval* is zero; then the value of *semzcnt* associated with the semaphore is decremented.
 - The *semid* is removed from the system. This is an error; *errno* is set to [EIDRM] and the function return -1 .
 - The calling process catches a signal. The value of *semncnt* associated with the semaphore is decremented and the calling process resumes execution as described by *sigaction()*.

Some operations adjust the value of *semadj*, a per-process variable stored in the process table. This happens if (*sem_flg* & SEM_UNDO) is non-zero.

The header file `<sys/sem.h>` includes `<sys/ipc.h>`.

RETURN VALUE

On success, *semop()* returns 0. for each semaphore in the array pointed to by *sops*, the value of *sempid* is set to the process ID of the calling process.

On failure, *semop()* returns -1 and sets *errno* to indicate the error.

ERRORS

Semop() can fail for any of these reasons:

[E2BIG]	The value of <i>nsops</i> is too large.
[EACCES]	The calling process did not have permission to perform the operation. Although a semaphore identifier exists for <i>key</i> , its permissions were not compatible with the low-order 9 bits of <i>sem_flg</i> .
[EAGAIN]	Although the call would suspend the calling process, the value of (<i>sem_flg</i> & IPC_NOWAIT) was non-zero
[EFBIG]	The value of <i>sem_num</i> was outside the range 0 to (number of semaphores in the set associated with <i>semid</i> –1).
[EIDRM]	The semaphore identifier <i>semid</i> was removed from the system.

[EINTR]	An interrupt occurred.
[EINVAL]	<i>Semid</i> was not a valid semaphore identifier.
[EINVAL]	The number of semaphores for which the process requested a SEM_UNDO exceeded the system maximum.
[ENOSPC]	The number of semaphores for which the process requested a SEM_UNDO exceeded the process maximum.
[ERANGE]	The operation would cause a <i>semval</i> to overflow the system limit, or the operation would cause a <i>semadj</i> to overflow the system limit.

SEE ALSO*exec**fork()*, *semctl()*, *semget()*

NAME

`send()`, `sendto()` – 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)
```

DESCRIPTION

`Send()` and `sendto()` are used to transmit a message to another socket. `Send()` may be used only when the socket is in a *connected* state, while `sendto()` 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:

MSG_OOB	process out-of-band data
MSG_DONTROUTE	bypass routing, use direct interface

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_DONTROUTE is usually used only by diagnostic or routing programs.

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.
[ENOTSOCK]	The argument <i>s</i> 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.
[ENOBUFS]	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
[ENOBUFS]	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.

SEE ALSO

fcntl(), recv(), select(), getsockopt(), socket(), write()

NAME

setbuf(), setvbuf() – stream buffering operations

SYNOPSIS

```
#include <stdio.h>

void setbuf (FILE *stream, char *buf)
int setvbuf (FILE *stream, char *buf, int mode, size_t size)
```

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is output or input is read from any stream attached to a terminal device (typically stdin). The function *fflush()* may be used to force the block out early. (See *fclose()*.)

Normally all files are block buffered. When the first I/O operation occurs on a file, *malloc()* is called, and an optimally-sized buffer is obtained. If a stream refers to a terminal (as stdout normally does) it is line buffered. The standard error stream stderr is always unbuffered.

The *setvbuf()* function may be used to alter the buffering behavior of a stream. The *mode* parameter must be one of the following three macros:

_IONBF unbuffered
_IOLBF line buffered
_IOFBF fully buffered

The *size* parameter may be given as zero to obtain deferred optimal-size buffer allocation as usual. If it is not zero, then except for unbuffered files, the *buf* argument should point to a buffer at least *size* bytes long; this buffer will be used instead of the current buffer.

The *setvbuf()* function may be used at any time, but may have peculiar side effects (such as discarding input or flushing output) if the stream is “active”. Portable applications should call it only once on any given stream, and before any I/O is performed.

Except for the lack of a return value, the *setbuf()* function is exactly equivalent to the call

```
setvbuf(stream, buf, buf ? _IOFBF: _IONBF, BUFSIZ);
```

RETURN VALUES

The *setvbuf()* function returns 0 on success, or EOF if the request cannot be honored (note that the stream is still functional in this case).

SEE ALSO

fopen(), *fclose()*, *fread()*, *malloc()*, *puts()*, *printf()*

NAME

setjmp(), longjmp(), sigsetjmp(), siglongjmp() – non-local jumps

SYNOPSIS

```
#include <setjmp.h>

int setjmp (jmp_buf env)
void longjmp (jmp_buf env, int val)
int sigsetjmp (sigjmp_buf env, int savemask)
void siglongjmp (sigjmp_buf env, int val)
```

DESCRIPTION

The *setjmp()* and *sigsetjmp()* functions save their calling environment in *env*. Each of these functions returns 0.

The corresponding *longjmp()* functions restore the environment saved by their most recent respective invocations of the *setjmp()* function. They then return so that program execution continues as if the corresponding invocation of the *setjmp()* call had just returned the value specified by *val*, instead of 0.

Sigsetjmp() and *siglongjmp()* save and restore the signal mask if the argument *savemask* is non-zero; otherwise they behave identically with *setjmp()* and *longjmp()*.

The *longjmp()* routines may not be called after the routine which called the *setjmp()* routines returns.

All accessible objects have values as of the time *longjmp()* routine was called, except that the values of objects of automatic storage invocation duration that do not have the **volatile** type and have been changed between the *setjmp()* invocation and *longjmp()* call are indeterminate.

Pairs of calls may be intermixed, i.e. both *sigsetjmp()* and *siglongjmp()* and *setjmp()* and *longjmp()* combinations may be used in the same program, however, individual calls may not, e.g. the *env* argument to *setjmp()* may not be passed to *siglongjmp()*.

RETURN VALUES

When invoked directly, *setjmp()* returns zero; when invoked as part of *longjmp()*, *setjmp()* returns non-zero.

When invoked directly, *sigsetjmp()* returns zero; when invoked as part of *siglongjmp()*, *sigsetjmp()* returns non-zero.

SEE ALSO

sigaction(), *signal()*

NAME

setlocale() – set or query the program's locale

SYNOPSIS

```
char *setlocale(int category, const char *locale)
```

DESCRIPTION

The *setlocale()* function changes or queries the program's locale. The *category* is used to specify part of the locale. It can have any one of these values (defined in **<locale.h>**):

LC_ALL	The entire locale
LC_COLLATE	Character collating sequence
LC_CTYPE	Character handling and multibyte functions
LC_MONETARY	Monetary formatting information
LC_NUMERIC	Decimal point character in I/O and string conversions
LC_TIME	Time formatting — see <i>strftime()</i>

The *locale* argument is a string naming a particular locale. All POSIX-conforming systems support the locales "C" and "POSIX". Up to and including release 2.0, INTERIX supports only the C and POSIX locales.

To query the current locale, use a null string ("") as the *locale*. *setlocale()* returns the name of the current locale for that *category*.

The return value may be overwritten by a subsequent call to *setlocale()*, so your program shouldn't modify it.

RETURN VALUE

On success, the *setlocale()* function returns a string that corresponds to the locale specified by *locale*. If the function fails, it returns NULL.

SEE ALSO

isalnum(), *isalpha()*, *iscntrl()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *localeconv()*, *strcoll()*, *strftime()*, *strxfrm()*

NAME

getmode(), setmode() – modify mode bits

SYNOPSIS

```
mode_t getmode (const void *set, mode_t mode)
void setmode (const char *mode_str)
```

DESCRIPTION

The *getmode()* function returns a copy of the file permission bits *mode* as altered by the values pointed to by *set*. While only the mode bits are altered, other parts of the file mode may be examined.

The *setmode()* function takes an absolute (octal) or symbolic value, as described in *chmod()*, as an argument and returns a pointer to mode values to be supplied to *getmode()*. Because some of the symbolic values are relative to the file creation mask, *setmode()* may call *umask()*. If this occurs, the file creation mask will be restored before *setmode()* returns. If the calling program changes the value of its file creation mask after calling *setmode()*, *setmode()* must be called again if *getmode()* is to modify future file modes correctly.

If the mode passed to *setmode()* is invalid, *setmode()* returns NULL.

ERRORS

The *setmode()* function may fail and set *errno* for any of the errors specified for the library routine *malloc()*.

SEE ALSO

chmod(), *stat()*, *umask()*, *malloc()*

NAME

setpgid() – set process group

SYNOPSIS

```
#include <unistd.h>

int setpgid (pid_t pid, pid_t pgrp)
```

DESCRIPTION

Setpgid() is used either to join an existing process group or to create a new process group within the session of the calling process. A new process group is created if and only if *pid* and *pgrp* denote the same process, and no errors occur.

Upon successful completion, the process group ID of the process specified by *pid* is set to *pgrp*.

If *pid* is zero, the process ID of the current process is used. If *pgrp* is zero, the process ID of the indicated group is used. This creates a new group, if the group doesn't already exist.

Setpgid() will not fail if the indicated process or the leader of the indicated process group are zombies. (This is consistent with the behavior of BSD systems.)

RETURN VALUES

Setpgid() returns 0 when the operation was successful. If the request failed, -1 is returned and the global variable *errno* indicates the reason.

ERRORS

Setpgid() will fail and the process group will not be altered if:

- | | |
|----------|---|
| [EACCES] | The value of <i>pid</i> matches the process ID of a child process and the child process has successfully executed one of the exec functions. |
| [EINVAL] | The value of the <i>pgrp</i> argument is non-zero, is not equal to the ID of the indicated process, and is not equal to the ID of any existing process group. |
| [EPERM] | The process indicated by <i>pid</i> is a session leader.

The value of <i>pid</i> matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process.

The value of <i>pgrp</i> is valid, but does not match the process ID of the process indicated by <i>pid</i> and there is no process with a process group ID that matches <i>pgrp</i> in the same session as the calling process. |
| [ESRCH] | The value of <i>pid</i> does not match the process ID of the calling process, or of a child of the calling process. |

SEE ALSO

getpgrp(), *setsid()*, *tcsetpgrp()*

NAME

setsid() – create session and set process group ID

SYNOPSIS

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

pid_t setsid(void)
```

DESCRIPTION

Setsid() creates a new session. The calling process is the session leader of the new session, is the process group leader of a new process group and has no controlling terminal. The calling process is the only process in either the session or the process group.

RETURN VALUES

Upon successful completion, the *setsid()* function returns the value of the process group ID of the new process group, which is the same as the process ID of the calling process.

ERRORS

If an error occurs, *setsid()* returns –1 and the global variable *errno* is set.

ERRORS

Setsid() can fail for the following reasons:

[EPERM]	The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.
---------	---

SEE ALSO

setpgid(), *tcgetpgrp()*, *tcsetpgrp()*

NAME

setuid(), setgid() – set user and group ID

SYNOPSIS

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

int setuid (uid_t uid)
int setgid (gid_t gid)
```

DESCRIPTION

Traditionally, the *setuid()* and *setgid()* functions set the real and effective user and group IDs of the current process. These functions cannot be implemented under Windows NT because of its strong security with respect to user authentication. These functions exist in INTERIX, but always return [EPERM] unless the user ID or group ID is the same as the user ID or group ID of the calling process. For more details on security and appropriate permissions, see the sections in the *Commands and Utilities User's Guide*, entitled “The Superuser and Appropriate Privileges” and “File Ownership and Windows NT Security”.

For reference purposes, the description of the functions is included below.

The *setuid()* function sets the real and effective user IDs and the saved set-user-ID of the current process to the specified value. The *setuid()* function is permitted if the specified ID is equal to the real user ID of the process.

The *setgid()* function sets the real and effective group IDs and the saved set-group-ID of the current process to the specified value. The *setgid()* function is permitted if the specified ID is equal to the real group ID of the process, or if the effective user ID is that of the super user.

RETURN VALUES

Upon success, these functions return 0; otherwise they return -1.

If the user is not the super user, or the *uid* specified is not the real, effective ID, or saved ID, these functions return -1.

In this implementation, the functions always return [EPERM] unless the user ID or group ID is the same as the user ID or group ID of the calling process. For more details on security and appropriate privileges, see the *INTERIX Commands and Utilities User's Guide*.

SEE ALSO

getuid(), *getgid()*

NAME

setuser() – change effective and real uid and gid of process

SYNOPSIS

```
int setuser(char *username, char *password, int flags)
```

DESCRIPTION

The *setuser()* function changes the effective and real uid and gid of the current process to that of the specified *username*. All of the security attributes and permissions become those of *username*. However, the process' current working directory does not change.

The *setuser()* utility takes the following arguments:

username

The name of the user. If the *username* is not fully qualified with a domain name (that is, *domain+name*), it is not changed. On a system configured for workgroups, specify the domain as NULL: that is, the string *+name*.

password

The plaintext password for the specified *username*.

flags

Control flags. At this time, the value of *flags* should always be 0.

There can be a performance degradation if a process changes identity to a user who does not have permission to be located in the current working directory. The best solution is to *chdir()* after a call to *setuser()*, to a directory known to be permitted for the new identity.

RETURN VALUE

On success, the *setuser()* function returns 0. On failure, it returns -1 and sets *errno*.

ERRORS

The *setuser()* function can fail for the following reasons:

[ENOMEM] Not enough memory to complete the request.

[EINVAL] An argument is invalid or is missing.

[EPERM] The requesting process does not have permission to change to the new user. Either or both of the username or password were incorrectly specified.

SEE ALSO

setuid(), *setgid()*, *exec_asuser()*, *authenticateuser()*

NAME

shmat() – attach shared memory segment

SYNOPSIS

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

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

DESCRIPTION

Shmat() attaches the shared memory segment identified by *shmid* to the address space of the calling process. *Shmid* is the value returned from *shmget()*. If *shmaddr* is NULL, the segment is attached at the first available address. Otherwise, the parameters *shmaddr* and *shmflag* determine the location of the added segment.

The data structure for shared memory segments and the control flags (such as SHM_RND) are defined in **<shm.h>**.

To specify the address, make *shmaddr* non-NULL and:

- if *shmflag* & SHM_RND is zero, the segment is attached at the address given by *shmaddr*.
- if *shmflag* & SHM_RND is non-zero, the segment is attached at the address

shmaddr – ((ptrdiff_t)*shmaddr*%SHMLBA))

where % is the C language modulus (remainder) operator.

Permissions on the shared memory segment are determined by the value of *shmflag*:

- If (*shmflag* & RD_ONLY) is zero, and the the calling process has read and write permission, the segment is attached for reading and writing.
- If (*shmflag* & SHM_RDONLY) is non-zero, and the calling process has read permission, the segment is attached for reading.

The header file **<sys/shm.h>** includes **<sys/ipc.h>**.

RETURN VALUE

On success, *shmat()* increments the value of *shm_nattach* in the data structure and returns the segment's start address.

On failure, *shmat()* returns -1 and sets *errno* to indicate the error.

ERRORS

The *shmat()* function can fail for the following reasons:

[EACCES]	Permission is denied to the calling process.
[EINVAL]	<i>Shmid</i> is not a valid shared memory identifier, or (if <i>shmaddr</i> is not NULL) the value of (<i>shmaddr</i> –((ptrdiff_t) <i>shmaddr</i> %SHMLBA)) is an illegal address, or <i>shmaddr</i> is not a valid address.
[EMFILE]	Adding this shared memory segment would exceed the system limit.
[ENOMEM]	The data space is not large enough to accommodate the shared memory segment.
[ENOSYS]	The function is not implemented.

EXAMPLES

See *shmctl()*.

SEE ALSO

shmctl(), *shmdt()*, *shmget()*

NAME

shmctl() – control shared memory operations

SYNOPSIS

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

