LynxOS Reference Manual

LynxOS Version 2.5

DOC-0173

Library Functions (N–Z)

U.S. Patents 5,469,571; 5,594,903

Printed in the United States of America.

Contents

Library Functions (N-Z)

Contents

Library Functions (N-Z)

NAME

net_error – get pointer to message for a network error code

SYNOPSIS

```
char *neterror(num)
int num;
```

DESCRIPTION

net_error returns a pointer to the message string corresponding to error number **num** in the system error message tables, sys_errlist. This is the message, without the newline, that would be printed by perror if it were called with **errno** having the value of **num**. net_error allows access to system error messages without indexing directly into the message tables. This is needed since the networking error messages are not necessarily contiguous with the rest of the system messages.

SEE ALSO

Library Functions - perror()

NAME

`nlist` – get entries from name list

SYNOPSIS

```
#include <a.out.h>

int nlist(filename, nl)
char *filename;
struct nlist *nl;
```

DESCRIPTION

`nlist` examines the name list in the executable file **`filename`**, and extracts a list of values and puts them in the array of `nlist` structures pointed to by **`nl`**. Each `nlist` structure in the name list pointed to by **`nl`** contains the name of a variable, its type and its value. The list is terminated by an `nlist` structure with a null name string. Each variable name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the `n_type` and `n_value` fields of the `nlist` structure. If the name is not found, both these fields are set to `0`.

DIAGNOSTICS

`nlist` returns `0` if successful, or `-1` if not.

SEE ALSO

File Format - `a.out, coff`

NAME

pathconf, fpathconf – get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>

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

long fpathconf(fd, name)
int fd;
int name;
```

DESCRIPTION

The pathconf() and fpathconf() functions provide a method for the application to determine the current value of a configurable limit or option (variable) that is associated with a file or directory.

For pathconf(), the **path** argument points to the pathname of a file or directory. For fpathconf(), the **fd** argument is an open file descriptor.

The **name** argument represents the variable to be queried relative to that file or directory. All of the variables listed in the following table are supported. The variables in this table come from <limits.h> or **<unistd.h>** and the symbolic constants, defined in **<unistd.h>**, that are the corresponding values used for **name**.

DIAGNOSTICS

If **name** is an invalid value, the pathconf() and fpathconf() functions returns -1.

If the variable corresponding to **name** has no limit for the path or file descriptor, the `pathconf()` and `fpathconf()` functions returns `-1` without changing **errno**.

Variable	**Name** Value	Notes
`LINK_MAX`	`_PC_LINK_MAX`	If *path* or *fd* refers to a directory, the value returned applies to the directory itself.
`MAX_CANON`	`PC_MAX_CANON`	If *path* or *fd* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
`MAX_INPUT`	`PC_MAX_INPUT`	If *path* or *fd* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
`NAME_MAX`	`PC_NAME_MAX`	If *path* or *fd* refers to a directory, the value returned applies to the filenames within the directory. If *path* or *fd* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
`PATH_MAX`	`_PC_PATH_MAX`	If *path* or *fd* refers to a directory, the value returned is the maximum length of a relative pathname when the specified directory is the working directory. If *path* or *fd* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.

Variable	**Name** Value	Notes
PIPE_BUF	_PC_PIPE_BUF	If *path* refers to a FIFO, or *fd* refers to a pipe or a FIFO, the value returned applies to the referenced object itself. If *path* or *fd* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *path* or *fd* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	If *path* or *fd* refers to a directory, the value returned applies to any files defined in this part of ISO/IEC 9945, other than directories, that exist or can be created within the directory.
_POSIX_NO_TRUNC	_PC_NO_TRUNC	If *path* or *fd* refers to a directory, the value returned applies to the filenames within the directory. If *path* or *fd* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
_POSIX_VDISABLE	_PC_VDISABLE	If *path* or *fd* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.

If the implementation needs to use **path** to determine the value of **name** and the implementation does not support the association of **name** with the file specified by *path*, or if the process did not have the appropriate privileges to query the file specified by **path**, or **path** does not exist, the `pathconf()` function returns `-1`.

If the implementation needs to use **fd** to determine the value of **name** and the implementation does not support the association of **name** with the file specified by **fd**, or if **fd** is an invalid file descriptor, the `fpathconf()` function returns `-1`.

Otherwise, the `pathconf()` and `fpathconf()` functions return the current variable value for the file or directory without changing **errno**. The value returned is not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's `<limits.h>` or **<unistd.h>**.

If any of the following conditions occur, the `pathconf()` and `fpathconf()` functions returns `-1` and sets **errno** to the corresponding value:

`EINVAL`	the value of the *name* argument is invalid.

For each of the following conditions occur, the `pathconf()` function returns `-1` and sets **errno** to the corresponding value;

`EACCES`	Search permission is denied for a component of the **path** prefix.
`EINVAL`	The implementation does not support an association of the variable name with the specified file.
`ENAMETOOLONG`	The length of the **path** argument exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
`ENOENT`	The named file does not exist, or the **path** argument points to an empty string.
`ENOTDIR`	A component of the **path** prefix is not a directory.

For each of the following conditions occur, the `fpathconf()` function returns`-1` and sets **errno** to the corresponding value;

EBADF	The **fd** argument is not a valid file descriptor.
EINVAL	The implementation does not support an association of the variable name with the specified file.

COPYRIGHT NOTICE

NAME

`pause` – wait for a signal

SYNOPSIS

`pause()`

DESCRIPTION

`pause` uses the system call `sigpause` to wait for a signal. It always returns with an error, `EINTR`. If the signal is not caught, and is fatal, then, of course, `pause` will never return. If the signal is caught, then `pause` returns after the signal handler exits.

`pause` is often used in conjunction with `alarm`.

RETURN

`pause` always returns -1, and **`errno`** is set to `EINTR`.

SEE ALSO

System Calls - `kill(), sigvec()`

Library Functions - `alarm()`

NAME

perror, psignal,
fperror, fpsignal – print system error message

SYNOPSIS

```
perror(str)
char *str;

fperror(str, fd)
char *str;
int fd;

int sys_nerr;
char **sys_errlist;

#include <signal.h>

psignal(sig, str)
unsigned int sig;
char *str;

fpsignal(sig, str, fd)
unsigned int sig;
char *str;
int fd;

char **sys_siglist;
char **sys_signames;
```

DESCRIPTION

perror prints a string describing the current value of the variable **errno**. If **str** contains a non-empty string, it is printed first followed by a colon. Output is directed to the standard error output (descriptor 2).

psignal prints a string describing signal **sig**. It too writes to the standard error output.

The routines fperror and fpsignal allow the information given by perror or psignal to be sent to any file descriptor **fd**.

By convention, the name of the program incurring the error (or signal) is passed as **str**. Sometimes, the program name is followed by one or two words describing the phase during which the error occurred.

The list of error messages is contained in **sys_errlist**. The number of messages currently available is contained in **sys_nerr**. The list of signal descriptions is contained in **sys_siglist**. The list of standard signal names is contained in **sys_signames**. All messages are stored without a trailing newline.

NOTE

The variable **errno** is set whenever a system routine encounters an error. It is not cleared by `perror` or by successful system routines.

SEE ALSO

System Calls - `sigvec()`

Library Functions - `abort(), exit(), fclose(), strerror()`

NAME

placebuf – obtain a suitably aligned data buffer

SYNOPSIS