```
int shmctl(int shmid, int cmd, struct shm_id_ds *buf)
```

DESCRIPTION

The *shmctl()* function gives you access to the data information associated with a shared memory segment, either to read or set the values of members, or to remove a shared memory identifier. argument.

The function parameters are:

- shmid* An identifier for the shared memory segment. The shared memory segment has a **shm_id_ds** data structure associated with it; see **<sys/shm.h>** for a description of the structure.
- cmd* The operation to be performed. The following are valid (defined in **<sys/shm.h>**):
- | | |
|----------|--|
| IPC_STAT | Copies the current values of the shm_id_ds data structure associated with <i>shmid</i> into <i>buf</i> . |
| IPC_SET | Sets the value of members of the shm_id_ds data structure associated with <i>shmid</i> from the same members in <i>buf</i> . The members copied are <i>shm_perm.uid</i> , <i>shm_perm.gid</i> , and the low-order nine bits of <i>shm_perm.mode</i> . |
| IPC_RMID | Removes the identifier specified by <i>shmid</i> from the system and destroys the shared memory segment and data structure associated with it. |
- buf* The buffer used to store information copied or to be copied.

The header file **<sys/shm.h>** includes **<sys/ipc.h>**.

RETURN VALUE

Shmctl() returns 0 on success. On failure, it returns -1 and sets *errno*.

ERRORS

The *shmctl()* function can fail for the following reasons:

- | | |
|----------|--|
| [EACCES] | The <i>cmd</i> is IPC_STAT and the calling process doesn't have read permission. |
| [EINVAL] | <i>Shmid</i> is not a valid shared memory identifier. |
| [ENOSYS] | The function isn't implemented. |
| [EPERM] | The <i>cmd</i> is IPC_RMID or IPC_SET and the calling process doesn't the appropriate privileges (that is, the ones set by <i>shm_perm.cuid</i> or <i>shm_perm.uid</i> in the data structure associated with <i>shmid</i>). |

The function may fail because of:

- | | |
|-------------|--|
| [EOVERFLOW] | The <i>cmd</i> is PC_STAT and the value of gid or uid is too large to be |
|-------------|--|

stored in the structure indicated by *buf*.

SEE ALSO

shmat(), *shmdt()*, *shmget()*

NAME

shmdt() – detach shared memory segment

SYNOPSIS

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

int shmdt(const void *shmaddr)
```

DESCRIPTION

The *shmdt()* function detaches the shared memory segment at *shmaddr* from the address space of the calling process.

The function parameters are:

shmaddr a pointer to a shared memory segment.

To delete a shared memory segment, use *shmctl()*.

The header file **<sys/shm.h>** includes **<sys/ipc.h>**.

RETURN VALUE

On success, the *shmdt()* function returns 0 and decrements the value of *shm_nattach* in the data structure associated with the shared memory segment.

On failure, the function return s-1 and sets *errno*.

ERRORS

The *shmdt()* function can fail for the following reasons:

- | | |
|----------|---|
| [EINVAL] | <i>Shmaddr</i> does not point to the start of a data segment for a shared memory segment. |
| [ENOSYS] | The function isn't implemented. |

SEE ALSO

shmat(), *shmctl()*, *shmget()*

NAME

shmget() – get shared memory segment

SYNOPSIS

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

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

DESCRIPTION

The *shmget()* function creates a shared memory identifier based upon the argument *key*. The function creates an identifier, data structure, and shared memory segment if:

- the value of *key* is `IPC_PRIVATE`; or
- *key* doesn't already have a shared memory identifier associated with it, and the value of (*shmflg* & `IPC_CREAT`) is non-zero.

The new shared memory segment is at least *size* bytes large. The permission mode (the low-order nine bits of the member *shm_perm.mode*) are set to the low-order nine bits of *shmflg*.

If there is already a shared memory segment associated with *key*, the function returns the identifier for that shared memory segment if the permissions on that shared memory segment allow the calling process to make use of it. If the permissions do not allow it, the function fails. (See the errors `[EACCES]` and `[EEXIST]`).

The members of the data structure associated with the shared memory identifier (described in `<sys/shm.h>`) are initialized as:

<i>shm_atime</i>	Zero
<i>shm_ctime</i>	Current time
<i>shm_dtime</i>	Zero
<i>shm_lpid</i>	Zero
<i>shm_nattch</i>	Zero
<i>shm_perm.cgid</i>	Effective group ID of calling process
<i>shm_perm.cuid</i>	Effective user ID of calling process
<i>shm_perm.gid</i>	Effective group ID of calling process
<i>shm_perm.uid</i>	Effective user ID of calling process
Low-order nine bits of <i>shm_perm.mode</i>	Low-order nine bits of <i>shmflg</i>
<i>shm_segsz</i>	Value of <i>size</i>

The header file `<sys/shm.h>` includes `<sys/ipc.h>`.

RETURN VALUE

On success, the *shmget()* function returns a shared memory identifier (a non-negative integer).

On failure, *shmget()* returns `-1` and sets *errno*.

ERRORS

The *shmget()* function can fail for the following reasons:

- | | |
|-----------------------|---|
| <code>[EACCES]</code> | A shared memory identifier already exists for <i>key</i> but the permissions specified by the low-order nine bits of <i>shmflg</i> wouldn't be granted. |
|-----------------------|---|

[EEXIST]	A shared memory identifier exists for <i>key</i> but the value (<i>shmflg</i> &IPC_CREAT)&&(<i>shmflg</i> &IPC_EXCL) is non-zero.
[EINVAL]	The value of <i>size</i> is invalid (outside system-imposed bounds), or a shared memory identifier already exists for <i>key</i> but the segment's size is less than <i>size</i> , and <i>size</i> is not zero.
[ENOENT]	Although (<i>shmflg</i> &IPC_CREAT) is 0, no shared memory identifier exists for <i>key</i> .
[ENOMEM]	There is not enough physical memory available to to create the identifier and the memory segment.
[ENOSPC]	Creating the shared memory identifier would exceed the system-wide maximum.
[ENOSYS]	the funtion is not implemented.

SEE ALSO

shmat(), *shmctl()*, *shmdt()*

NAME

shutdown() – shut down part of a full-duplex connection

SYNOPSIS

```
#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.

DIAGNOSTICS

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

ERRORS

The call succeeds unless:

- [EBADF] *S* is not a valid descriptor.
- [ENOTSOCK] *S* is a file, not a socket.
- [ENOTCONN] The specified socket is not connected.

SEE ALSO

connect(), *socket()*

NAME

sigaction() – software signal facilities

SYNOPSIS

```
#include <signal.h>
```

```
sigaction (int sig, struct sigaction *act, struct sigaction *oact)
```

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Upon receiving a signal, a process takes some action; the process may *ignore* the signal, it may deliver the signal to a specified *handler* function, it may *block* the signal (delivery is postponed until the signal is unblocked), or it may take the default action. The *sigaction()* call allows the calling process to examine or specify the action to be taken when a signal is received.

Sig specifies the signal. The action is described in a **struct sigaction** structure, defined in **<signal.h>**, which contains the following members:

```
void      (*sa_handler)(int);
sigset_t  sa_mask;
int       sa_flags;
```

The argument *act* is the new action associated with the signal and *oact* is the old action. If *act* is NULL, *sigaction()* fills in the members of *oact*, providing a way to query the system about the current action associated with a signal.

Within the **sigaction** structure:

- *sa_handler* is the action to be taken: one of SIG_DFL (default), SIG_IGN (ignore), or a pointer to a handler function.
- *sa_mask* is a set of additional signals (beyond those globally masked) to be blocked during the execution of the signal-catching function.
- *sa_flags* contains flags to modify the behavior of the specified signal. Currently this is limited to SA_NOCLDSTOP: do not generate the SIGCHLD signal when child processes stop.

Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally empty). It may be changed with a *sigprocmask()* call, or when a signal is delivered to the process.

These signals are defined in the file **<signal.h>**:

Symbol	Effect	Description
SIGABRT	terminate process	<i>abort()</i> call (formerly SIGIOT)
SIGALRM	terminate process	real-time timer expired
SIGBUS	terminate process	bus error
SIGCHLD	discard signal	child status has changed

SIGCONT	discard signal	continue after stop
SIGFPE	terminate process	floating-point exception
SIGHUP	terminate process	terminal line hangup
SIGILL	terminate process	illegal instruction
SIGINT	terminate process	interrupt program
SIGIO	discard signal	I/O possible on a descriptor (see <i>fcntl()</i>)
SIGKILL	terminate process	kill program
SIGPIPE	terminate process	write on a pipe with no reader
SIGPOLL		
SIGPROF	terminate process	profiling timer alarm
SIGQUIT	terminate process	quit program
SIGSEGV	terminate process	segmentation violation
SIGSTOP	stop process	stop (cannot be caught or ignored)
SIGSYS	terminate process	invalid argument to system call
SIGTERM	terminate process	software termination signal
SIGTRAP	terminate process	trace trap
SIGTSTP	stop process	stop signal generated from keyboard
SIGTTIN	stop process	background read attempted from control terminal
SIGTTOU	stop process	background write attempted to control terminal
SIGURG	discard signal	urgent condition present on socket
SIGUSR1	terminate process	User defined signal 1
SIGUSR2	terminate process	User defined signal 2
SIGVTALRM	terminate process	virtual time alarm
SIGXCPU	terminate process	CPU time limit exceeded
SIGXFSZ	terminate process	file size limit exceeded
SIGWINCH	discard signal	window size change

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process then it is delivered to the process. Signals may be delivered any time a process enters the operating system (e.g., during a system call, trap, or clock interrupt). If multiple signals are ready to be delivered at the same time, any signals that could be caused by traps are delivered first. Additional signals may be processed at the same time, with each appearing to interrupt the handlers for the previous signals before their first instructions. The set of pending signals is returned by the *sigpending()* function. When a caught signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a *sigprocmask* call is made). This mask is formed by taking the union of the current signal mask set, the signal to be delivered, and the signal mask associated with the handler to be invoked.

Once a signal handler is installed, it remains installed until another *sigaction()* call is made, or an *exec()* is performed.

After a *fork()* all signals and the signal mask, are inherited by the child.

Exec() functions reinstate the default action for all signals which were caught and resets all signals to be caught on the user stack. Ignored signals remain ignored and the signal mask remains the same.

NOTES

The mask specified in *act* is not allowed to block SIGKILL or SIGSTOP. This is ignored silently by the system.

Sigaction() is the POSIX.1 replacement for the traditional *signal()* call and mechanism. The *sigaction()* mechanism is more reliable than old-style signals; for a discussion, see a reference such as Zlotnick's *The POSIX.1 Standard: A Programmer's Guide*.

RETURN VALUES

A 0 value indicated that the call succeeded. A -1 return value indicates an error occurred and *errno* is set to indicated the reason.

ERRORS

Sigaction() will fail and no new signal handler will be installed if one of the following occurs:

- | | |
|----------|--|
| [EFAULT] | Either <i>act</i> or <i>oact</i> points to memory that is not a valid part of the process address space. |
| [EINVAL] | <i>Sig</i> is not a valid signal number. |
| [EINVAL] | An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP. |

NOTES

This implementation does not support the SA_RESTART bit for the *sa_flags* member of **struct sigaction**.

SEE ALSO

signal(), *sigpending()*, *sigprocmask()*, *sigsuspend()*

NAME

sigblock(), sigmask(), sigsetmask() – BSD4.3 compatibility signal functions

SYNOPSIS

```
#include <signal.h>

int sigblock (int mask)
int sigmask (int signum)
int sigsetmask (int mask)
```

DESCRIPTION

This interface is made obsolete by: *sigprocmask()*.

Sigblock() adds the signals specified in *mask* to the set of signals currently being blocked from delivery. Signals are blocked if the corresponding bit in *mask* is a 1. *Sigmask()* is provided to construct the mask for a given *signum*.

It is not possible to block SIGKILL or SIGSTOP; this restriction is silently imposed by the system.

RETURN VALUES

The functions *sigblock()* and *sigsetmask()* return the previous set of masked signals.

Sigmask() returns a mask that can be used to block the signal *signum*.

NOTES

The *sigblock()* call is provided for compatibility with BSD 4.3 only; portable code should use *sigprocmask()*.

The declaration of these APIs is visible only when *_ALL_SOURCE* is defined. These routines are only viable while the total number of signals supported on INTERIX is less than 32. If more signals are supported, then the signal mask will no longer fit in an *int*.

SEE ALSO

kill(), *sigprocmask()*, *sigaction()*, *sigemptyset()*, *signal()*, *sigsetmask()*

NAME

signal() – specify signal handling (C Standard version)

SYNOPSIS

```
#include <signal.h>
```

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

DESCRIPTION

The *signal()* call allows a process to catch, ignore, or to generate an interrupt on receiving a signal. (The exceptions are SIG_KILL and SIG_STOP, which cannot be caught or ignored.) The recommended call instead of *signal()* is the POSIX *sigaction()* call, which is slightly more robust.

The *sig* is a signal number (possible values are listed below or in the file <signal.h>).

The *func* argument is either a function to catch and handle the signal or one of these macros:

SIG_DFL Set the signal to the default action (listed below).

SIG_IGN Ignore the signal and discard pending instances. If SIG_IGN is not used, further occurrences of the signal are automatically blocked and *func* is called.

Signals allow the manipulation of a process from outside its domain as well as allowing the process to manipulate itself or copies of itself (children). There are two general types of signals: those that cause termination of a process and those that do not. Signals which cause termination of a program might result from an irrecoverable error or might be the result of a user at a terminal typing the ‘interrupt’ character. For example, signals are used when a process is stopped to access the process’s control terminal while in the background. Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals result in the termination of the process receiving them if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise.

For a list of signals defined in the file <signal.h>, see the *sigaction()* reference page.

The handled signal is unblocked with the function returns and the process continues from where it left off when the signal occurred.

When a process which has installed signal handlers forks, the child process inherits the signals. All caught signals may be reset to their default action by a call to the *execve()* function; ignored signals remain ignored.

RETURN VALUES

On success, *signal()* returns the previous action. Otherwise, it returns –1 and sets *errno* to indicate the error.

ERRORS

Signal will fail and no action will take place if one of the following occur:

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

SEE ALSO

kill

kill(), *sigaction()*, *sigprocmask()*, *sigsuspend()*, *setjmp()*

NAME

sighold(), sigignore(), sigpause(), sigrelse(), sigset() – System V signal management routines

SYNOPSIS

```
#include <signal.h>

int sighold(int signum)
int sigignore(int signum)
int sigpause(int signum)
int sigrelse(int signum)
void (*sigset(int signum, void (*handler)(int)))(int);
```

DESCRIPTION

These functions provide a simplified form of signal management; they are implemented using the *sigaction()* call.

Sigset() modifies the handling of the signal specified by *signum*. The signal may be any signal but SIGKILL and SIGSTOP. The *handler* is SIG_DFL (default), SIG_IGN (ignore), SIG_HOLD (block signal), or a pointer to a handler function. If *handler* points to a signal handler, *signum* is added to the calling process' signal mask before executing the handler. When the handler returns, the system restores the signal mask to its previous state. If the handler is SIG_HOLD, *sigset()* adds the signal to the process' signal mask but doesn't change how the signal is handled. If the handler is SIG_DFL or SIG_IGN, the signal is removed from the signal mask.

The *sighold()* function adds *signum* to the calling process' signal mask.

The *sigrelse()* function removes *signum* from the calling process' signal mask.

The *sigignore()* function sets the disposition of *signum* to SIG_IGN.

The *sigpause()* function removes the signal *signum* from the process' signal mask and then suspends the process until a signal is received.

RETURN VALUE

On success, *sigset()* returns the signal's previous disposition if the signal is not currently blocked. It returns SIG_HOLD if the signal is blocked. On failure, it returns SIG_ERR and sets *errno* to indicate the error.

The *sigpause()* function suspends execution of the process until a signal is received; then it returns -1 and sets *errno* to [EINTR].

The other functions return 0 on success. On failure, they return -1 and set *errno* to indicate the error.

ERRORS

The functions can fail if:

[EINVAL] The value of *signum* is an illegal signal number.

The *sigset()* and *sigignore()* calls will also fail if:

[EINVAL] They attempt to catch a signal that cannot be caught, or to ignore a signal that cannot be ignored.

NOTES

This implementation of *sigpause()* follows the behavior described in the *Single UNIX*

Specification, not BSD practice.

SEE ALSO

sigaction(), *signal()*

NAME

sigpending() – get pending signals

SYNOPSIS

```
#include <signal.h>

int sigpending(sigset_t *set)
```

DESCRIPTION

The *sigpending()* function returns a mask of the signals pending for delivery to the calling process in the location indicated by *set*. Signals may be pending because they are currently masked, or transiently before delivery (although the latter case is not normally detectable).

RETURN VALUES

A 0 value indicated that the call succeeded. A –1 return value indicates an error occurred and *errno* is set to indicated the reason.

ERRORS

The *sigpending()* function detects the following error:

[EFAULT] The value of *set* is an invalid pointer.

SEE ALSO

sigaction(), *sigprocmask()*

NAME

sigprocmask() – manipulate current signal mask

SYNOPSIS

```
#include <signal.h>
```

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset)
```

DESCRIPTION

The *sigprocmask()* function examines and / or changes the current signal mask (those signals that are blocked from delivery). Signals that belong to the current signal mask set are blocked.

The *oset* argument is set by the call to the *previous* value of the signal mask. If it is set to NULL, it is ignored.

The *set* argument points to a signal set that describes the changes to be made to the process's signal mask. If *set* is NULL, the mask isn't changed.

The *how* argument describes the change being requested (these values are defined in **<signal.h>**):

SIG_BLOCK

The new mask is the union of the current mask and the specified *set*.

SIG_UNBLOCK

The new mask is the intersection of the current mask and the complement of the specified *set*.

SIG_SETMASK

The current mask is replaced by the specified *set*.

If *set* is NULL, *how* is ignored. You can examine the current signal set by using NULL for *set* and loading the current signal set into *oset*.

You cannot block SIGKILL or SIGSTOP.

RETURN VALUES

A 0 value indicated that the call succeeded. A -1 return value indicates an error occurred and *errno* is set to indicated the reason.

ERRORS

The *sigprocmask()* call will fail and the signal mask will be unchanged if one of the following occurs:

[EINVAL] *how* has a value other than those listed here.

SEE ALSO

kill(), *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *sigsuspend()*

NAME

`sigemptyset()`, `sigfillset()`, `sigaddset()`, `sigdelset()`, `sigismember()` – manipulate signal sets

SYNOPSIS

```
#include <signal.h>

sigemptyset (sigset_t *set)
sigfillset (sigset_t *set)
sigaddset (sigset_t *set, int signo)
sigdelset (sigset_t *set, int signo)
sigismember (sigset_t *set, int signo)
```

DESCRIPTION

These functions manipulate signal sets stored in a **sigset_t**. You must call either `sigemptyset()` or `sigfillset()` for every object of type `sigset_t` before any other use of the object.

The `sigemptyset()` function initializes a signal set to be empty of the POSIX signals.

The `sigfillset()` function initializes a signal set to contain all of the POSIX signals.

The `sigaddset()` function adds the specified signal `signo` to the signal set.

The `sigdelset()` function deletes the specified signal `signo` from the signal set.

The `sigismember()` function returns whether a specified signal `signo` is contained in the signal set.

These functions are provided as macros in the include file **<signal.h>**. Actual functions are available if their names are undefined (with **#undef name**).

RETURN VALUES

The `sigismember()` function returns 1 if the signal is a member of the set, 0 otherwise.

The other functions return 0.

ERRORS

`sigaddset()`, `sigdelset()`, and `sigismember()` (that is, all except `sigemptyset()` and `sigfillset()`) may fail because:

[EINVAL] The specified `signo` is invalid.

SEE ALSO

`kill()`, `sigaction()`, `sigprocmask()`, `sigsuspend()`

NAME

sigsuspend() – wait for a signal

SYNOPSIS

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask)
```

DESCRIPTION

Sigsuspend() replaces the current signal mask with the set pointed to by *sigmask*, and then waits for a signal to arrive; on return it returns the previous set of masked signals. The signal mask set is usually empty to indicate that all signals are to be unblocked for the duration of the call.

In normal usage, a signal is blocked using *sigprocmask()* to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using *sigsuspend()* with the previous mask returned by *sigprocmask()*.

RETURN VALUES

If the *sigsuspend()* call returns, it returns -1 and sets *errno*.

ERRORS

The *sigsuspend()* function always terminates by being interrupted; it sets *errno* to this value:

[EINTR] An interrupt signal was received.

SEE ALSO

pause(), *sigprocmask()*, *sigaction()*

NAME

sigvec() – BSD software signal facility

SYNOPSIS

```
#include <signal.h>
struct sigvec {
    void    (*sa_handler)();
    int     sa_mask;
    int     sa_flags;
};

sigvec (int signum, struct sigvec *vec, struct sigvec *ovec)
```

DESCRIPTION

This interface is made obsolete by *sigaction()*. See the *sigaction()* reference page for a description of signals and signal handling and a list of signals.

Sigvec() assigns a handler for a specific signal. If *vec* is non-zero, it specifies a handler routine and mask to be used when delivering the specified signal. If *ovec* is non-zero, the routine returns the previous handling information for the signal.

Once a signal handler is installed, it remains installed until another *sigvec()* call is made, or until an *exec()* is performed. A signal-specific default action may be reset by setting *sa_handler* to SIG_DFL. The defaults are process termination; no action; stopping the process; or continuing the process. See the list in *sigaction()* for each signal's default action. If *sa_handler* is SIG_IGN, current and pending instances of the signal are ignored and discarded.

After a *fork()* all signals, the signal mask, the signal stack, and the restart/interrupt flags are inherited by the child.

The *exec()* family of routines reinstates the default action for all signals which were caught, and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that interrupt system calls continue to do so.

RETURN VALUES

A 0 value indicated that the call succeeded. A -1 return value indicates an error occurred and *errno* is set to indicated the reason.

ERRORS

Sigvec() will fail and no new signal handler will be installed if one of the following occurs:

[EFAULT]	Either <i>vec</i> or <i>ovec</i> points to memory that is not a valid part of the process address space.
[EINVAL]	<i>Signum</i> is not a valid signal number.
[EINVAL]	An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

NOTES

The mask cannot block SIGKILL or SIGSTOP. This is enforced silently by the system.

This implementation does not support such flags as SV_INTERRUPT or SV_ONSTACK. The only flag available to *sa_flags* is SA_NOCLDSTOP. See *sigaction()*.

SEE ALSO

kill

kill(), *setjmp()*, *sigaction()*, *sigblock()*, *signal()*, *sigprocmask()*, *sigsetmask()*,
sigsuspend()

NAME

sinh() – hyperbolic sine function

SYNOPSIS

```
#include <math.h>
```

```
double sinh (double x)
```

DESCRIPTION

The *sinh()* function computes the hyperbolic sine of *x*.

RETURN VALUES

The *sinh()* function returns the hyperbolic sine value unless the magnitude of *x* is too large, in which case it sets *errno*.

ERRORS

Sinh() can fail for the following reasons:

[ERANGE] The argument *x* is too large.

SEE ALSO

acos(), *asin()*, *atan()*, *atan2()*, *cos()*, *cosh()*, *math()*, *sin()*, *tan()*, *tanh()*

NAME

sin() – sine function

SYNOPSIS

```
#include <math.h>

double sin (double x)
```

DESCRIPTION

The *sin()* function computes the sine of *x* (measured in radians). A large magnitude argument may yield a result with little or no significance.

RETURN VALUES

The *sin()* function returns the sine value.

SEE ALSO

acos(), *asin()*, *atan()*, *atan2()*, *cos()*, *cosh()*, *sinh()*, *tan()*, *tanh()*

NAME

sleep() – suspend process execution for interval of seconds

SYNOPSIS

```
#include <unistd.h>

u_int sleep (u_int seconds)
```

DESCRIPTION

The *sleep()* function suspends execution of the calling process for at least *seconds* of time, or until a signal is delivered to the calling process which invokes a signal-catching function or terminates the process. System activity or time spent in processing the call may lengthen the sleep.

RETURN VALUES

sleep() returns 0 if its sleep was completed, or it returns the number of seconds left “unslept” if it was interrupted by a signal.

NAME

socket() – create an endpoint for communication

SYNOPSIS

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

int socket (int protofamily, int type, int protocol)
```

DESCRIPTION

Socket() creates an endpoint for communication and returns a descriptor.

The *protofamily* parameter specifies a communications protofamily 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>**. Currently, the sockets understand only one format:

AF_INET ARPA Internet protocols

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

```
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
```

- 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. They 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. Since only AF_INET protocols are supported in this implementation and because sockets of type SOCK_STREAM and SOCK_DGRAM only support one protocol each, use the value 0 for the *protocol*.

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. 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 an 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

The function returns a descriptor referencing the socket; if the function fails, it returns -1 and sets *errno* to indicate the error.

ERRORS

The *socket()* call fails if:

- [EPROTONOSUPPORT] The protocol type or the specified protocol is not supported within this protfamily.
- [EMFILE] The per-process descriptor table is full.
- [ENFILE] The system file table is full.
- [EACCESS] 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()*, *getsockname()*, *getsockopt()*, *listen()*, *read()*, *recv()*, *select()*, *send()*, *write()*

protocols

NAME

cbrt(), sqrt() – cube root and square root functions

SYNOPSIS

```
#include <math.h>

double cbrt (double x)
double sqrt (double x)
```

DESCRIPTION

The *cbrt()* function computes the cube root of *x*.

The *sqrt()* computes the non-negative square root of *x*.

RETURN VALUES

The *cbrt()* function returns the requested cube root. The *sqrt()* function returns the requested square root unless an error occurs. An attempt to take the

ERRORS

The *cbrt()* function can fail for the following reasons:

[EDOM] The argument *x* was *NaN*.

sqrt() function can fail for the following reasons:

[EDOM] The argument *x* was a negative number, or is *NaN*.

ERROR (due to Roundoff etc.)

The *cbrt()* function is accurate to within 0.7 *ulps*. *Sqrt()* on a machine that conforms to IEEE 754 is correctly rounded in accordance with the rounding mode in force; the error is less than half an *ulp* in the default mode (round-to-nearest). An *ulp* is one *Unit* in the *Last Place* carried.

SEE ALSO

math

NAME

stat(), fstat() – get file status

SYNOPSIS

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

int stat (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 pathname leading to the file must be searchable.

The *fstat()* obtains the same information about an open file identified 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;      device file resides on
    ino_t      st_ino;      file serial number
    mode_t     st_mode;     protection mode
    nlink_t    st_nlink;    number of hard links to the file
    uid_t      st_uid;      user-ID of owner
    gid_t      st_gid;      group-ID of owner
    off_t      st_size;     file size, in bytes
    long       st_blksize;   optimal file system I/O ops blocksize
    time_t     st_atime;     time of last access
    time_t     st_mtime;     time of last data modification
    time_t     st_ctime;     time of last file status change
};
```

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

st_atime Time when file data last accessed. Changed by the *utime()* and *read()* calls.

st_mtime

Time when file data last modified. Changed by the *utime()* and *write()* calls.

st_ctime Time when file status was last changed (file serial data modification). Changed by the *chmod()*, *chown()*, *link()*, *rename()*, *unlink()*, *utime()* and *write()* calls.

The size-related fields of the **struct stat** are:

st_blksize

The optimal I/O block size for the file.

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

Symbol	Meaning
S_IFMT	Type of file
S_IFIFO	named pipe (fifo)
S_IFCHR	character special
S_IFDIR	directory
S_IFBLK	block special
S_IFREG	regular
S_IFSOCK	file is socket
S_ISUID	set user ID on execution
S_ISGID	set group ID on execution
S_IRWXU	
S_IRUSR	read permission, owner
S_IWUSR	write permission, owner
S_IXUSR	execute / search permission, owner
S_IRWXG	
S_IRGRP	read permission, group
S_IWGRP	write permission, group
S_IXGRP	execute / search permission, group
S_IRWXO	
S_IROTH	read permission, file other
S_IWOTH	write permission, file other
S_IXOTH	execute / search permission, file other

For a list of access modes, see **<sys/stat.h>**, *access()* and *chmod()*.

Note that the S_ISUID and S_ISGID bits are ignored.

For a FIFO file, the *st_size* field contains its PIPE_BUF value.

Determining File Type

To determine the type of a file, several macros are defined in **<sys/stat.h>**; these all take the value of the *st_mode* from the **stat** structure as their argument:

S_ISDIR(<i>m</i>)	True if directory file, false otherwise
S_ISCHR(<i>m</i>)	True if character special file, false otherwise
S_ISBLK(<i>m</i>)	True if block special file, false otherwise
S_ISREG(<i>m</i>)	True if regular file, false otherwise
S_ISFIFO(<i>m</i>)	True if pipe or FIFO special file, false otherwise
S_ISSOCK(<i>m</i>)	True if a socket, false otherwise

RETURN VALUES

Upon successful completion, *stat()* returns 0. Otherwise it returns -1 and sets *errno* to indicate the error.

ERRORS

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

[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 {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
- [ENOENT] The named file does not exist.
- [EACCES] Search permission is denied for a component of the path prefix.
- [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.

NOTE

If the Win32 Read-Only attribute is set on a file, *stat()* reports that the file is unwriteable, no matter what the permissions are. To turn off the Win32 Read-Only attribute, you can use *chmod()* to enable a write permission; *chmod()* will unset the Read-Only attribute.

SEE ALSO

chmod(), *chown()*, *group_from_gid()*, *user_from_uid()*, *utime()*

NAME

fstatvfs(), statvfs() – get information about file system

SYNOPSIS

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

int fstatvfs(int fildes, struct statvfs *buf)
int statvfs(const char *path, struct statvfs *buf)
```

DESCRIPTION

These two functions return information about the file system by filling the **statvfs** structure referenced by *buf*. The *fstatvfs()* function returns information about the file referenced by the file descriptor *filde*s; the *statvfs()* function returns information about the file named by *path*.