```
#include <types.h>

char *placebuf(buf, bufsize, datasize, align)
char *buf;
off_t bufsize;
off_t datasize;
off_t align;
```

DESCRIPTION

The placebuf() function returns a pointer to a suitably aligned buffer of size **datasize**, from a data area of size **bufsize** pointed to by **buf**. The buffer will start on a address that is aligned modulo **align**.

DIAGNOSTICS

If any of the following conditions occur, placebuf() function will return -1 and set **errno** to the corresponding value:

ENOMEM	A buffer area of the requested size could not be allocated.

SEE ALSO

Library Functions - rtcreate(), getattr(), setattr()

NAME

popen, popenr,
pclose, pwait,
pwait3 – create a subprocess with a pipe connection

SYNOPSIS

```
#include <stdio.h>

FILE *popen(command, mode)
char *command, *mode;

FILE *popenr(command, name, mode)
char *command, *name, *mode;

pclose(stream)
FILE *stream;

#include <wait.h>
#include <types.h>
#include <time.h>
#include <resource.h>

int pwait(status)
union wait *status;

int pwait3(status, flag, resources)
union wait *status;
int flag;
struct rusage *resources;
```

DESCRIPTION

`popen` starts a subshell and feeds it the string **command** to be executed. It also creates a pipe and attaches it to the standard input or standard output of the subprocess, according as whether **mode** is the string `w` or `r`. The other end of the pipe is returned to the calling process as a standard I/O stream, ready for use with `fprintf`, `fscanf`, `fgets`, etc. Thus a program can easily filter input or output through a separate system or user-supplied utility program.

`popenr` operates similarly to `popen`, except that the file **`name`** is opened and attached to the standard input or output (whichever is not attached to the pipe) of the subprocess.

A stream opened with `popen` or `popenr` must be closed with `pclose`, never `fclose`.

Because `pclose` calls `wait` to ensure that the child processes have finished, information about child processes started by the calling process in other ways (calls directly to `fork`, for example) might be retrieved inadvertently by `pclose`. Therefore, any process which uses these routines and also must call `wait` or `wait3` should instead use `pwait` or `pwait3`. The information returned by these routines is identical to that returned by the low-level system calls, but the user is ensured that no processes will be "lost".

DIAGNOSTICS

`popen` and `popenr` return NULL if the process cannot be created. If the child process fails, a valid stream will be returned, but end of file will be reached immediately.

`pclose` returns `0` if successful. It returns `EOF` if unsuccessful, or the stream is not associated with the `popen` or `popenr` command.

SEE ALSO

System Calls - `pipe(), wait()`

Library Functions - `fopen()`

NAME

`printf, fprintf, sprintf,`
`unlocked_printf,`
`unlocked_fprintf,`
`unlocked_sprintf` – print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf(format [ , arg ] . . . )
char *format;

int fprintf(stream, format [ , arg ] . . . )
FILE *stream;
char *format;

int sprintf(s, format [ , arg ] . . . )
char *s, format;

int unlocked_printf(format [ , arg ] . . . )
char *format;

int unlocked_fprintf(stream, format [ , arg  ] . . . )
FILE *stream;
char *format;

int unlocked_sprintf(s, format [ , arg ] . . . )
char *s, format;
```

DESCRIPTION

Each of these functions converts, formats, and prints its arguments under control of the format. The format is a character string that contains two types of objects: plain characters, which are simply copied into the output stream, and conversion specifications, each of which causes zero or more additional arguments to be fetched. Results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are ignored.

`printf` writes the formatted data to standard output. `fprintf` writes to the named stream. `sprintf` writes into a user-supplied buffer, terminating the data with a null. (The calling environment must ensure that enough space is available.) Each routine returns the number of bytes sent to the output file or buffer.

`printf`, `fprintf`, and `sprintf` obtain mutually-exclusive access to the relevant stream, and are therefore thread-safe. `unlocked_printf`, `unlocked_fprintf`, and `unlocked_sprintf` are identical to their thread-safe brethren but perform no locking. If the application wishes to do its own locking, it can do so by using `flockfile` and `funlockfile`.

Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

1. An optional flag which modifies the meaning of the conversion specification (see below).
2. An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag has been given), with spaces, to the field width. A leading zero indicates zeros should be used instead of spaces.
3. A precision which gives the maximum number of characters to be printed from a string, or the number of digits to be printed to the right of the decimal point for `float` or `double` formats.
4. An optional `l` specifying that a following `d`, `o`, `u`, or `x` conversion character applies to a `long` arg.
5. A character that indicates the type of conversion to be applied.

The only flag character is the minus sign (–). When used, the result of the conversion will be left-justified within the field. The conversion characters and their meanings are:

d,o,u,x,X The `int` (or `long` if a `l` appeared before the conversion character) argument is converted to signed decimal, unsigned octal, unsigned decimal, or unsigned hexadecimal notation. The **x** and **X** conversions both produce unsigned hexadecimal representations, but **x** produces lower case letters while **X** produces upper case.

f — The `float` or `double` argument is converted to decimal notation in the style:

`[-]<digits>.<digits>`

The number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are written; if the precision is zero, no decimal point appears.

e, g — The `float` or double argument is converted to the style:

`[-]<digits>.<digits>e(+|-)<digits>`

One digit is placed before the decimal point, and the number of digits after is equal to the precision. If the precision is missing, six digits are written; if the precision is zero, no decimal point appears.

c — The character argument is printed.

s — The argument is taken to be a string (character pointer) and characters from the string are printed until a null character (`\0`) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, all characters up to the first null character are printed. A NULL argument will yield undefined results. Note that a NULL argument is distinct from an argument which points to the empty string, which consumes no space in the final output.

% — Print a `%`; no argument is converted.

Only in the case of strings does a non-existent or small field width cause truncation of a field. If the result of any numeric conversion is wider than the field width, the field is simply expanded to contain the conversion result.

NOTE

In many cases, there is no positively safe way to use `sprintf`.