The calling process does not need read, write, or execute permission on the file. For *statvfs()* the calling process must be able to search all of the directories in the file's absolute pathname, or an [EACCES] error will result.

The **statvfs** structure contains the following members:

```
#define MNAMELEN 128
struct statvfs {
    unsigned long    f_bsize;
    unsigned long    f_frsize;
    unsigned long    f_blocks;
    unsigned long    f_bfree;
    unsigned long    f_bavail;
    unsigned long    f_files;
    unsigned long    f_ffree;
    unsigned long    f_favail;
    unsigned long    f_fsid;
    unsigned long    f_flag;
    unsigned long    f_namemax;
    unsigned long    f_type;
    unsigned long    f_iosize;
    char              f_mntonname[MNAMELEN+1];
    char              f_mntfromname[MNAMELEN+1];
};
```

The members represent:

f_bsize	The block size on the filesystem.
f_frsize	Fundamental block size on the filesystem.
f_blocks	Total number of blocks on the filesystem in units of f_frsize .
f_bfree	Total number of free blocks.
f_bavail	Total number of free blocks available to a non-privileged process.
f_files	Total number of file serial numbers on the filesystem. Currently the implementation returns 0.
f_ffree	Total number of free file serial numbers on the filesystem. Currently the implementation returns 0.
f_favail	Total number of file serial numbers available to a non-privileged process. Currently the implementation returns 0.

f_fsid	ID of the filesystem.																				
f_flag	A bit mask of values describing the filesystem; defined values are: <table> <tr> <td>ST_RDONLY</td><td>A read-only filesystem.</td></tr> <tr> <td>ST_NOSUID</td><td>The filesystem does not support setuid / setgid semantics.</td></tr> </table>	ST_RDONLY	A read-only filesystem.	ST_NOSUID	The filesystem does not support setuid / setgid semantics.																
ST_RDONLY	A read-only filesystem.																				
ST_NOSUID	The filesystem does not support setuid / setgid semantics.																				
f_namemax	The maximum length of a filename on the filesystem.																				
f_type	The type of the filesystem, which is one of the following: <table> <tr> <td>ST_FSTYPE_CDFS</td><td>ISO 9660 CD-ROM drive.</td></tr> <tr> <td>ST_FSTYPE_CDROM</td><td>ISO 9660 CD-ROM drive.</td></tr> <tr> <td>ST_FSTYPE_FATFS</td><td>Microsoft DOS File Allocation Table filesystem.</td></tr> <tr> <td>ST_FSTYPE_HPFS</td><td>IBM's OS/2 High Performance File System.</td></tr> <tr> <td>ST_FSTYPE_ISO9660</td><td>ISO 9660 CD-ROM drive.</td></tr> <tr> <td>ST_FSTYPE_MSDOS</td><td>Microsoft DOS File Allocation Table filesystem.</td></tr> <tr> <td>ST_FSTYPE_NTFS</td><td>Microsoft's New Technology File System.</td></tr> <tr> <td>ST_FSTYPE_OFS</td><td>OFS Microsoft Windows NT's Object File System.</td></tr> <tr> <td>ST_FSTYPE_SAMBA</td><td>Samba SMB filesystem.</td></tr> <tr> <td>ST_FSTYPE_UNKNOWN</td><td>Unknown filesystem type.</td></tr> </table>	ST_FSTYPE_CDFS	ISO 9660 CD-ROM drive.	ST_FSTYPE_CDROM	ISO 9660 CD-ROM drive.	ST_FSTYPE_FATFS	Microsoft DOS File Allocation Table filesystem.	ST_FSTYPE_HPFS	IBM's OS/2 High Performance File System.	ST_FSTYPE_ISO9660	ISO 9660 CD-ROM drive.	ST_FSTYPE_MSDOS	Microsoft DOS File Allocation Table filesystem.	ST_FSTYPE_NTFS	Microsoft's New Technology File System.	ST_FSTYPE_OFS	OFS Microsoft Windows NT's Object File System.	ST_FSTYPE_SAMBA	Samba SMB filesystem.	ST_FSTYPE_UNKNOWN	Unknown filesystem type.
ST_FSTYPE_CDFS	ISO 9660 CD-ROM drive.																				
ST_FSTYPE_CDROM	ISO 9660 CD-ROM drive.																				
ST_FSTYPE_FATFS	Microsoft DOS File Allocation Table filesystem.																				
ST_FSTYPE_HPFS	IBM's OS/2 High Performance File System.																				
ST_FSTYPE_ISO9660	ISO 9660 CD-ROM drive.																				
ST_FSTYPE_MSDOS	Microsoft DOS File Allocation Table filesystem.																				
ST_FSTYPE_NTFS	Microsoft's New Technology File System.																				
ST_FSTYPE_OFS	OFS Microsoft Windows NT's Object File System.																				
ST_FSTYPE_SAMBA	Samba SMB filesystem.																				
ST_FSTYPE_UNKNOWN	Unknown filesystem type.																				
f_iosize	Optimal transfer block size.																				
f_mntonname	The mountpoint for the filesystem. This is normally a pathname corresponding to a drive letter, such as //C or //F.																				
f_mntfromname	The mounted filesystem.																				

RETURN VALUE

On success, each function returns 0. Otherwise, it returns -1 and sets *errno* to indicate the error.

ERRORS

Both functions will fail if:

[EIO]	An I/O error occurred while reading from or writing to the file system.
[EINTR]	A signal was caught while the function was executing.

In addition, *fstatvfs()* will fail if:

[EBADF]	The descriptor <i>fildev</i> is invalid.
---------	--

The *statvfs()* function will fail if:

[EACCES]	The calling process did not have search permission on a directory in the file's pathname.
[ENAMETOOLONG]	A component of the file's pathname exceeded {NAME_MAX} characters, or the entire pathname exceeded {PATH_MAX} characters.
[ENOENT]	The file does not exist.
[ENOTDIR]	Some component of the <i>path</i> prefix is not a directory.

SEE ALSO

stat()

NAME

stdarg – variable argument lists

SYNOPSIS

```
#include <stdarg.h>

void va_start (va_list ap, last)
type va_arg (va_list ap, type)
void va_end (va_list ap)
```

DESCRIPTION

A function may be called with a varying number of arguments of varying types. The include file **<stdarg.h>** declares a type (**va_list**) and defines three macros for stepping through a list of arguments whose number and types are not known to the called function.

The called function must declare an object of type **va_list** which is used by the macros *va_start()*, *va_arg()*, and *va_end()*.

The *va_start()* macro initializes *ap* for subsequent use by *va_arg()* and *va_end()*, and must be called first.

The parameter *last* is the name of the last parameter before the variable argument list, i.e. the last parameter of which the calling function knows the type.

Because the address of this parameter is used in the *va_start()* macro, it should not be declared as a register variable, or as a function or an array type.

The *va_start()* macro returns no value.

The *va_arg()* macro expands to an expression that has the type and value of the next argument in the call. The parameter *ap* is the **va_list** *ap* initialized by *va_start()*. Each call to *va_arg()* modifies *ap* so that the next call returns the next argument. The parameter *type* is a type name specified so that the type of a pointer to an object that has the specified type can be obtained simply by adding a * to *type*.

If there is no next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), random errors will occur.

The first use of the *va_arg()* macro after that of the *va_start()* macro returns the argument after *last*. Successive invocations return the values of the remaining arguments.

The *va_end()* macro handles a normal return from the function whose variable argument list was initialized by *va_start()*.

The *va_end()* macro returns no value.

EXAMPLES

The function *foo* takes a string of format characters and prints out the argument associated with each format character based on the type.

```

void foo(char *fmt, ...)
{
    va_list ap;
    int d;
    char c, *p, *s;
    va_start(ap, fmt);
    while (*fmt)
        switch(*fmt++) {
            case 's':                /* string */
                s = va_arg(ap, char *);
                printf("string %s\n", s);
                break;
            case 'd':                /* int */
                d = va_arg(ap, int);
                printf("int %d\n", d);
                break;
            case 'c':                /* char */
                c = va_arg(ap, char);
                printf("char %c\n", c);
                break;
        }
    va_end(ap);
}

```

BUGS

Unlike the *varargs* macros, the *stdarg* macros do not permit programmers to code a function with no fixed arguments. This problem generates work mainly when converting *varargs* code to *stdarg* code, but it also creates difficulties for variadic functions that wish to pass all of their arguments on to a function that takes a **va_list** argument, such as *vfprintf()*.

NAME

strcasecmp(), strncasecmp() – compare strings, ignoring case

SYNOPSIS

```
#include <strings.h>

int strcasecmp (const char *s1, const char *s2)
int strncasecmp (const char *s1, const char *s2, size_t len)
```

DESCRIPTION

The *strcasecmp()* and *strncasecmp()* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, if *s1* is lexicographically greater than, equal to, or less than *s2* (respectively), after translating each corresponding character to lower-case. The strings themselves are not modified. The comparison is done using unsigned characters, so that `\200` is greater than `\0`.

The *strncasecmp()* compares at most *len* characters.

SEE ALSO

bcmp(), *memcmp()*, *strcmp()*, *strcoll()*, *strxfrm()*

NAME

strcat(), strncat() – concatenate strings

SYNOPSIS

```
#include <string.h>

char * strcat (char *s, const char *append)
char * strncat (char *s, const char *append, size_t count)
```

DESCRIPTION

The *strcat()* and *strncat()* functions append a copy of the null-terminated string *append* to the end of the null-terminated string *s*, then add a terminating `\0`. The string *s* must have sufficient space to hold the result.

The *strncat()* function appends not more than *count* characters.

RETURN VALUES

The *strcat()* and *strncat()* functions return the pointer *s*.

SEE ALSO

bcopy(), *memcpy()*, *memmove()*, *strcpy()*

NAME

strchr() – locate character in string

SYNOPSIS

```
#include <string.h>
```

```
char * strchr (const char *s, int c)
```

DESCRIPTION

The *strchr()* function locates the first occurrence of *c* in the string pointed to by *s*. The terminating NUL character is considered part of the string. If *c* is `\0`, *strchr()* locates the terminating `\0`.

RETURN VALUES

The function *strchr()* returns a pointer to the located character, or `NULL` if the character does not appear in the string.

SEE ALSO

memchr(), *strcspn()*, *strpbrk()*, *strrchr()*, *strsep()*, *strspn()*, *strstr()*, *strtok()*

NAME

strcmp(), strncmp() – compare strings

SYNOPSIS

```
#include <string.h>

int strcmp (const char *s1, const char *s2)
int strncmp (const char *s1, const char *s2, size_t len)
```

DESCRIPTION

The *strcmp()* and *strncmp()* functions lexicographically compare the NUL-terminated strings *s1* and *s2*.

RETURN VALUES

The *strcmp()* and *strncmp()* return an integer greater than, equal to, or less than 0, according as the string *s1* is greater than, equal to, or less than the string *s2*. The comparison is done using unsigned characters, so that \200 is greater than \0.

Unlike *strcmp()*, *strncmp()* compares not more than *len* characters.

SEE ALSO

memcmp(), *strcoll()*, *strxfrm()*

NAME

strcoll() – compare strings according to current collation

SYNOPSIS

```
#include <string.h>
```

```
int strcoll (const char *s1, const char *s2)
```

DESCRIPTION

The *strcoll()* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according as *s1* is greater than, equal to, or less than *s2*.

SEE ALSO

memcmp(), *setlocale()*, *strcmp()*, *strxfrm()*

NAME

strcpy(), strncpy() – copy strings

SYNOPSIS

```
#include <string.h>

char * strcpy (char *dst, const char *src)
char * strncpy (char *dst, const char *src, size_t len)
```

DESCRIPTION

The *strcpy()* and *strncpy()* functions copy the string *src* to *dst* (including the terminating \0 character).

The *strncpy()* copies not more than *len* characters into *dst*, appending \0 characters if *src* is less than *len* characters long, and terminating *dst* if *src* is more than *len* characters long.

RETURN VALUES

The *strcpy()* and *strncpy()* functions return *dst*.

EXAMPLES

The following sets “chararray” to “abc\0\0\0:”

```
(void)strncpy(chararray, "abc", 6).
```

The following sets “chararray” to “abcdef:”

```
(void)strncpy(chararray, "abcdefgh", 6);
```

SEE ALSO

bcopy(), *memcpy()*, *memmove()*

NAME

strcspn() – span the complement of a string

SYNOPSIS

```
#include <string.h>
```

```
size_t strcspn (const char *s, const char *charset)
```

DESCRIPTION

The *strcspn()* function spans the initial part of the null-terminated string *s* as long as the characters from *s* do not occur in string *charset* (it spans the *complement* of *charset*).

RETURN VALUES

The *strcspn()* function returns the number of characters spanned.

SEE ALSO

memchr(), *strchr()*, *strpbrk()*, *strrchr()*, *strsep()*, *strspn()*, *strstr()*, *strtok()*

NAME

strdup() – save a copy of a string

SYNOPSIS

```
#include <string.h>
```

```
char * strdup (const char *str)
```

DESCRIPTION

The *strdup()* function allocates sufficient memory for a copy of the string *str*, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function *free()*.

If insufficient memory is available, NULL is returned.

SEE ALSO

malloc(), *free()*

NAME

`perror()`, `strerror()`, `hstrerror()`, – system error messages

SYNOPSIS

```
#include <stdio.h>

void perror (const char *string)

#include <string.h>

char * strerror (int errnum)
char * hstrerror (int errnum)
```

DESCRIPTION

The `strerror()` and `perror()` functions look up the error message string corresponding to an error number.

The `strerror()` function accepts an error number argument *errnum* and returns a pointer to the corresponding message string.

The `perror()` function finds the error message corresponding to the current value of the global variable *errno* (see the *errors* page) and writes it, followed by a newline, to the standard error file descriptor. If the argument *string* is non-NULL, it is prepended to the message string and separated from it by a colon and space (:). If *string* is NULL, only the error message string is printed.

If *errnum* is not a recognized error number, the error message string will contain “Unknown error: ” followed by the error number in decimal.

The `hstrerror()` call is a **#define** for `strerror()`, provided for convenience in working with sockets.

SEE ALSO

errno

NAME

strftime() – format date and time

SYNOPSIS

```
#include <time.h>

size_t strftime (char *buf, size_t maxsize, const char *format,
                const struct tm *timeptr)
```

DESCRIPTION

The *strftime()* function formats the information from *timeptr* into the buffer *buf* according to the string pointed to by *format*.

The *format* string consists of zero or more conversion specifications and ordinary characters. All ordinary characters are copied directly into the buffer. A conversion specification consists of a percent sign % and one other character.

No more than *maxsize* characters will be placed into the array. If the total number of resulting characters, including the terminating \0 character, is not more than *maxsize*, *strftime()* returns the number of characters in the array, not counting the terminating NUL. Otherwise, zero is returned.

Each conversion specification is replaced by the characters as follows which are then copied into the buffer.

- %A is replaced by the full weekday name.
- %a is replaced by the abbreviated weekday name, where the abbreviation is the first three characters.
- %B is replaced by the full month name.
- %b or %h is replaced by the abbreviated month name, where the abbreviation is the first three characters.
- %C is equivalent to %a%b%e (the format produced by *asctime()*).
- %c is equivalent to %m/%d/%y.
- %D is replaced by the date in the format mm/dd/yy.
- %d is replaced by the day of the month as a decimal number (01–31).
- %e is replaced by the day of month as a decimal number (1–31); single digits are preceded by a blank.
- %H is replaced by the hour (24-hour clock) as a decimal number (00–23).
- %I is replaced by the hour (12-hour clock) as a decimal number (01–12).
- %j is replaced by the day of the year as a decimal number (001–366).
- %k is replaced by the hour (24-hour clock) as a decimal number (0–23); single digits are preceded by a blank.
- %l is replaced by the hour (12-hour clock) as a decimal number (1–12); single digits are preceded by a blank.

%M	is replaced by the minute as a decimal number (00–59).
%m	is replaced by the month as a decimal number (01–12).
%n	is replaced by a newline.
%p	is replaced by either “AM” or “PM” as appropriate.
%R	is equivalent to %H:%M
%r	is equivalent to %I:%M:%S%p.
%t	is replaced by a tab.
%S	is replaced by the second as a decimal number (00–60).
%T or %X is equivalent to %H:%M:%S.	
%U	is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00–53).
%W	is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (00–53).
%w	is replaced by the weekday (Sunday as the first day of the week) as a decimal number (0–6).
%x	is equivalent to %m / %d / %y %H:%M:%S.
%Y	is replaced by the year with century as a decimal number.
%y	is replaced by the year without century as a decimal number (00–99).
%Z	is replaced by the time zone name.
%%	is replaced by %.

SEE ALSO

printf, *date*
ctime(), *printf()*

NAME

stripdomainprefix() – remove leading domain names from strings

SYNOPSIS

```
#include <opennt/security.h>

char * stripdomainprefix (const char *fullname,
                          const char *domainname,
                          int casesensitive);
```

DESCRIPTION

The *stripdomainprefix()* utility accepts a string containing a *domain / name* pair; the separator can be +, /, \, or :.

If the domain in the *fullname* matches the value of *domainname*, *stripdomainprefix()* strips off the domain and returns the string *name*. If it doesn't match, *fullname* is returned.

If *domainname* is NULL, any domain prefixes will be stripped.

If *casesensitive* is 0, the match between *domainname* and the domain is not case-sensitive.

If it is non-zero, the match is case-sensitive.

RETURN VALUES

Returns a pointer to the user name if the domain name matches, or to *fullname* if the name doesn't match.

NAME

strlen() – find length of string

SYNOPSIS

```
#include <string.h>

size_t strlen (const char *s)
```

DESCRIPTION

The *strlen()* function computes the length of the string *s*.

RETURN VALUES

The *strlen()* function returns the number of characters that precede the terminating NUL character.

NAME

strmode() – convert file serial number status information into a symbolic string

SYNOPSIS

```
#include <string.h>
void strmode (mode_t mode, char *bp)
```

DESCRIPTION

The *strmode()* function converts a file *mode* (the type and permission information associated with a file, see *stat()*) into a symbolic string which is stored in the location referenced by *bp*. This stored string is eleven characters in length plus a trailing NULL.

The first character is the file type, and will be one of the following:

- regular file
- b block special
- c character special
- d directory
- l symbolic link
- p fifo
- s socket
- ? unknown type

The next nine characters encode three sets of permissions, in three characters each. The first three characters are the permissions for the owner of the file, the second three for the group the file belongs to, and the third for the “other”, or default, set of users.

Permission checking is done as specifically as possible. If read permission is denied to the owner of a file in the first set of permissions, the owner of the file will not be able to read the file. This is true even if the owner is in the file’s group and the group permissions allow reading or the “other” permissions allow reading.

If the first character of the three character set is an “r”, the file is readable for that set of users; if a dash “–”, it is not readable.

If the second character of the three character set is a “w”, the file is writable for that set of users; if a dash “–”, it is not writable.

The third character is the first of the following characters that apply:

- | | |
|---|--|
| S | If the character is part of the owner permissions and the file is not executable or the directory is not searchable by the owner, and the set-user-id bit is set. |
| S | If the character is part of the group permissions and the file is not executable or the directory is not searchable by the group, and the set-group-id bit is set. |
| T | If the character is part of the other permissions and the file is not executable or the directory is not searchable by others. |
| s | If the character is part of the owner permissions and the file is executable or the directory searchable by the owner, and the set-user-id bit is set. |
| s | If the character is part of the group permissions and the file is executable or the directory searchable by the group, and the set-group-id bit is set. |

- t If the character is part of the other permissions and the file is executable or the directory searchable by others.
- x The file is executable or the directory is searchable.
- None of the above apply.

The last character is a plus sign “+” if any there are any alternate or additional access control methods associated with the file, otherwise it will be a space.

RETURN VALUES

The *strmode()* function always returns 0.

SEE ALSO

chmod, *find*

stat(), *setmode()*

NAME

strpbrk() – locate multiple characters in string

SYNOPSIS

```
#include <string.h>
```

```
char * strpbrk (const char *s, const char *charset)
```

DESCRIPTION

The *strpbrk()* function locates in the NUL-terminated string *s* the first occurrence of any character in the string *charset* and returns a pointer to this character. If no characters from *charset* occur anywhere in *s*, *strpbrk()* returns NULL.

RETURN CODES

On success, *strpbrk()* returns a pointer to a character. On failure, it returns NULL.

SEE ALSO

memchr(), *strchr()*, *strcspn()*, *strrchr()*, *strsep()*, *strspn()*, *strstr()*, *strtok()*

NAME

strrchr() – locate character in string

SYNOPSIS

```
#include <string.h>
```

```
char * strrchr (const char *s, int c)
```

DESCRIPTION

The *strrchr()* function locates the last occurrence of *c* (converted to a char) in the string *s*. If *c* is `\0`, *strrchr()* locates the terminating `\0`.

RETURN VALUES

The *strrchr()* function returns a pointer to the character, or a null pointer if *c* does not occur anywhere in *s*.

SEE ALSO

memchr(), *strchr()*, *strcspn()*, *strpbrk()*, *strsep()*, *strspn()*, *strstr()*, *strtok()*

NAME

strsep – separate strings

SYNOPSIS

```
#include <string.h>
char * strsep (char **stringp, char *delim)
```

DESCRIPTION

The *strsep()* function locates, in the string referenced by **stringp*, the first occurrence of any character in the string *delim* (or the terminating `\0` character) and replaces it with a `\0`. The location of the next character after the delimiter character (or NULL, if the end of the string was reached) is stored in **stringp*. The original value of **stringp* is returned.

An “empty” field, i.e. one caused by two adjacent delimiter characters, can be detected by comparing the location referenced by the pointer returned in **stringp* to `\0`.

If **stringp* is initially NULL, *strsep()* returns NULL.

EXAMPLES

The following uses *strsep()* to parse a string, containing tokens delimited by white space, into an argument vector:

```
char **ap, *argv[10], *inputstring;
for (ap = argv; (*ap = strsep(&inputstring, " \t")) != NULL;)
    if (**ap != '\0')
        ++ap;
```

NOTES

The *strsep()* function is intended as a replacement for the *strtok()* function. While the *strtok()* function should be preferred for portability reasons (it conforms to ISO C) it is unable to handle empty fields, i.e. detect fields delimited by two adjacent delimiter characters, or to be used for more than a single string at a time. The *strsep()* function first appeared in 4.4BSD.

SEE ALSO

strtok()

NAME

strspn() – span a string

SYNOPSIS

```
#include <string.h>
```

```
size_t strspn (const char *s, const char *charset)
```

DESCRIPTION

The *strspn()* function spans the initial part of the null-terminated string *s* as long as the characters from *s* occur in string *charset*.

RETURN VALUES

The *strspn()* function returns the number of characters spanned.

SEE ALSO

memchr(), *strchr()*, *strcspn()*, *strpbrk()*, *strrchr()*, *strsep()*, *strstr()*, *strtok()*

NAME

strstr() – locate a substring in a string

SYNOPSIS

```
#include <string.h>
```

```
char * strstr (const char *big, const char *little)
```

DESCRIPTION

The *strstr()* function locates the first occurrence of the null-terminated string *little* in the null-terminated string *big*. If *little* is the empty string, *strstr()* returns *big*; if *little* occurs nowhere in *big*, *strstr()* returns NULL; otherwise *strstr()* returns a pointer to the first character of the first occurrence of *little*.

SEE ALSO

memchr(), *strchr()*, *strcspn()*, *strpbrk()*, *strrchr()*, *strsep()*, *strspn()*, *strtok()*

NAME

strtod() – convert ASCII string to double

SYNOPSIS

```
#include <stdlib.h>
```

```
double strtod (const char *nptr, char **endptr)
```

DESCRIPTION

The *strtod()* function converts the initial portion of the string pointed to by *nptr* to **double** representation.

The expected form of the string is an optional plus (+) or minus sign (–) followed by a sequence of digits optionally containing a decimal-point character, optionally followed by an exponent. An exponent consists of an “E” or “e”, followed by an optional plus or minus sign, followed by a sequence of digits.

Leading white-space characters in the string (as defined by the *isspace()* function) are skipped.

RETURN VALUES

The *strtod()* function returns the converted value, if any.

If *endptr* is not NULL, a pointer to the character after the last character used in the conversion is stored in the location referenced by *endptr*.

If no conversion is performed, zero is returned and the value of *nptr* is stored in the location referenced by *endptr*.

If the correct value would cause overflow, plus or minus {HUGE_VAL} is returned (according to the sign of the value), and [ERANGE] is stored in *errno*. If the correct value would cause underflow, zero is returned and [ERANGE] is stored in *errno*.

ERRORS

[ERANGE] Overflow or underflow occurred.

SEE ALSO

atof(), *atoi()*, *atol()*, *strtol()*, *strtoul()*

NAME

strtok() – string tokens

SYNOPSIS

```
#include <string.h>
```

```
char * strtok (char *str, const char *sep)
```

DESCRIPTION

The *strtok()* function is used to isolate sequential tokens in a null-terminated string, *str*. These tokens are separated in the string by at least one of the characters in *sep*. The first time that *strtok()* is called, *str* should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead. The separator string, *sep*, 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 token itself with a NUL character. When no more tokens remain, a null pointer is returned.

SEE ALSO

memchr(), *strchr()*, *strcspn()*, *strpbrk()*, *strrchr()*, *strsep()*, *strspn()*, *strstr()*

NAME

strtol() – convert string value to a long integer

SYNOPSIS

```
#include <stdlib.h>
#include <limits.h>

long strtol (char *nptr, char **endptr, int base)
```

DESCRIPTION

The *strtol()* function converts the string in *nptr* to a **long** value.

The string may begin with an arbitrary amount of white space (as determined by *isspace()*) followed by a single optional + or - sign. If *base* is zero or 16, the string may then include a 0x prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is 0, in which case it is taken as 8 (octal).

The remainder of the string is converted to a **long** value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter A or a represents 10, B or b represents 11, and so forth, with Z or z representing 35.)

If *endptr* is not NULL, *strtol()* stores the address of the first invalid character in **endptr*. If there were no digits at all, however, *strtol()* stores the original value of *nptr* in **endptr*. (Thus, if **nptr* is not \0 but ***endptr* is \0 on return, the entire string was valid.)

RETURN VALUES

The *strtol()* function returns the result of the conversion, unless the value would underflow or overflow. If an underflow occurs, *strtol()* returns {LONG_MIN.} If an overflow occurs, *strtol()* returns LONG_MAX. In both cases, *errno* is set.

ERRORS

The *strtol()* function can fail for any of these reasons:

[ERANGE]	The given string was out of range; the value converted has been clamped.
----------	--

SEE ALSO

atof(), *atoi()*, *atol()*, *strtod()*, *strtoul()*

NAME

strtoul() – convert a string to an unsigned long

SYNOPSIS

```
#include <stdlib.h>
```

```
#include <limits.h>
```

```
unsigned long strtoul (const char *nptr, char **endptr, int base)
```

DESCRIPTION

The *strtoul()* function converts the string in *nptr* to an value. The conversion is done according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by *isspace()*) followed by a single optional + or - sign. If *base* is zero or 16, the string may then include a 0x prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is 0, in which case it is taken as 8 (octal).

The remainder of the string is converted to an **unsigned long** value in the obvious manner, stopping at the end of the string or at the first character that does not produce a valid digit in the given base. (In bases above 10, the letter A in either upper or lower case represents 10, B represents 11, and so forth, with Z representing 35.)

If *endptr* is not nil, *strtoul()* stores the address of the first invalid character in **endptr*. If there were no digits at all, however, *strtoul()* stores the original value of *nptr* in **endptr*. (Thus, if **nptr* is not \0 but ***endptr* is \0 on return, the entire string was valid.)

RETURN VALUES

The *strtoul()* function returns either the result of the conversion or, if there was a leading minus sign, the negation of the result of the conversion, unless the original (non-negated) value would overflow; in the latter case, *strtoul()* returns {ULONG_MAX} and sets the global variable *errno* to [ERANGE].

ERRORS

[ERANGE]	The given string was out of range; the value converted has been clamped.
----------	--

SEE ALSO

strtol()

NAME

strxfrm() – transform a string under locale

SYNOPSIS

```
#include <string.h>
```

```
size_t strxfrm (char *dst, const char *src, size_t n)
```

DESCRIPTION

The *strxfrm()* function transforms a string, *src*, to the current locale and stores at most *n* bytes of it in *dst*. The transformed string will still compare properly

This function is a subset of *strcoll()*'s functionality, and is primarily a tool for efficient comparisons. When many comparisons must be made, using *strxfrm()* and *strcmp()* can be faster than using *strcoll()*.