SEE ALSO

Library Functions - `fopen(), flockfile(), funlockfile()`

NAME

pthread_attr_create,
pthread_attr_delete – initialize a pthread_attr_t structure

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_create(attr)
pthread_attr_t *attr;

int pthread_attr_delete(attr)
pthread_attr_t *attr;
```

DESCRIPTION

pthread_attr_create initializes a **pthread_attr_t** structure at address **attr** to system default values. pthread_attr_delete does not do anything and can be used as a handy source of the value 0, which it returns.

System Thread Creation Attributes

The **attr** argument refers to a structure of type **pthread_attr_t**, which contains the following members:

pthread_attr_stacksize
: minimum pthread stack size in bytes

pthread_attr_prio
: pthread scheduling priority

pthread_attr_inheritsched
: inherit scheduling parameters from creating pthread

pthread_attr_sched
: initial pthread scheduler

pthread_attr_scope
: initial pthread scheduling scope

`pthread_attr_stacksize` determines the pthread's stack size in bytes. This number is a minimum; the stack of the **st** will probably be rounded up to an integral number of pages.

`pthread_attr_prio` sets the pthread's scheduling priority, if the pthread is not inheriting its scheduling parameters from the creating thread. `pthread_attr_prio` must be a number between 0 and 255. 0 is a bad value to use, as LynxOS runs a null process at that priority in an infinite loop.

`pthread_attr_inheritsched` determines whether the pthread created with these attributes will inherit its scheduling algorithm and priority from its creator or not. If `pthread_attr_inheritsched` is nonzero, then any pthread created with these attributes will be initialized with the scheduling priority and policy of the pthread executing `pthread_create`. Otherwise, the new pthread will have its scheduling policy and parameters initialized according to the `pthread_attr_prio` and `pthread_attr_sched` attributes.

`pthread_attr_scope` is there because POSIX demands it be there. It serves no useful purpose in LynxOS and can be ignored. The default values for these attributes are:

`pthread_attr_stacksize`	`THREAD_DEFAULT_STACK`
`pthread_attr_prio`	`0`
`pthread_attr_inheritsched`	`1`
`pthread_attr_sched`	`SCHED_FIFO`

NOTE

Conforming POSIX.4a applications, for portability reasons, should use `pthread_attr_create` and `pthread_attr_delete` to initialize POSIX thread attributes structures.

DIAGNOSTICS

`pthread_attr_create` always returns 0. If the address passed to `pthread_attr_create` is bad, then the pthread may receive a `SIGSEGV` or a `SIGBUS` signal.

CONFORMANCE

`pthread_attr_create` and `pthread_attr_delete` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_create()`, `pthread_attr_getsched()`, `pthread_attr_getinheritsched()`, `pthread_attr_getprio()`, `pthread_attr_getstacksize()`

NAME

pthread_attr_getinheritsched, pthread_attr_setinheritsched – get/set scheduler inheritance value in attributes structure

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getinheritsched(attr)
pthread_attr_t *attr;

int pthread_attr_setinheritsched(attr, inheritscheduler)
pthread_attr_t *attr;
int inheritscheduler;
```

DESCRIPTION

pthread_attr_getinheritsched returns the scheduler inheritance value stored in the attributes structure **attr**. pthread_attr_setinheritsched sets the scheduler inheritance value in **attr** to **inheritscheduler**.

The scheduler inheritance value stored in **attr** is used, in part, to determine the scheduling attributes of any threads created using **attr**. See pthread_create for a complete description of this determination.

NOTE

The scheduler inheritance value may be set to a bogus value using pthread_attr_setinheritsched; this may result in the attributes structure being unusable until the scheduler inheritance value is set correctly.

DIAGNOSTICS

pthread_attr_getinheritsched returns the scheduler inheritance value as previously set in **attr**. pthread_attr_setinheritsched always returns 0. If the **attr** address passed to pthread_attr_getinheritsched or pthread_attr_setinheritsched is bad, then the pthread may receive a SIGSEGV or a SIGBUS signal.

CONFORMANCE

`pthread_attr_getinheritsched` and `pthread_attr_setinheritsched` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_attr_create()`, `pthread_create()`

NAME

`pthread_attr_getprio`, `pthread_attr_setprio` - get/set scheduler priority value in attributes structure

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getprio(attr)
pthread_attr_t *attr;

int pthread_attr_setprio(attr, prio)
pthread_attr_t *attr;
int prio;
```

DESCRIPTION

`pthread_attr_getprio` returns the scheduler priority value stored in the attributes structure **`attr`**. `pthread_attr_setprio` sets the scheduler priority value in **`attr`** to **`prio`**.

The scheduler priority value stored in **`attr`** is used, in part, to determine the scheduling attributes of any threads created using **`attr`**. See `pthread_create` for a complete description of this determination.

NOTE

The scheduler priority value may be set to a bogus value using `pthread_attr_setprio`; this may result in the attributes structure being unusable until the scheduler priority value is set correctly.

DIAGNOSTICS

`pthread_attr_getprio` returns the scheduler priority value as previously set in **`attr`**. `pthread_attr_setprio` always returns 0. If the **`attr`** address passed to `pthread_attr_getprio` or `pthread_attr_setprio` is bad, then the pthread may receive a `SIGSEGV` or a `SIGBUS` signal.

CONFORMANCE

`pthread_attr_getprio` and `pthread_attr_setprio` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_attr_create()`, `pthread_create()`

NAME

pthread_attr_getsched,
pthread_attr_setsched - get/set scheduler value in attributes structure

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getsched(attr)
pthread_attr_t *attr;

int pthread_attr_setsched(attr, scheduler)
pthread_attr_t *attr;
int scheduler;
```

DESCRIPTION

pthread_attr_getsched returns the scheduler value stored in the attributes structure **attr**. pthread_attr_setsched sets the scheduler value in **attr** to **scheduler**.

The scheduler value stored in **attr** is used, in part, to determine the scheduling attributes of any threads created using **attr**. See pthread_create for a complete description of this determination.

NOTE

scheduler may be set to a bogus value using pthread_attr_setsched; this may result in the attributes structure being unusable until the scheduler value is set correctly.

DIAGNOSTICS

pthread_attr_getsched returns the scheduler value as previously set in **attr**. pthread_attr_setsched always returns 0. If the **attr** address passed to pthread_attr_getsched or pthread_attr_setsched is bad, then the pthread may receive a SIGSEGV or a SIGBUS signal.

CONFORMANCE

`pthread_attr_getsched` and `pthread_attr_setsched` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_attr_create(), pthread_create()`

NAME

pthread_attr_getstacksize,
pthread_attr_setstacksize – get/set stack size value in attributes structure

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getstacksize(attr)
pthread_attr_t *attr;

int pthread_attr_setstacksize(attr, stacksize)
pthread_attr_t *attr;
long stacksize;
```

DESCRIPTION

pthread_attr_getstacksize returns the stack size value stored in the attributes structure **attr**. pthread_attr_setstacksize sets the stack size value in **attr** to **stacksize**.

The stack size value stored in **attr** is used to determine the minimum number of bytes required for the stack of any threads created using **attr**.

NOTE

The stack size value may be set to an impossibly bogus value using pthread_attr_setstacksize; this may result in the attributes structure being unusable until the stack size value is set correctly.

DIAGNOSTICS

pthread_attr_getstacksize returns the stack size value as previously set in **attr**. pthread_attr_setstacksize always returns 0. If the **attr** address passed to pthread_attr_getstacksize or pthread_attr_setstacksize is bad, then the pthread may receive a SIGSEGV or a SIGBUS signal.

CONFORMANCE

`pthread_attr_getstacksize` and `pthread_attr_setstacksize` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_attr_create()`, `pthread_create()`

NAME

`pthread_cancel` – cancel a pthread

SYNOPSIS