In the C and POSIX locales, it is nearly identical to *strncmp()*, but the return value is different.

RETURN VALUE

strxfrm() returns the length of the transformed string.

SEE ALSO

memcmp(), *strcasecmp()*, *strcmp()*, *strcoll()*

NAME

swab() – swap adjacent bytes

SYNOPSIS

```
#include <string.h>
```

```
void swab (const void *src, void *dst, size_t len)
```

DESCRIPTION

The function *swab()* copies *len* bytes from the location referenced by *src* to the location referenced by *dst*, swapping adjacent bytes.

The argument *len* must be even number.

SEE ALSO

bzero(), *memset()*

NAME

sysconf() – get configurable system variables

SYNOPSIS

```
#include <unistd.h>

long sysconf (int name)
```

DESCRIPTION

This interface is defined by POSIX.1-1988 standard.

The *sysconf()* function provides a method for applications to determine the current value of a configurable system limit or option variable. The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file **<unistd.h>**.

The definitive list of symbolic constants is in the include file. As of this writing, the available values are as follows:

sysconf() Name	Description
_SC_2_C_BIND	Whether system provides POSIX.2 C language bindings
_SC_2_C_DEV	Whether system supports the POSIX.2 C-Language Development Utilities Option
_SC_2_CHAR_TERM	Whether system supports at least one terminal type capable of supporting all operations described in POSIX.2
_SC_2_FORT_RUN	Whether system supports POSIX.2 FORTRAN Runtime Utilities Option
_SC_2_FORT_DEV	Whether system supports POSIX.2 FORTRAN Development Utilities Option
_SC_2_LOCALEDEF	Whether system supports the creation of locales
_SC_2_SW_DEV	Whether system supports POSIX.2 Software Development Portability Utilities Option
_SC_2_UPE	Whether system supports POSIX.2 User Portability Utilities Option
_SC_2_VERSION	Version of ISO / IEC 9945 (POSIX 1003.2) with which the system attempts to comply.
_SC_ARG_MAX	Maximum bytes of argument to <i>exec()</i> .
_SC_BC_BASE_MAX	Maximum obase value allowed by <i>bc</i>
_SC_BC_DIM_MAX	Maximum number of elements in a <i>bc</i> array
_SC_BC_SCALE_MAX	Maximum scale value allowed by <i>bc</i>
_SC_BC_STRING_MAX	Maximum length of a string constant in <i>bc</i>

<i>sysconf()</i> Name	Description
_SC_CHILD_MAX	Maximum number of simultaneous processes per user ID.
_SC_CLK_TCK	Number of micro-seconds per hz tick.
_SC_COLL_WEIGHTS_MAX	In a locale definition file, the maximum number of weights that can be assigned to the LC_COLLATE order keyword
_SC_EXPR_NEST_MAX	Maximum number of expressions that can be nested within parentheses by <i>expr</i>
_SC_JOB_CONTROL	Return 1 if job control is available on this system, otherwise -1.
_SC_LINE_MAX	Maximum length in bytes, including trailing newline, of an input line for a text-processing utility
_SC_NGROUPS_MAX	Maximum number of supplemental groups.
_SC_OPEN_MAX	Maximum number of open files per user ID.
_SC_PAGE_SIZE	The virtual memory page size.
_SC_PAGESIZE	The virtual memory page size.
_SC_RE_DUP_MAX	Maximum number of repetitions of a BRE allowed in $\{m,n\}$ notation
_SC_SAVED_IDS	Returns 1 if saved set-group and saved set-user ID is available, otherwise -1.
_SC_STREAM_MAX	Minimum maximum number of streams that a process may have open at any one time.
_SC_TZNAME_MAX	Minimum maximum number of types supported for the name of a timezone.
_SC_VERSION	The version of ISO / IEC 9945 (POSIX 1003.1) with which the system attempts to comply.

RETURN VALUES

If the call to *sysconf* is not successful, the function returns -1 and sets *errno* appropriately. If the variable is associated with functionality that isn't supported or if the variable is not valid, the function returns -1 and does *not* set *errno*.

Otherwise, the current variable value is returned.

ERRORS

The *sysconf()* function may fail and set *errno* for any of these errors:

[EINVAL] The value of the *name* argument is invalid.

NAME

syslog(), vsyslog(), openlog(), closelog(), setlogmask() – control system log

SYNOPSIS

```
#include <syslog.h>
#include <stdarg.h>

void syslog (int priority, const char *message, ...)
void vsyslog (int priority, const char *message, va_list args)
void openlog (const char *ident, int logopt, int facility)
void closelog (void)
int setlogmask (int maskpri)
```

DESCRIPTION

The `syslog()` function writes *message* to the system message logger. The message is then written to log files, logged-in users, or forwarded to other machines as appropriate. (See *syslogd*.) If the *syslogd* program is not running, the message is written to the file

\$OPENNT_ROOT/var/adm/log/logger.

The message is identical to a `printf()` format string, except that **%m** is replaced by the current error message (as denoted by the global variable *errno*; see *strerror()*.) A trailing newline is added if none is present.

The `vsyslog()` function is an alternate form in which the arguments have already been captured using the variable-length argument facilities of *stdarg()* (note that it is *not* *varargs*, as in many implementations).

The message is tagged with *priority*. Priorities are encoded as a *facility* and a *level*. The facility describes the part of the system generating the message. The level is selected from the following *ordered* (high to low) list:

LOG_EMERG

A panic condition. This is normally broadcast to all users.

LOG_ALERT

A condition that should be corrected immediately, such as a corrupted system database.

LOG_CRIT

Critical conditions, e.g., hard device errors.

LOG_ERR

Errors.

LOG_WARNING

Warning messages.

LOG_NOTICE

Conditions that are not error conditions, but should possibly be handled specially.

LOG_INFO

Informational messages.

LOG_DEBUG

Messages that contain information normally of use only when debugging a program.

The `openlog()` function provides for more specialized processing of the messages sent by `syslog()` and `vsyslog()`. The parameter *ident* is a string that will be prepended to every message. The *logopt* argument is a bit field specifying logging options, which is formed

by ORing one or more of the following values:

LOG_NDELAY

Open the connection to *syslogd* immediately. Normally the open is delayed until the first message is logged. Useful for programs that need to manage the order in which file descriptors are allocated.

LOG_PERROR

Write the message to standard error output as well to the system log.

LOG_PID

Log the process id with each message: useful for identifying instantiations of daemons.

The *facility* parameter encodes a default facility to be assigned to all messages that do not have an explicit facility encoded:

LOG_AUTH

The authorization system: *login* etc.

LOG_AUTHPRIV

The same as LOG_AUTH, but logged to a file readable only by selected individuals.

LOG_CRON

The clock daemon.

LOG_DAEMON

System daemons, such as *routed*, that are not provided for explicitly by other facilities.

LOG_KERN

Messages generated by the kernel. These cannot be generated by any user processes.

LOG_FTP

The file transfer protocol (FTP) daemon.

LOG_LPR

The line printer spooling system: *lp* etc.

LOG_MAIL

The mail system.

LOG_NEWS

The network news system.

LOG_SYSLOG

Messages generated internally by *syslogd*.

LOG_USER

Messages generated by random user processes. This is the default facility identifier if none is specified.

LOG_UUCP

The uucp system.

LOG_LOCAL0

Reserved for local use. Similarly for LOG_LOCAL1 through LOG_LOCAL7.

The *closelog()* function can be used to close the log file.

The *setlogmask()* function sets the log priority mask to *maskpri* and returns the previous mask. Calls to *syslog()* with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro *LOG_MASK()**pri*; the mask for all priorities up to and including *toppri* is given by the macro *LOG_UPTO()**toppri*. The default allows all priorities to be logged.

RETURN VALUES

The routines *closelog()*, *openlog()*, *syslog()* and *vsyslog()* return no value.

The routine *setlogmask()* always returns the previous log mask level.

EXAMPLES

```
syslog(LOG_ALERT, "who: internal error 23");
openlog("ftpd", LOG_PID, LOG_DAEMON);
setlogmask(LOG_UPTO(LOG_ERR));
syslog(LOG_INFO, "Connection from host %d", CallingHost);
syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```

SEE ALSO

logger, *syslogd*

NAME

system() – pass a command to the shell

SYNOPSIS

```
#include <stdlib.h>
```

```
int system (const char *string)
```

DESCRIPTION

The *system()* function hands the argument *string* to the command interpreter *sh*. The calling process waits for the shell to finish executing the command, ignoring SIGINT and SIGQUIT, and blocking SIGCHLD.

If *string* is a NULL pointer, *system()* will return non-zero if the command interpreter *sh* is available, and zero if it is not.

RETURN VALUES

The *system()* function returns the exit status of the shell, or -1 if the *wait()* for the shell failed. A return value of 127 means the execution of the shell failed.

SEE ALSO

sh

exec(), *wait()*, *popen()*

NAME

tanh – hyperbolic tangent function

SYNOPSIS

```
#include <math.h>
```

```
double tanh (double x)
```

DESCRIPTION

The *tanh()* function computes the hyperbolic tangent of *x*. For a discussion of error due to roundoff, see the *math* reference page.

RETURN VALUES

The *tanh()* function returns the hyperbolic tangent value.

SEE ALSO

acos(), *asin()*, *atan()*, *atan2()*, *cos()*, *cosh()*, *sin()*, *sinh()*, *tan()*, *math()*

NAME

tan() – tangent function

SYNOPSIS

```
#include <math.h>

double tan (double x)
```

DESCRIPTION

The *tan()* function computes the tangent of *x* (measured in radians). A large magnitude argument may yield a result with little or no significance.

RETURN VALUES

The *tan()* function returns the tangent value.

SEE ALSO

acos(), *asin()*, *atan()*, *atan2()*, *cos()*, *cosh()*, *sin()*, *sinh()*, *tanh()*

NAME

tcgetpgrp() – get foreground process group ID

SYNOPSIS

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

pid_t tcgetpgrp (int fd)
```

DESCRIPTION

This call returns the ID number of the terminal's foreground process group.

A member of a background process group can call this routine, but the value of the process group ID may be later changed by a foreground process.

If there is no foreground process group, *tcgetpgrp()* returns an invalid number.

RETURN VALUES

On success, *tcgetpgrp()* returns a process group ID as a **pid_t**. On failure, it returns `-1` and sets *errno*.

ERRORS

tcgetpgrp() can fail for the following reasons:

- | | |
|----------|--|
| [EBADF] | The <i>fd</i> argument is not a valid file descriptor. |
| [ENOTTY] | The calling process does not have a controlling terminal or the underlying terminal device represented by <i>fd</i> is not the controlling terminal. |

SEE ALSO

setpgid(), *setsid()*, *tcsetpgrp()*

NAME

tcsendbreak(), tcdrain(), tcflow(), tcflush() – line control functions

SYNOPSIS

```
#include <termios.h>

int tcdrain (int fd)
int tcflow (int fd, int action)
int tcflush (int fd, int queue_selector)
int tcsendbreak (int fd, int len)
```

DESCRIPTION

These functions control the terminal line, but do not directly affect the terminal attributes (for those, see *tcsetattr()*).

The *tcdrain()* function waits until all output written to the terminal referenced by *fd* has been transmitted to the terminal.

The *tcflow()* function suspends transmission of data to or the reception of data from the terminal referenced by *fd* depending on the value of *action*. The value of *action* must be one of the following:

TCOOFF

Suspend output.

TCOON

Restart suspended output.

TCIOFF Transmit a STOP character, which is intended to cause the terminal to stop transmitting data to the system.

TCION Transmit a START character, which is intended to cause the terminal to start transmitting data to the system.

The *tcflush()* function discards any data written to the terminal referenced by *fd* which has not been transmitted to the terminal, or any data received from the terminal but not yet read, depending on the value of *queue_selector*. The value of *queue_selector* must be one of the following:

TCIFLUSH

Flush data received but not read.

TCOFLUSH

Flush data written but not transmitted.

TCIOFLUSH

Flush both data received but not read and data written but not transmitted.

The *tcsendbreak()* function transmits a continuous stream of zero-valued bits for four-tenths of a second to the terminal referenced by *fd*.

RETURN VALUES

Upon successful completion, all of these functions return a value of zero. If an error occurs, all of these functions return a value of -1 and set *errno*.

ERRORS

These functions can fail for the following reasons:

[EBADF]	The <i>fd</i> argument is not a valid file descriptor.
[EINVAL]	The <i>action</i> or <i>queue_selector</i> argument is not a proper value.
[ENOTTY]	The file associated with <i>fd</i> is not a terminal.
[EINTR]	A signal interrupted the <i>tcdrain()</i> function.

SEE ALSO*tcsetattr()**Microsoft Windows NT POSIX Subsystem POSIX Conformance Document*

NAME

`cfgetispeed()`, `cfsetispeed()`, `cfgetospeed()`, `cfsetospeed()`, `tcgetattr()`, `tcsetattr()` – manipulate the **termios** structure

SYNOPSIS

```
#include <termios.h>

speed_t cfgetispeed (struct termios *t)
int cfsetispeed (struct termios *t, speed_t speed)
speed_t cfgetospeed (struct termios *t)
int cfsetospeed (struct termios *t, speed_t speed)
int tcgetattr (int fd, struct termios *t)
int tcsetattr (int fd, int action, struct termios *t)
```

DESCRIPTION

The `tcgetattr()` and `tcsetattr()` functions are provided for getting and setting the **termios** structure.

The `cfgetispeed()`, `cfsetispeed()`, `cfgetospeed()` and `cfsetospeed()` functions are provided for getting and setting the baud rate values in the **termios** structure. These functions set attributes for the terminal, but those attributes do not take effect, nor are all errors detected, until the `tcsetattr()` function is called. Certain values for baud rates set in the **termios** structure and passed to `tcsetattr()` have special meanings. These are discussed in the portion of the manual page that describes the `tcsetattr()` function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the **termios** structure. The unsigned integer **speed_t** is **typedef**'d in the include file `<termios.h>`. Use the symbolic values defined there, since the value of the integer does *not* necessarily correspond to the baud rate being represented. (For example, B9600 may not be 9600.)

Name	Description	Name	Description
B0	Hang up	B600	600 baud
B50	50 baud	B1200	1200 baud
B75	75 baud	B1800	1800 baud
B110	110 baud	B2400	2400 baud
B134	134.5 baud	B4800	4800 baud
B150	150 baud	B9600	9600 baud
B200	200 baud	B19200	19,200 baud
B300	300 baud	B38400	38,400 baud

The `cfgetispeed()` function returns the input baud rate in the **termios** structure referenced by *tp*.

The `cfsetispeed()` function sets the input baud rate in the **termios** structure referenced by *tp* to *speed*.

The `cfgetospeed()` function returns the output baud rate in the **termios** structure referenced by *tp*.

The `cfsetospeed()` function sets the output baud rate in the **termios** structure referenced by *tp* to *speed*.

Upon successful completion, the functions *cfsetispeed()* and *cfsetospeed()* return a value of 0. Otherwise, a value of -1 is returned and the global variable *errno* is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation and the SIGTTOU signal is not sent.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The *tcgetattr()* function copies the parameters associated with the terminal referenced by *fd* in the **termios** structure referenced by *tp*. This function is allowed from a background process, however, the terminal attributes may be subsequently changed by a foreground process.

The *tcsetattr()* function sets the parameters associated with the terminal from the **termios** structure referenced by *tp*. The *action* field is created by ORing the following values, as specified in the include file **<termios.h>**.

TCSANOW

The change occurs immediately.

TCSADRAIN

The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.

TCSAFLUSH

The change occurs after all output written to *fd* has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.

The 0 baud rate is used to terminate the connection. If 0 is specified as the output speed to the function *tcsetattr()*, modem control will no longer be asserted on the terminal, disconnecting the terminal.

If zero is specified as the input speed to the function *tcsetattr()*, the input baud rate will be set to the same value as that specified by the output baud rate.

RETURN VALUES

If *tcsetattr()* is unable to make any of the requested changes, it returns -1 and sets *errno*. Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions *tcgetattr()* and *tcsetattr()* return a value of 0. Otherwise, they return -1 and the global variable *errno* is set to indicate the error.

ERRORS

The functions *tcgetattr()* and *tcsetattr()* can set *errno* to the following values:

[EBADF]	The <i>fd</i> argument to <i>tcgetattr()</i> or <i>tcsetattr()</i> was not a valid file descriptor.
[EINTR]	The <i>tcsetattr()</i> function was interrupted by a signal.
[EINVAL]	The <i>action</i> argument to the <i>tcsetattr()</i> function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
[ENOTTY]	The file associated with the <i>fd</i> argument to <i>tcgetattr()</i> or <i>tcsetattr()</i> is not a terminal.

NOTES

As of this writing, setting (or getting) the baud rate for a console window is unimportant, since on single PCs there is no modulated phone line connecting the terminal and the kernel.

SEE ALSO

tcsendbreak()

termios

NAME

tcsetpgrp() – set foreground process group ID

SYNOPSIS

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

int tcsetpgrp (int fd, pid_t pgrp_id)
```

DESCRIPTION

If the process has a controlling terminal, the *tcsetpgrp* function sets the foreground process group ID associated with the terminal device to *pgrp_id*. The terminal device associated with *fd* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must be the same as the process group ID of a process in the same session as the calling process.

Upon successful completion, *tcsetpgrp* returns a value of zero.

ERRORS

If an error occurs, *tcsetpgrp* returns –1 and the global variable *errno* is set to indicate the error, as follows:

[EBADF]	The <i>fd</i> argument is not a valid file descriptor.
[EINVAL]	An invalid value of <i>pgrp_id</i> was specified.
[ENOTTY]	The calling process does not have a controlling terminal, or the file represented by <i>fd</i> is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.
[EPERM]	The <i>pgrp_id</i> argument does not match the process group ID of a process in the same session as the calling process.

SEE ALSO

setpgid(), *setsid()*, *tcgetpgrp()*

NAME

tempnam() – temporary file routine

SYNOPSIS

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

```
char * tempnam (const char *dir, char *px)
```

DESCRIPTION

The *tempnam()* function is similar to *tmpnam()* — it returns a pointer to a pathname of a file that does not exist at the time of the call. Unlike *tmpnam()*, you can specify the directory containing the file (*dir*), and you can specify the filename's prefix, (*px*).

The *tempnam()* function also looks in different directories by default than does *tmpnam()*. If the *dir* argument is NULL or does not reference an appropriate directory, *tempnam()* uses the path prefix defined in in **<stdio.h>**. If that fails, it searches for **\$OPENNT_ROOT/tmp**, then the contents of *TMPDIR*.

The argument *px*, if non-NULL, is used to specify a file name prefix, which will be the first part of the created file name. The *px* can be up to five characters long.

Tempnam() allocates space for the string; the pointer can be used in a call to *free()*.

RETURN VALUES

The *tempnam()* function returns a pointer to a file name on success, and a NULL pointer on error.

ERRORS

The *tempnam()* function may fail and set the global variable *errno* for this reason:

[ENOMEM] There isn't enough memory.

NOTES

Using *tempnam()* is discouraged.

SEE ALSO

mktemp(), *tmpfile()*

NAME

tgetent(), tgetnum(), tgetflag(), tgetstr(), tgoto(), tputs() – terminal independent operation (termcap) routines

SYNOPSIS

```
#include <curses.h>

char PC;
char *BC;
char *UP;
short ospeed;

int tgetent (char *bp, char *name)
int tgetnum (char *id)
int tgetflag (char *id)
char * tgetstr (char *id, char **area)
char * tgoto (char *cm, char *destcol, char *destline)
char * tputs (register char *cp, int affcnt, void (*outc)())
```

DESCRIPTION

These functions extract and use capabilities from a terminal capability data base, usually **\$OPENNT_ROOT/usr/share/termcap**. (The actual parent of **usr/share** depends upon where you installed INTERIX.) These are low level routines that access the termcap database; high-level routines are available in the *curses* package, but they access the terminfo database.

The *tgetent()* function extracts the entry for terminal *name* into the buffer at *bp*. The *bp* argument should be a character buffer of size 1024 and must be retained through all subsequent calls to *tgetnum()*, *tgetflag()*, and *tgetstr()*. The *tgetent()* function returns -1 if none of the **termcap** database files could be opened, 0 if the terminal name given does not have an entry, and 1 if all goes well.

Tgetent() looks in the environment for a **TERMCAP** variable. If found, and the value does not begin with a slash, and the terminal type *name* is the same as the environment string **TERM**, the **TERMCAP** string is used instead of reading a **termcap** file. If it does begin with a slash, the string is used as a pathname of the **termcap** file to search. If **TERMCAP** does not begin with a slash and *name* is different from **TERM**, *tgetent()* searches the files **\$HOME/.termcap** and **\$OPENNT_ROOT/usr/share/termcap**, in that order — unless the environment variable **TERMPATH** exists, in which case it specifies a list of file pathnames (separated by spaces or colons) to be searched instead. Whenever multiple files are searched and a **tc** field occurs in the requested entry, the entry it names must be found in the same file or one of the succeeding files. This can speed up entry into programs that call *tgetent()*, as well as help debug new terminal descriptions or make one for your terminal if you can't write the file **\$OPENNT_ROOT/usr/share/termcap**.

The *tgetnum()* function gets the numeric value of capability *id*, returning -1 if it is not given for the terminal. The *tgetflag()* function returns 1 if the specified capability is present in the terminal's entry, 0 if it is not.

The *tgetstr()* function returns the string value of the capability *id*, places it in the buffer at *area*, and advances the *area* pointer. It decodes the abbreviations for this field, except for cursor addressing and padding information. The *tgetstr()* function returns NULL if the capability was not found.

The *tgoto()* function returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables *UP* (from the *up* capability) and *BC* (if *bc* is given rather than *bs*) if necessary to avoid placing *\n*, *^D* or *^@* in the returned string. (Programs which call *tgoto()* should be sure to turn off the XTABS bit(s), since *tgoto()* may now output a tab. Note that programs using termcap should in general turn off XTABS anyway since some terminals use control-I for other functions, such as nondestructive space.) If a % sequence is given which is not understood, then *tgoto()* returns (OOPS).

The *tputs()* function decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by *stty()*. The external variable *PC* should contain a pad character to be used (from the *pc* capability) if a null (*^@*) is inappropriate.

FILES

/usr/lib/libtermcap.a

The **-ltermcap** library (also known as **-lterm**lib).

\$OPENNT_ROOT/usr/share/termcap

The standard terminal capability database (the actual location on your system will depend upon where INTERIX is installed).

\$HOME/.termcap

The user's terminal capability database.

NOTES

Although the functions are declared in **<curses.h>**, there are two different sets of them: one in the termcap library and one set in the *curses* library that refer to the terminfo database.

To include the termcap versions, link against the termcap library (**libtermcap.a**); to include the terminfo versions, link against the curses library

SEE ALSO

ex

curses

termcap *terminfo*

NAME

time() – get time of day

SYNOPSIS

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

time_t time (time_t *tloc)
```

DESCRIPTION

The *time()* function returns the value of time in seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time.

A copy of the time value may be saved to the area indicated by the pointer *tloc*. If *tloc* is a NULL pointer, no value is stored.

RETURN VALUES

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

ERRORS

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

[EFAULT] An argument address referenced invalid memory.

Time() will also return -1 if the date is not in the range of 1970 to 2105.

SEE ALSO

ctime()

NAME

times() – get process times

SYNOPSIS

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

clock_t times (struct tms *tp)
```

DESCRIPTION

The *times()* function returns the value of time in CLK_TCK's of a second since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time.

It also fills in the structure pointed to by *tp* with time-accounting information.

The *tms* structure is defined as follows:

```
typedef struct {
    clock_t tms_utime;
    clock_t tms_stime;
    clock_t tms_cutime;
    clock_t tms_cstime;
}
```

The elements of this structure are defined as follows:

tms_utime

The CPU time charged for the execution of user instructions.

tms_stime

The CPU time charged for execution by the system on behalf of the process.

tms_cutime

The sum of the *tms_utimes* and *tms_cutimes* of the child processes.

tms_cstime

The sum of the *tms_stimes* and *tms_cstimes* of the child processes.

All times are in CLK_TCK's of a second.

The times of a terminated child process are included in the *tms_cutime* and *tms_cstime* elements of the parent when one of the *wait()* functions returns the process ID of the terminated child to the parent. If an error occurs, *times()* returns the value -1 and sets *errno* to indicate the error.

The return value from *times()* can overflow the range of a *clock_t* object.

ERRORS

The *times()* function may fail for these errors:

[EFAULT] The *tp* argument is an invalid address.

SEE ALSO

time

wait()

NAME

tmpfile(), tmpnam() – temporary file routines

SYNOPSIS

```
#include <stdio.h>

FILE * tmpfile (void)
char * tmpnam (char *str)
```

DESCRIPTION

The *tmpfile()* function returns a pointer to a stream associated with a file descriptor returned by the routine *mkstemp()*. The created file is unlinked before *tmpfile()* returns, causing the file to be automatically deleted when the last reference to it is closed. The file is opened with the access value *w+*.

The *tmpnam()* function returns a pointer to a file name which did not reference an existing file at some indeterminate point in the past. If the argument *s* is non-NULL, the file name is copied to the buffer it references. Otherwise, the file name is copied to a static buffer. In either case, *tmpnam()* returns a pointer to the file name.

The buffer referenced by *s* is expected to be at least {L_tmpnam} bytes in length. {L_tmpnam} is defined in the include file *<stdio.h>*.

The environment variable *TMPDIR* (if set), the argument *tmpdir* (if non-NULL), and the directory */tmp* are tried, in that order, as directories in which to store the temporary file.

RETURN VALUES

The *tmpfile()* function returns a pointer to an open file stream on success, and a NULL pointer on error.

The *tmpnam()* returns a pointer to a file name on success, and a NULL pointer on error.

ERRORS

The *tmpfile()* function may fail and set the global variable *errno* for any of the errors specified for the library functions *fdopen()* or *mkstemp()*.

The *tmpnam()* function may fail and set *errno* for any of the errors specified for the library function *mktemp()*.

The *tempnam()* function may fail and set *errno* for any of the errors specified for the library functions *malloc()* or *mktemp()*.

SEE ALSO

mktemp(), *tempnam()*

NAME

toascii – convert a byte to 7-bit ASCII

SYNOPSIS

```
#include <ctype.h>
int toascii (int c)
```

DESCRIPTION

The *toascii()* function strips all but the low 7 bits from a letter, including parity or other marker bits.

RETURN VALUES

The *toascii()* function always returns a valid ASCII character.

SEE ALSO

isascii(), *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *tolower()*, *toupper()*

NAME

tolower(), _tolower() – upper case to lower case letter conversion

SYNOPSIS

```
#include <ctype.h>

int tolower (int c)
int _tolower (int c)
```

DESCRIPTION

The *tolower()* function converts an upper-case letter to the corresponding lower-case letter. If *c* is not an upper-case letter, it is not converted.

The *_tolower()* function does the same, but the argument *c* must be an upper-case letter. This routine is faster but will break with the wrong input.

RETURN VALUES

If the argument is an upper-case letter, the *tolower()* function returns the corresponding lower-case letter if there is one; otherwise the argument is returned unchanged.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *toupper()*

NAME

`toupper()`, `_toupper()` – lower case to upper case letter conversion

SYNOPSIS

```
#include <ctype.h>

int toupper (int c)
int _toupper (int c)
```

DESCRIPTION

The `toupper()` function converts a lower-case letter to the corresponding upper-case letter. If `c` is not a lower-case letter, it is not converted.

The `_toupper()` function does the same, but the argument `c` must be a lower-case letter. This routine is faster but will break with the wrong input.

RETURN VALUES

If the argument is a lower-case letter, the `toupper()` function returns the corresponding upper-case letter if there is one; otherwise the argument is returned unchanged.

SEE ALSO

`isascii()`, `isalnum()`, `isalpha()`, `isblank()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `tolower()`

NAME

tdelete(), tfind(), tsearch(), twalk() – manage a binary search tree

SYNOPSIS

```
#include <search.h>

void *tsearch (const void *key, void **rootp,
               int (*compar)(const void *, const void *))
void *tfind (const void *key, void **rootp,
             int (*compar)(const void *, const void *))
void *tdelete (const void *key, void **rootp,
               int (*compar)(const void *, const void *))
void twalk (const void *root, void (*action)(const void *, VISIT, int));
```

DESCRIPTION

The *tsearch()*, *tfind()*, *tdelete()*, and *twalk()* function manage binary search trees. *tsearch()* is used to build and access the tree; *tfind()* searches the tree but will not insert nodes. The *tdelete()* function deletes a node from a tree, and the *twalk()* function traverses a tree.

The arguments for *tsearch()*, *tfind()*, and *tdelete()* are identical. The *key* is a pointer to the element to be located (for *tsearch()* and *tfind()*), added (for *tsearch()*), or deleted (for *tdelete()*). The *rootp* is a pointer to a variable that points to the root node of the tree. If *rootp* is NULL, it indicates an empty tree. The *compar* argument is the address of a user-supplied comparison routine which is called with the pointers to the elements being compared. The comparison function doesn't need to compare every byte; the elements of the tree can contain arbitrary data as well as the values actually being compared. The routine returns an **int**:

```
<0  if the first argument is less than the second argument
0   if the first argument is equal to the second argument
>0  if the first argument is greater than the second argument
```

Both *tsearch()* and *tfind()* search the tree. If the element pointed to by *key* is found, they return a pointer to that element. It is the responsibility of the calling function to copy the element. If the element can't be found, *tsearch()* inserts it in the tree and returns a pointer to this new node, while *tfind()* simply returns a null pointer.

The *tdelete()* function deletes the node indicated by *key* and returns a pointer to the successor of the deleted node (one of the children), or if there are no children, returns a pointer to the root of the tree. If the root node is deleted, the function changes the variable pointed to by *rootp*. *Tdelete()* returns NULL if the node can't be found or when the last node has been deleted.

The *twalk()* function traverses a binary search tree, starting with the node pointed to by *root*. (The *root* of the walk can be any node in the tree; the walk visits the tree below that node.) The second argument to *twalk()*, *action*, is a pointer to a function which is invoked at each node. The function pointed to by *action* takes three arguments:

- The address of the node to be visited;
- a value from the enumerated type **VISIT** (defined in **<search.h>**). which indicates whether this is the first, second, or third time the The types assume a depth-first, left-to-right walk of the tree, and are:

preorder:

Visiting a node before any of its children.

postorder:

Visiting a node after its left child and before its right.

endorder:

Visiting a node after its children.

leaf:

The node is a leaf node.

- the level of the node in the tree, where the root is level 0.

RETURN VALUE

The *tsearch()* function returns a pointer to the node if it was found; if the item was inserted, the function returns a pointer to the inserted item. If there is not enough space to create a new node, or if *rootp* is a null pointer on entry, *tsearch()* returns a null pointer.

The *tfind()* function returns a pointer to the node if it was found; if it was not found, or if *rootp* is a null pointer on entry, the function returns a null pointer.

The *tdelete()* function returns a pointer to the parent of the deleted node; if the node was not found, or if *rootp* is a null pointer on entry, it returns a null pointer.

The *twalk()* function does not return a value.

NOTES

Don't be confused by the meaning of "postorder". Here it refers to visiting a node after its left child and before its right. Some people refer to the order of visiting tree nodes as "preorder", "inorder", and "postorder".

This implementation uses *malloc()* across trees. Using *tdelete()* to remove all of the nodes in a given tree may not free all of the memory associated with that tree because some of the memory may also be associated with another tree. When all of the nodes used by all of the trees associated with a given program are *tdelete()*d, all of the memory is freed.

SEE ALSO

bsearch(), *hsearch()*, *lsearch()*

NAME

ttyname(), isatty() – get name of associated terminal (tty) from file descriptor

SYNOPSIS

```
#include <unistd.h>

char * ttyname (int fd)
int isatty (int fd)
```

DESCRIPTION

These functions operate on the system file descriptors for terminal type devices. These descriptors are not related to the standard I/O **FILE typedef**.

The *isatty()* function determines if the file descriptor *fd* refers to a valid terminal type device.

The *ttyname()* function gets the related device name of a file descriptor for which *isatty()* is true. The name has the form *ttynxx*, where *x* is a digit.

RETURN VALUES

The *ttyname()* function returns the null-terminated name if the tty device is found and *isatty()* is true; otherwise it returns NULL.

Isatty() returns zero if *fd* is not a valid terminal device and non-zero if it is.

SEE ALSO

ctermid()

NAME

tzset() – set time zone and daylight savings time variables

SYNOPSIS

```
#include <time.h>
extern char *tzname[];
extern long int timezone;
extern int daylight;

void tzset (void)
```

DESCRIPTION

The `tzset()` function uses the value of the `TZ` environment variable to set the time conversion information used by the library routines `localtime()`, `ctime()`, `strftime()`, and `mktime()`. It also sets the external variables `tzname`, `timezone`, and `daylight`:

```
extern char *tzname[2] = { "std", "dst"};
extern long int timezone;
extern int daylight;
```

where *std* and *dst* have the same meaning as in the *TZ* specification (see below).

The *timezone* variable contains the difference (in seconds) between UTC and local time. The *daylight* variable is non-zero when daylight savings time is in effect.

The value of *TZ* has the following syntax (spaces inserted for clarity):

```
std offset [ dst [ offset ] [ rule ] ]
```

Where:

std and *dst*

Three or more bytes that designate the standard (*std*) or summer (*dst*) time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper and lowercase letters are explicitly allowed. Any characters except a leading colon (:), digits, comma ,, minus -, plus +), and ASCII NUL are allowed.

offset Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

```
hh [ : mm [: ss ] ]
```

The minutes *mm* and seconds *ss* are optional. The hour *hh* is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between zero and 24, and the minutes (and seconds) — if present — between zero and 59. If preceded by a (–) the time zone shall be east of the Prime Meridian; otherwise it shall be west (which may be indicated by an optional preceding (+).)

rule Indicates when to change to and back from summer time. The *rule* has the form:

```
date / time , date / time
```

where the first *date* describes when the change from standard to summer time occurs and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* is one of the following:

- Jn* The Julian day *n* ($1 < n < 365$). Leap days are not counted; that is, in all years — including leap years — February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.
- n* The zero-based Julian day ($0 < n < 365$). Leap days are counted, and it is possible to refer to February 29.
- Mm.n.d* The *d*'th day ($0 < d < 6$) of week *n* of month *m* of the year ($1 < n < 5$), ($1 < m < 12$), where week 5 means “the last *d* day in month *m*” which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day zero is Sunday.

The *time* has the same format as *offset* except that no leading sign (–) or (+) is allowed. The default, if *time* is not given, is 02:00:00.

For compatibility with System V Release 3.1, a semicolon (;) may be used to separate the *rule* from the rest of the specification.

If the *TZ* variable is absent from the environment, it defaults to PST8PDT.

SEE ALSO

date

gettimeofday(), *ctime()*, *getenv()*, *time()*

NAME

uidtontsid() – retrieve user's NT security ID

SYNOPSIS

```
#include <opennt/security.h>

char *uidtontsid(uid_t uid)
```

DESCRIPTION

The *uidtontsid()* call retrieves the Windows NT security ID of the user specified by *uid*, and returns a pointer to a static buffer that contains the security ID as an ASCII string.

Windows NT security IDs are unique throughout time and space; a POSIX user ID under Windows NT is only unique within a trusted domain network.

DIAGNOSTICS

On success, *uidtontsid()* returns a pointer to a static buffer. On failure, it returns NULL and sets *errno* to indicate the cause of the error.

ERRORS

The following errors can be set:

[EINVAL]

The *uid* is invalid.

SEE ALSO

NAME

`umask()` – set file creation mode mask

SYNOPSIS

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

mode_t umask (mode_t numask)
```

DESCRIPTION

The `umask()` routine sets the process's file mode creation mask to *numask* and returns the previous value of the mask. *numask* is a set of mode bits (as described for `stat()` or `chmod()`). These access permission bits are used by system calls, including `open()`, `mkdir()`, and `mkfifo()`, to turn off corresponding bits requested in file mode. This clearing allows each user to restrict the default access to his files.

The default mask value is `S_IWGRP | S_IWOTH` (022, write access for the owner only). Child processes inherit the mask of the calling process.

RETURN VALUES

The previous value of the file mode mask is returned by the call.

ERRORS

The `umask()` function is always successful.

SEE ALSO

`chmod()`, `open()`, `stat()`

NAME

uname() – get system identification

SYNOPSIS

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

int uname (struct utsname *name)
```

DESCRIPTION

The *uname()* function stores null-terminated strings of information identifying the current system into the structure referenced by *name*.

The *utsname* structure is defined in the **<sys/utsname.h>** header file, and contains the following members:

sysname Name of the operating system implementation.
nodename Network name of this machine.
release Release level of the operating system.
version Version level of the operating system.
machine Machine hardware platform.

RETURN VALUES

If *uname()* is successful, it returns 0; otherwise, it returns –1 and set *errno* to indicate the error.

ERRORS

Uname() detects the following errors:

[EFAULT] The *name* argument is an invalid address.

NAME

ungetc() – un-get character from input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc (int c, FILE *stream)
```

DESCRIPTION

The *ungetc()* function pushes the character *c* (converted to an unsigned char) back onto the input stream pointed to by *stream*. The pushed-back characters will be returned by subsequent reads on the stream (in reverse order). A successful intervening call, using the same stream, to one of the file positioning functions (*fseek()*, *fsetpos()*, or *rewind()*) will discard the pushed back characters.

One character of push-back is guaranteed, but as long as there is sufficient memory, an effectively infinite amount of pushback is allowed.

If a character is successfully pushed-back, the end-of-file indicator for the stream is cleared.

RETURN VALUES

The *ungetc()* function returns the character pushed-back after the conversion, or EOF if the operation fails. If the value of the argument *c* character equals EOF, the operation will fail and the stream will remain unchanged.

SEE ALSO

getc(), *fseek()*, *setvbuf()*

NAME

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.

Calling *unlink()* with a directory is not supported.

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:

- | | |
|----------------|---|
| [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. |
| [ENAMETOOLONG] | A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters. |
| [ENOENT] | The named file does not exist. |
| [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. |
| [EIO] | An I/O error occurred while deleting the directory entry or deallocating the file serial number. |
| [EROFS] | The named file resides on a read-only file system. |

SEE ALSO

close(), *link()*, *remove()*, *rmdir()*

NAME

utime() – set file times

SYNOPSIS

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

int utime (const char *file, const struct utimbuf *timep)
```

DESCRIPTION

The *utime()* function sets the access and modification times of the named file from the members of the *timep* argument.

If the times are specified (the *timep* argument is non-NULL) the caller must be the owner of the file.

If the times are not specified (the *timep* argument is NULL) the caller must be the owner of the file, have permission to write the file.

Timep is defined in **<utime.h>** and contains the following members:

time_t	<i>actime</i>	Access time
time_t	<i>modtime</i>	Modification time

ERRORS

The *utime()* function may fail and set *errno* for any of these errors:

- | | |
|----------------|---|
| [EACCES] | The effective user ID of the process doesn't own the file and write access is denied, or search permission is denied on some directory in the <i>file</i> pathname. |
| [ENAMETOOLONG] | The <i>file</i> was longer than {PATHMAX} characters, or a component of <i>file</i> exceeded {NAMEMAX} characters. |
| [ENOENT] | The <i>file</i> doesn't exist or is an empty string. |
| [ENOTDIR] | Some component of the pathname prefix of <i>file</i> is not a directory. |
| [EPERM] | The <i>times</i> argument is non-NULL, and the effective user ID of the process is not the owner of the file, nor does it have the appropriate privileges. |
| [EROFS] | |

SEE ALSO

stat()

NAME

wait(), waitpid() – wait for process termination

SYNOPSIS

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

pid_t wait (int *status)
pid_t waitpid (pid_t wpid, int *status, int options)
```

DESCRIPTION

The *wait()* function suspends execution of its calling process until *status* information is available for any terminated child process, or a signal is received. On return from a successful *wait()* call, the *status* area contains termination information about the process that exited as defined below. *waitpid()* is a more general interface that allows you more control over the behavior.

The *status* parameter is a pointer to an integer description of the status of the child process, which can be interpreted using macros:

The following macros may be used to test the manner of exit of the process. One of the first three macros will evaluate to a non-zero (true) value:

*WIFEXITED(*status)*

True if the process terminated normally by a call to *_exit()* or *exit()*.

*WIFSIGNALED(*status)*

True if the process terminated due to receipt of a signal.

*WIFSTOPPED(*status)*

True if the process has not terminated, but has stopped and can be restarted.
This macro can be true only if the wait call specified the *WUNTRACED* option.

Depending on the values of those macros, the following macros produce the remaining status information about the child process:

*WEXITSTATUS(*status)*

If *WIFEXITED()* is true, evaluates to the low-order 8 bits of the argument passed to *_exit()* or *exit()* by the child.

*WTERMSIG(*status)*

If *WIFSIGNALED()* is true, evaluates to the number of the signal that caused the termination of the process.

*WCOREDUMP(*status)*

If *WIFSIGNALED()* is true, evaluates as true if the termination of the process was accompanied by the creation of a core file containing an image of the process when the signal was received.

*WSTOPSIG(*status)*

If *WIFSTOPPED()* is true, evaluates to the number of the signal that caused the process to stop.

The *waitpid()* call also takes the *pid* argument, specifying which child processes will be waited for, and the *options* argument, a bitwise OR of flags controlling *waitpid()*'s behavior.

Values for *pid* can be:

- pid*=-1: Status is requested for all child processes (this is identical to *wait()*).
- pid*>0: Status is requested for the child process with process ID *pid*.
- pid*=0: Status is requested for any child process whose process group ID is equal to that of the calling process.
- pid*<-1: Status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

The *options* parameter contains the bitwise OR of any of the following options, defined in **<sys/wait.h>**:

WNOHANG

the call should not block if there are no processes that wish to report status.

WUNTRACED

children of the current process that are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal also have their status reported.

RETURN VALUES

If *wait()* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

If *waitpid()* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If there are no children not previously awaited, -1 is returned with *errno* set to [ECHILD]. Otherwise, if WNOHANG is specified and there are no stopped or exited children, 0 is returned. If an error is detected or a caught signal aborts the call, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

Wait() will fail and return immediately if:

- | | |
|----------|---|
| [ECHILD] | The calling process has no existing unwaited-for child processes. |
| [EINTR] | The call was interrupted by a caught signal. |
| [EINVAL] | The <i>options</i> argument was invalid. |

NAME

write(), writev() – write output

SYNOPSIS

```
#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 {
    void *iov_base;
    size_t iov_len;
};
```

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.

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.

When calling the *write()* function on a file that is not a regular file with *nbytes* set to zero, the function returns 0.

When the *nbytes* argument exceeds {SSIZE_MAX}, the return value is truncated to type **ssize_t**.

ERRORS

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

- | | |
|----------|--|
| [EAGAIN] | The file was marked for non-blocking I/O, and no data could be written immediately. |
| [EBADF] | <i>D</i> is not a valid descriptor open for writing. |
| [EFAULT] | Part of <i>iov</i> or data to be written to the file points outside the process's allocated address space. |

[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
[EINVAL]	The pointer associated with <i>d</i> was negative.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EPIPE]	An attempt is made to write to a socket of type SOCK_STREAM that is not connected to a peer socket.
[ENOSPC]	There is no free space remaining on the file system containing the file.
[EPIPE]	An attempt is made to write to a pipe that is not open for reading by any process.

In addition, *writenv()* may return one of the following errors:

[EINVAL]	<i>Iovcnt</i> was less than or equal to 0, or greater than {UIO_MAXIOV}.
[EINVAL]	One of the <i>iov_len</i> values in the iov array was negative.
[EINVAL]	The sum of the <i>iov_len</i> values in the iov array overflowed a 32-bit integer.

SEE ALSO

fcntl(), *lseek()*, *open()*, *pipe()*, *select()*