```
#include <pthread.h>

int pthread_cancel(tid)
pthread_t tid;
```

DESCRIPTION

`pthread_cancel` cancels or arranges the cancellation of the pthread whose ID is given by **`tid`**. It is not possible to cancel a pthread in another process.

If the pthread's general cancelability state is `TRUE` and the pthread's asynchronous cancelability state is also `TRUE`, the pthread will be immediately canceled. If the system thread's general cancelability state is `TRUE` but the asynchronous cancelability state ID `FALSE`, then the pthread will be canceled the next time it reaches a cancellation point. If the pthread's general cancelability state is `FALSE`, then the cancel request is held pending until the pthread's cancelability changes.

System thread cancellation points occur:

> when the pthread issues a `pthread_cond_wait`.
>
> when the pthread issues a `pthread_testcancel`.
>
> when the pthread issues a `pthread_join`.

When the pthread is canceled, any cleanup routines on the pthread's cleanup stack will be executed in LIFO order by the pthread. A value of `THREAD_CANCELED` is given as the pthread's exit status to any pthreads performing a `pthread_join` with the canceled pthread.

A pthread's initial general cancellation state is `TRUE`. A pthread's initial asynchronous cancellation state is `FALSE`.

NOTE

A pthread that has called `pthread_exit` or already been successfully canceled, but which has not been detached with `pthread_detach`, can still be successfully canceled with `pthread_cancel`. Such a call to `pthread_cancel` will have no effect; but it will also not fail.

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

DIAGNOSTICS

`pthread_cancel` 0 if the target thread exists, or -1 upon error. If an error is detected, **`errno`** will be set to one of the following values:

`ESRCH`	The value given by **`tid`** does not refer to any known pthread.

CONFORMANCE

`pthread_cancel` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

System Calls -	`pthread_exit()`, `pthread_join()`, `pthread_detach()`, `pthread_cond_wait()`
Library Functions -	`pthread_pushrecover()`, `pthread_poprecover()`, `pthread_testcancel()`, `pthread_setcancel()`, `pthread_new()`
Example -	`pthread_testcancel.c`

NAME

pthread_cleanup_push,
pthread_cleanup_pop – push/pop recovery routines

SYNOPSIS

```
#include <pthread.h>

int pthread_cleanup_push(func, arg)
void(*func)();
void *arg;

int pthread_cleanup_pop(execute)
int execute;
```

DESCRIPTION

pthread_cleanup_push and pthread_cleanup_pop manage a stack of cleanup routines to be executed if the calling thread is canceled via pthread_cancel or pthread_testcancel, or if the thread itself exits (either prematurely, by using pthread_exit, or by returning from the function at which the thread began executing). The cleanup stack is initially empty; the cleanup routine **func** and its single argument **arg** are pushed, as a single stack element, on the cleanup stack using pthread_cleanup_push. The stack is popped using pthread_cleanup_pop; if the **execute** argument is nonzero, then the function that was popped is called with the argument that was popped. Otherwise, the popped stack element is simply discarded. In the event that the thread exits, (due either to being canceled, to calling pthread_exit itself, or to returning from its initial function), the cleanup stack is popped until empty, each function being called with the pushed parameter in LIFO order.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, SIGTHREADKILL. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle SIGTHREADKILL.

DIAGNOSTICS

`pthread_cleanup_push` returns 0 on successful completion of the push. If the function fails, it returns -1 and sets **errno** to one of the following values:

`ENOSPC`	Insufficient memory to push the new stack element.

`pthread_cleanup_pop` returns 0 on successful completion of the pop. If the function fails, it returns -1 and sets **errno** to one of the following values:

`EAGAIN`	The cleanup stack is empty.

CONFORMANCE

`pthread_cleanup_push` and `pthread_cleanup_pop` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	`pthread_cancel()`, `pthread_setcancel()`, `pthread_testcancel()`
Example -	`pthread_cleanup_push.c`, `pthread_cleanup_pop.c`.

NAME

pthread_condattr_create,
pthread_condattr_delete – initialize pthread_condattr_t to default values

SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_create(condattr)
pthread_condattr_t *condattr;

int pthread_condattr_delete(condattr)
pthread_condattr_t *condattr;
```

DESCRIPTION

A POSIX condition variable is created using an attributes structure to define the characteristics of the condition variable. This attributes structure must be initialized using pthread_condattr_create. Any resources that may have been allocated when creating a condition variable attributes structure can be de-allocated using pthread_condattr_delete.

NOTE

Conforming POSIX.4a applications, for portability reasons, should use pthread_condattr_create and pthread_condattr_delete to initialize condition variable attributes structures.

DIAGNOSTICS

pthread_condattr_create and pthread_condattr_delete always return 0. They do not fail, since, under the current system, they do not do anything. However, if the address passed to pthread_condattr_create is bad, then the pthread may receive a SIGSEGV or a SIGBUS signal.

CONFORMANCE

`pthread_condattr_create` and `pthread_condattr_delete` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_cond_init()`

NAME

pthread_cond_broadcast – signal a condition to all awaiting threads

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_broadcast(cond)
pthread_cond_t *cond;
```

DESCRIPTION

pthread_cond_broadcast is used to signal a condition's occurrence. When a pthread_cond_broadcast occurs, all threads waiting for **cond** are awakened. Those threads all must then re-acquire the mutex lock they gave up in waiting for the condition variable (via an implicit pthread_mutex_lock), and then return to the user. The caller of pthread_cond_broadcast need not possess the mutex lock. Only one thread at a time can posses the mutex lock. Therefore, the threads waiting on **cond** will continue their execution one by one as they acquire their mutex in turn.

A condition variable is a synchronization mechanism used by two or more threads of execution to synchronize their access to shared resources. A condition variable is so called because it is used to indicate a certain "condition", or logical predicate, regarding the shared resource. The typical use for a condition variable is for a thread to examine the logical predicate, and then, seeing that the predicate is not yet satisfied, to wait on the condition variable.

A condition variable is a stateless object. A thread performing a pthread_cond_wait will only be awakened by threads that do pthread_cond_signal or pthread_cond_broadcast after the waiting thread is blocked. There is no "locked" or "unlocked" state associated with a condition variable.

Because of the stateless nature of condition variables, it is unsafe in the abstract to use a condition variable without the protection of a mutex. The logical predicate must be checked while the thread has mutually-exclusive access to the resource; otherwise a race condition exists, wherein the thread could check the condition and find it false, then, immediately before blocking on the condition variable, another thread could preempt and make the logical predicate true. When the first thread ran again, it would immediately block forever. This is an abstract argument, which serves to motivate why a mutex lock is required before a condition variable can be waited upon. Obviously, the mutex should be acquired before the condition variable is signalled, as well; however, it is not common practice to require this, and so we do not.

It is common practice for condition variables to be spuriously signalled, thus waking up the blocked thread when the corresponding logical predicate is not yet true. For this reason, condition variable blocking should always be done in a while loop, checking for the truth of the predicate both before and after blocking on the condition variable. The following code is the incorrect way to use a condition variable:

```
pthread_mutex_lock(m);
if (! condition_satisfied)
pthread_cond_wait(c,m);
/*
* Now we assume that condition_satisfied is TRUE,
* and this may not be the case
*/
...        /* Modify critical shared data */
pthread_mutex_unlock(m);
```

The correct way to use a condition variable is to use a `while` construct in place of an `if`:

```
pthread_mutex_lock(m);
while (! condition_satisfied)
  pthread_cond_wait(c,m);
/*
* Now we know that condition_satisfied is TRUE,
* because we checked it after waiting.
*/
...        /* Modify critical shared data */
pthread_mutex_unlock(m);
```

NOTE

Using a condition variable that has not been initialized with `pthread_cond_init`, or which has been destroyed with `pthread_cond_destroy`, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`pthread_cond_broadcast` returns 0 when successful, or -1 upon error. If the call fails, **`errno`** will contain one of the following:

`EINVAL`	`cv` does not refer to a valid condition variable.

CONFORMANCE

`pthread_cond_broadcast` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	`pthread_mutex_lock()`, `pthread_mutex_init()`, `pthread_mutex_unlock()`, `pthread_cond_wait()`, `pthread_cond_signal()`
Example -	`pthread_cond_broadcast.c`

NAME

`pthread_cond_destroy` – delete a user condition variable

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_destroy(cond)
pthread_cond_t *cond;
```

DESCRIPTION

`pthread_cond_destroy` frees up the system resources for the condition variable pointed to by **`cond`** and removes it from the list of active condition variables.

DIAGNOSTICS

`pthread_cond_destroy` returns 0 if the condition variable was destroyed, or -1 if not. If the call fails, **`errno`** will contain one of the following:

`EBUSY`	Some thread is waiting on the condition variable.
`EINVAL`	Condition variable does not exist.

CONFORMANCE

`pthread_con_destroy` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_cond_init()`, `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`

NAME

`pthread_cond_init` – initialize a user condition variable

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_init(cond, attr)
pthread_cond_t *cond;
pthread_cond_attr_t attr;
```

DESCRIPTION

`pthread_cond_init` initializes the user condition variable referred to by **`cond`**. The attributes object **`attr`** is required by POSIX, and should be initialized using `pthread_condattr_create`. However, this object is not used by LynxOS. It is present for standards compliance only.

A condition variable is a synchronization variable that can be used by threads to atomically release a mutex lock and wait for some condition to be satisfied (hence the name). When the condition is satisfied, presumably by some other thread, that thread will execute either `pthread_cond_signal` or `pthread_cond_broadcast`. This will cause one or all of the waiting threads, respectively, to awaken, re-acquire the mutex they released, and then return from the call. The use of the mutex ensures that only one thread accesses a shared resource at a time. The typical use for a condition variable is for a thread to examine the logical predicate, and then, seeing that the predicate is not yet satisfied, to wait on the condition variable.

A condition variable is a stateless object. A thread performing a `pthread_cond_wait` will only be awakened by threads that do `pthread_cond_signal` or `pthread_cond_broadcast` after the waiting thread is blocked. There is no “locked” or “unlocked” state associated with a condition variable.

Because of the stateless nature of condition variables, it is unsafe in the abstract to use a condition variable without the protection of a mutex. The logical predicate must be checked while the thread has mutually exclusive access to the resource; otherwise a race condition exists, wherein the thread could check the condition and find it false, then, immediately before blocking on the condition variable, another thread could preempt and make the logical predicate true. When the first thread ran again, it would immediately block forever. This is an abstract argument, which serves to motivate why a mutex lock is required before a condition variable can be waited upon. Obviously, the mutex should be acquired before the condition variable is signalled, as well; however, it is not common practice to require this, and so we do not.

It is common practice for condition variables to be spuriously signalled, thus waking up the blocked thread when the corresponding logical predicate is not yet true. For this reason, condition variable blocking should always be done in a while loop, checking for the truth of the predicate both before and after blocking on the condition variable. The following code is the incorrect way to use a condition variable:

```
pthread_mutex_lock(m);
if (! condition_satisfied)
  pthread_cond_wait(c,m);
/*
* Now we assume that condition_satisfied is TRUE,
* and this may not be the case
*/
...        /* Modify critical shared data */
pthread_mutex_unlock(m);
```

The correct way to use a condition variable is to use a `while` construct in place of an `if`:

```
pthread_mutex_lock&m);
while (! condition_satisfied)
  pthread_cond_wait(c,m);
/*
* Now we know that condition_satisfied is TRUE,
* because we checked it after waiting.
*/
...        /* Modify critical shared data */
pthread_mutex_unlock(m);
```

NOTE

Using a condition variable that has not been initialized with `pthread_cond_init`, or which has been destroyed with `pthread_cond_destroy`, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`pthread_cond_init` returns -1 if it fails to get the condition variable. If the call fails, **`errno`** will contain one of the following:

`EAGAIN`	No more room for user condition variables.

CONFORMANCE

`pthread_con_init` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_cond_destroy()`, `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`, `pthread_condattr_create()`

NAME

`pthread_cond_signal` – signal a condition to a single awaiting thread

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_signal(cond)
pthread_cond_t *cond;
```

DESCRIPTION

`pthread_cond_signal` is used to signal a condition's occurrence. When a `pthread_cond_signal` occurs, the highest priority thread that has been waiting the longest for **cond** is awakened. That thread must then re-acquire the mutex lock it gave up in waiting for the condition variable (via an implicit `pthread_mutex_lock`), and then return to the user. The caller of `pthread_cond_signal` need not possess the mutex lock.

A condition variable is a synchronization mechanism used by two or more threads of execution to synchronize their access to shared resources. A condition variable is so called because it is used to indicate a certain "condition", or logical predicate, regarding the shared resource. The typical use for a condition variable is for a thread to examine the logical predicate, and then, seeing that the predicate is not yet satisfied, to wait on the condition variable.

A condition variable is a stateless object. A thread performing a `pthread_cond_wait` will only be awakened by threads that do `pthread_cond_signal` or `pthread_cond_broadcast` after the waiting thread is blocked. There is no "locked" or "unlocked" state associated with a condition variable.

Because of the stateless nature of condition variables, it is unsafe in the abstract to use a condition variable without the protection of a mutex. The logical predicate must be checked while the thread has mutually exclusive access to the resource; otherwise a race condition exists, wherein the thread could check the condition and find it false, then, immediately before blocking on the condition variable, another thread could preempt and make the logical predicate true. When the first thread ran again, it would immediately block forever. This is an abstract argument, which serves to motivate why a mutex lock is required before a condition variable can be waited upon. Obviously, the mutex should be acquired before the condition variable is signalled, as well; however, it is not common practice to require this, and so we do not.

It is common practice for condition variables to be spuriously signalled, thus waking up the blocked thread when the corresponding logical predicate is not yet true. For this reason, condition variable blocking should always be done in a while loop, checking for the truth of the predicate both before and after blocking on the condition variable. The following code is the incorrect way to use a condition variable:

```
pthread_mutex_lock(m);
if (! condition_satisfied)
  pthread_cond_wait(c,m);
/*
 * Now we assume that condition_satisfied is TRUE,
 * and this may not be the case
 */
...        /* Modify critical shared data */
pthread_mutex_unlock(m);
```

The correct way to use a condition variable is to use a `while` construct in place of an `if`:

```
pthread_mutex_lock(m);
while (! condition_satisfied)
  pthread_cond_wait(c,m);
/*
 * Now we know that condition_satisfied is TRUE,
 * because we checked it after waiting.
 */
 ...        /* Modify critical shared data */
pthread_mutex_unlock(m);
```

NOTE

Using a condition variable that has not been initialized with `pthread_cond_init`, or which has been destroyed with `pthread_cond_destroy`, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`pthread_cond_signal` returns 0 when successful, or -1 upon error. If the call fails **`errno`** will contain one of the following:

`EINVAL`	`cv` does not refer to a valid condition variable.

CONFORMANCE

`pthread_cond_signal` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	`pthread_mutex_lock()`, `pthread_mutex_init()`, `pthread_mutex_unlock()`, `pthread_cond_wait()`, `pthread_cond_broadcast()`
Example -	`pthread_cond_signal.c`

NAME

pthread_cond_wait,
pthread_cond_timedwait – wait for a condition to be signalled

SYNOPSIS

```
#include <pthread.h>

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

int pthread_cond_timedwait(cond, mutex, timeout)
pthread_cond_t *cond;
pthread_mutex_t *mutex;
struct timespec *timeout;
```

DESCRIPTION

pthread_cond_wait is used by the owner of mutex **mutex** to wait for a condition to be signalled by another thread. The mutex ownership is released and the thread blocks awaiting a pthread_cond_broadcast or pthread_cond_signal on the condition variable **cond**.

A condition variable is a synchronization mechanism used by two or more threads of execution to synchronize their access to shared resources. A condition variable is so called because it is used to indicate a certain "condition", or logical predicate, regarding the shared resource. The typical use for a condition variable is for a thread to examine the logical predicate, and then, seeing that the predicate is not yet satisfied, to wait on the condition variable.

A condition variable is a stateless object. A thread performing a pthread_cond_wait will only be awakened by threads that do pthread_cond_signal or pthread_cond_broadcast after the waiting thread is blocked. There is no "locked" or "unlocked" state associated with a condition variable.

Because of the stateless nature of condition variables, it is unsafe in the abstract to use a condition variable without the protection of a mutex. The logical predicate must be checked while the thread has mutually exclusive access to the resource; otherwise a race condition exists, wherein the thread could check the condition and find it false, then, immediately before blocking on the condition variable, another thread could preempt and make the logical predicate true. When the first thread ran again, it would immediately block forever. This is an abstract argument, which serves to motivate why a mutex lock is required before a condition variable can be waited upon. Obviously, the mutex should be acquired before the condition variable is signalled, as well; however, it is not common practice to require this, and so we do not.

It is common practice for condition variables to be spuriously signalled, thus waking up the blocked thread when the corresponding logical predicate is not yet true. For this reason, condition variable blocking should always be done in a while loop, checking for the truth of the predicate both before and after blocking on the condition variable. The following code is the incorrect way to use a condition variable:

```
pthread_mutex_lock(m);
if (! condition_satisfied)
  pthread_cond_wait(c,m);
/*
  * Now we assume that condition_satisfied is TRUE,
 * and this may not be the case
 */
 ...        /* Modify critical shared data */
pthread_mutex_unlock(m);
```

The correct way to use a condition variable is to use a `while` construct in place of an `if`:

```
pthread_mutex_lock(m);
while (! condition_satisfied)
  pthread_cond_wait(c,m);
/*
 * Now we know that condition_satisfied is TRUE,
 * because we checked it after waiting.
 */
 ...        /* Modify critical shared data */
pthread_mutex_unlock(m);
```

`pthread_cond_timedwait` is identical to `pthread_cond_wait`, except for that the **`timeout`** argument specifies a maximum amount of time to await the mutex lock. If the argument is NULL, the thread blocks indefinitely. If the argument refers to a time of duration 0, the thread will not block at all.

Once the thread is awakened by the signalling of the condition or by a **`timeout`**, the thread performs an implicit `pthread_mutex_enter` to regain the mutex lock it gave up when it performed the `pthread_cond_wait`.

Behavior With Respect To Thread Cancellation

A call to `pthread_cond_wait` is a thread cancellation point. That is, if a thread has set general cancelability and not asynchronous cancelability, and if a cancellation request is issued against this thread before or during the duration of the call to `pthread_cond_wait`, the thread will be canceled.

NOTE

Using a condition variable that has not been initialized with `pthread_cond_init`, or which has been destroyed with `pthread_cond_destroy`, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`pthread_cond_wait` and `pthread_cond_timedwait` return 0 when successful, or -1 upon error. Even if the call fails, the mutex is reacquired. If the call fails, **`errno`** will contain one of the following:

`EINVAL`	`cv` does not refer to a valid condition variable.
`EINVAL`	**`mutex`** does not refer to a valid mutex.
`EINTR`	The wait was interrupted by a signal delivered to the thread.
`EINTR`	(`pthread_cond_timedwait` only) The wait timed out without receiving a `pthread_cond_signal`.

NOTE

In any case, no matter what the success or failure return is of `pthread_cond_wait`, the mutex **mutex** is always re-acquired.

CONFORMANCE

`pthread_cond_wait` and `pthread_cond_timedwait` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `mutex_exit()`, `mutex_create()`, `mutex_enter()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`

Example - `pthread_cond_timedwait.c`

NAME

pthread_create – create a new thread

SYNOPSIS

```
#include <pthread.h>

int pthread_create(tidp, attr, startroutine, arg)
pthread_t *tidp;
pthread_attr_t attr;
void *(*start_routine)();
void *arg;
```

DESCRIPTION

pthread_create creates a new pthread in the current process context. The pthread is initialized to begin running at the routine **start_routine**, which is passed the single parameter **arg**. The various creation characteristics of the newly-created pthread are determined by **attr**. Upon successful pthread creation, the ID of the new pthread is stored at the location pointed to by **tidp**. The pthread is made runnable immediately upon creation.

pthread Creation Attributes

The **attr** argument is a structure of type **pthread_attr_t**, which contains the following members:

pthread_attr_stacksizeminimum pthread stack size in bytes

pthread_attr_priopthread scheduling priority

pthread_attr_inheritschedinherit scheduling parameters from creating pthread

pthread_attr_schedinitial pthread scheduler

pthread_attr_scopeinitial pthread scheduling scope

pthread_attr_stacksize determines the pthread's stack size in bytes. This number is a minimum; the stack will probably be rounded up to an integral number of pages.

pthread_attr_prio sets the pthread's scheduling priority, if the pthread is not inheriting its scheduling parameters from the creating thread. pthread_attr_prio must be a number between 0 and 255. 0 is a bad value to use, as LynxOS runs a null process at that priority in an infinite loop.

pthread_attr_inheritsched determines whether the pthread created with these attributes will inherit its scheduling algorithm and priority from its creator or not. If pthread_attr_inheritsched is nonzero, then any pthread created with these attributes will be initialized with the scheduling priority and policy of the pthread executing pthread_create. Otherwise, the new pthread will have its scheduling policy and parameters initialized according to the pthread_attr_prio and pthread_attr_sched attributes.

pthread_attr_scope is there because POSIX demands it be there. It serves no useful purpose in LynxOS and can be ignored. The default values for these attributes are:

```
pthread_attr_stacksizeTHREAD_DEFAULT_STACK
pthread_attr_prio0
pthread_attr_inheritsched1
pthread_attr_schedSCHED_FIFO
```

The pthread's initial cancellation state is CANCEL_GENERAL.

DIAGNOSTICS

pthread_create returns 0 when successful, or -1 upon error. If the call fails, **errno** will contain one of the following:

EINVAL	The stack size given is bogus.
EINVAL	The priority value is not a valid priority.
EFAULT	**start_routine** does not point to a location in the process' address space.
ENOMEM	Out of room for pthreads.

CONFORMANCE

`pthread_create` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

NOTE

In Draft 6 of the POSIX 1003.4a (threads) standard, the parameter `attr` is defined as

```
pthread_attr_t *attr;
```

instead of

```
pthread_attr_t attr;
```

SEE ALSO

System Calls - `fork()`, `execve()`

Library Functions - `pthread_exit()`, `pthread_join()`, `pthread_attr_create()`, `pthread_setcancel()`, `pthread_testcancel()`, `pthread_cancel()`

NAME

`pthread_detach` – free resources associated with a pthread

SYNOPSIS

```
#include <pthread.h>

int pthread_detach(tid)
pthread_t *tid;
```

DESCRIPTION

`pthread_detach` indicates to the system that no further pthreads will be attempting to join with the pthread whose ID is given by **`tid`**. The resources associated with the pthread can be immediately freed when it `pthread_exits` or is `pthread_canceled`.

`pthread_detach` does not free those resources necessary to run the pthread unless it has already `pthread_exited` or been successfully `pthread_canceled`. Thus, a pthread can be detached while still running and will continue to execute as before.

Subsequent `pthread_joins` with this system thread will return with **`errno`** set to `EINVAL`.

DIAGNOSTICS

`pthread_detach` returns 0 when successful and -1 when unsuccessful. If the call fails, **`errno`** will contain one of the following:

`EINVAL`	The pthread ID given does not refer to a currently-existing pthread

CONFORMANCE

`pthread_detach` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_join()`, `pthread_exit()`, `pthread_create()`

NAME

`pthread_equal` – compare two thread IDs

SYNOPSIS

```
#include <pthread.h>

int pthread_equal(tid1, tid2)
pthread_t tid1, tid2;
```

DESCRIPTION

`pthread_equal` returns true if **tid1** and **tid2** are equal; i.e., if they refer to the same thread.

DIAGNOSTICS

`pthread_equal` never fails. It returns 0 if the two thread IDs are not equal.

CONFORMANCE

`pthread_equal` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_create()`, `pthread_exit()`, `pthread_detach()`, `pthread_join()`, `pthread_self()`

NAME

`pthread_exit` – terminate the calling pthread

SYNOPSIS

```
#include <pthread.h>

void pthread_exit(status)
void *status;
```

DESCRIPTION

`pthread_exit` terminates the pthread calling it. The value **status** becomes the pthread's `exit status`, and is passed to other pthreads who are doing a `pthread_join` with this pthread. The system thread's `exit status` remains available to other pthreads via `pthread_join` until a `pthread_detach` is performed to free up all resources associated with this pthread.

If any cleanup routines have been pushed onto the thread's cleanup stack and not popped when the thread calls `pthread_exit`, then each routine and its argument shall be popped off the thread cleanup stack and called with that argument.

DIAGNOSTICS

`pthread_exit` never returns.

CONFORMANCE

`pthread_exit` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_join()`, `pthread_detach()`, `pthread_create()`, `pthread_cancel()`, `pthread_cleanup_push()`, `pthread_cleanup_pop()`

NAME

`pthread_getprio` – get system thread scheduling priority

SYNOPSIS

```
#include <pthread.h>

int pthread_getprio(tid)
pthread_t tid;
```

DESCRIPTION

`pthread_getprio` returns the scheduling priority of the thread identified by **tid**.

DIAGNOSTICS

If **tid** refers to a currently existing thread, `pthread_getprio` returns its priority. Otherwise, `pthread_getprio` returns -1 and sets **errno** to one of the following values:

ESRCH	The thread referred to by **tid** does not exist.

CONFORMANCE

`pthread_getprio` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	`pthread_create()`, `pthread_attr_getprio()`, `pthread_attr_setprio()`, `pthread_setprio()`
Example -	`pthread_setprio.c`

NAME

`pthread_getspecific`,
`pthread_setspecific` – retrieve, set a thread-specific data value

SYNOPSIS

```
#include <pthread.h>

int pthread_setspecific(key, value)
int key;
void *value;

int pthread_getspecific(key, value)
int key;
void **value;
```

DESCRIPTION

`pthread_getspecific` retrieves the data value associated with **`key`** for the calling thread, storing the value in **`value`**. `pthread_setspecific` sets the data value associated with **`key`** to **`value`**. Different threads have different values associated with a given **`key`**, allowing a limited form of private thread storage.

In the case of `pthread_setspecific`, if this thread has previously set a value for **`key`**, and the destructor for this key is not NULL, then the destructor function will be called with the old value before the new value is saved. The destructor will not be called until a value has been set with `pthread_setspecific`.

DIAGNOSTICS

`pthread_getspecific` stores the data value in **`value`** and returns 0 if it successfully retrieved the data value associated with **`key`**. `pthread_setspecific` returns 0 if it successfully stored the new **`value`** associated with **`key`**. If either call fails, it will return -1.

If the call fails, **`errno`** will contain one of the following:

`EINVAL`	Invalid key value. The key was not created using `pthread_keycreate`.

NOTE

If the destructor function address supplied to `pthread_keycreate` is bogus, then subsequent calls to `pthread_keycreate` may cause the program to receive a signal. The destructor address is not checked for validity; only for NULL or non-NULL.

CONFORMANCE

`pthread_getspecific` and `pthread_setspecific` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_keycreate()`

NAME

`pthread_join` – await the termination of another pthread.

SYNOPSIS

```
#include <pthread.h>

int pthread_join(tid, status)
pthread_t tid;
void **status;
```

DESCRIPTION

`pthread_join` blocks the calling pthread until the pthread whose ID is given by **tid** terminates. At that time, the pthread performing the `pthread_join` is unblocked and the exit status of the terminated pthread is returned in the **status** argument.

Behavior With Respect To Thread Cancellation

A call to `pthread_join` is a thread cancellation point. That is, if a thread has set general cancelability and not asynchronous cancelability, and if a cancellation request is issued against this thread before or during the duration of the call to `pthread_join`, the thread will be canceled. Special exit statuses are used to indicate that a pthread was `pthread_canceled` by another pthread rather than `pthread_exiting` normally.

DIAGNOSTICS

`pthread_join` returns 0 when successful, and -1 when it fails. When `pthread_join` fails, **errno** will be set to one of the following values:

EINVAL	The value specified by **tid** does not refer to any existing pthread.
ESRCH	The pthread specified by **tid** exists but has been `pthread_detached`.

CONFORMANCE

`pthread_join` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	`pthread_exit()`, `pthread_detach()`, `pthread_cancel()`
Example -	`pthread_join.c`

NAME

pthread_keycreate – create a thread-specific data key

SYNOPSIS

```
#include <pthread.h>

int pthread_keycreate(key, destructor)
int *key;

void(*destructor)();
```

DESCRIPTION

pthread_keycreate creates a new key for thread-specific data. This key value can be used with pthread_setspecific and pthread_getspecific to maintain data values that are specific to each thread within a process. The new key value is stored in **key**. The **destructor** argument refers to a function that should be called, implicitly by pthread_setspecific, with the value of the previous thread-specific data value for that key. If **destructor** is NULL, no destructor function is registered for the new key value. If **destructor** is not a valid function address, the program will probably get a signal when the bogus destructor is called.

The **destructor** will not be called until a value has been set with pthread_setspecific.

DIAGNOSTICS

pthread_keycreate stores the new key if **key** and returns 0 if it successfully creates a new thread-specific data key. If it fails, pthread_keycreate will return -1. If the call fails, **errno** will contain one of the following:

ENOSPC	No more thread-specific data keys are available.

NOTE

If the destructor function address supplied to `pthread_keycreate` is bogus, then subsequent calls to `pthread_keycreate` may cause the program to receive a signal. The destructor address is not checked for validity; only for NULL or non-NULL.

CONFORMANCE

`pthread_keycreate` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_setspecific()`, `pthread_getspecific()`

NAME

`pthread_kill` – send a signal to a thread

SYNOPSIS

```
#include <signal.h>

int pthread_kill(thread, sig)
pthread_t thread;
int sig;
```

DESCRIPTION

`pthread_kill` is used to direct a signal to be asynchronously delivered to the specified thread whose id is given by **thread**.

DIAGNOSTICS

`pthread_kill` returns `0` when successful, or `-1` upon error. If it fails, no signal shall be sent. If the call fails, **errno** will contain one of the following:

`EINVAL`	The value of the **sig** argument is an invalid or unsupported signal number.
`EINVAL`	The thread ID specified by **thread** is invalid.

CONFORMANCE

`pthread_kill` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

System Calls - `kill()`

Library Functions - `pthread_self()`

NAME

`pthread_mutexattr_create`,
`pthread_mutexattr_delete` – manage mutex attribute structures

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_create(mutexattr)
pthread_mutexattr_t *mutexattr;

int pthread_mutexattr_delete(mutexattr)
pthread_mutexattr_t *mutexattr;
```

DESCRIPTION

A POSIX mutex is created using an attributes structure to define the characteristics of the mutex. This attributes structure must be initialized using `pthread_mutexattr_create`. Any resources that may have been allocated when creating a mutex attributes structure can be de-allocated using `pthread_mutexattr_delete`.

Conforming POSIX.4a applications, for portability reasons, should use `pthread_mutexattr_create` and `pthread_mutexattr_delete` to initialize mutex attributes structures.

DIAGNOSTICS

`pthread_mutexattr_create` and `pthread_mutexattr_delete` always return 0. They do not fail, since, under the current system, they do not do anything. However, if the address passed to `pthread_mutexattr_create` is bad, then the pthread may receive a `SIGSEGV` or a `SIGBUS` signal.

CONFORMANCE

`pthread_mutexattr_create` and `pthread_mutexattr_delete` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_mutex_init()`

NAME

`pthread_mutex_destroy` – delete a user mutex

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_destroy(mut)
pthread_mutex_t *mut;
```

DESCRIPTION

`pthread_mutex_destroy` frees up the system resources for the mutex pointed to by **mut** and removes it from the list of active mutexes.

DIAGNOSTICS

`pthread_mutex_destroy` returns 0 if the mutex was destroyed, or -1 if not. If the call fails **errno** will contain one of the following:

EBUSY	Some thread is waiting on the mutex.
EINVAL	Mutex does not exist.

CONFORMANCE

`pthread_mutex_destroy` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_mutex_trylock()`

NAME

`pthread_mutex_init` – initialize a user mutex

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_init(mut, attr)
pthread_mutex_t *mut;
pthread_mutexattr_t attr;
```

DESCRIPTION

`pthread_mutex_init` initializes the user mutex referred to by **`mut`**. The attributes object **`attr`** is required by POSIX, and should be initialized using `pthread_mutexattr_create`. However, this object is not used by LynxOS. It is present for standards compliance only.

A mutex is a mutual exclusion (hence the name) synchronization variable that can be used by threads to ensure that only one thread accesses a shared resource at a time. Between the time that a thread has returned from `pthread_mutex_lock` and the time it executes `pthread_mutex_unlock`, no other thread will be allowed to successfully complete a `pthread_mutex_lock` call using mutex **`mut`**. Threads that must ensure that they are the only threads accessing a particular shared resource should bracket the code accessing the resource with calls to `pthread_mutex_lock` and `pthread_mutex_unlock`.

Of course, it may well be necessary for a thread to examine some condition associated with the shared resource and possibly block awaiting for some condition to be satisfied before proceeding with an operation on shared data. A condition variable can be used in conjunction with a mutex to allow the thread to atomically release the mutex and block awaiting a particular condition. Mutexes and condition variables are closely related in the way they are used. A mutex can be used without any condition variables, if nothing more than simple mutual exclusion is required. However, a condition variable must be used in conjunction with a mutex. Other uses of condition variables are unsafe and disallowed by the system.

NOTE

Using a mutex that has not been initialized with `pthread_mutex_init`, or which has been destroyed with `pthread_mutex_destroyed`, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`pthread_mutex_init` returns -1 if it fails to get the mutex. If the call fails, **`errno`** will contain one of the following:

`EAGAIN`	No more room for user mutexes.

CONFORMANCE

`pthread_mutex_init` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_mutex_destroy()`, `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_unlock()`, `pthread_mutexattr_create()`

NAME

pthread_mutex_lock,
pthread_mutex_trylock – acquire a mutex lock, blocking or otherwise

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_lock(m)
pthread_mutex_t *m;

int pthread_mutex_trylock(m)
pthread_mutex_t *m;
```

DESCRIPTION

pthread_mutex_lock is used to acquire a mutual exclusion lock on a shared resource guarded by mutex **m**. A thread executing a pthread_mutex_lock or a pthread_mutex_trylock on an "unowned" mutex will be granted ownership of the mutex. A thread executing a pthread_mutex_lock on an "owned" mutex will be blocked until the thread comes to the head of the queue of threads awaiting the mutex and the owner of the mutex releases ownership by executing a pthread_mutex_unlock or a pthread_cond_wait. A thread executing a pthread_mutex_trylock on an owned mutex will not be blocked, but will rather return -1 with **errno** equal to EINTR. Threads waiting for a mutex are serviced in order of priority; highest-priority thread first.

NOTE

Using a mutex that has not been created with pthread_mutex_init, or which has been deleted with pthread_mutex_destroy, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

pthread_mutex_lock and pthread_mutex_trylock return 0 when successful, or -1 upon error. If the call fails, **errno** will contain one of the following:

`EINVAL`	**m** does not refer to a valid mutex.
`EINTR`	The mutex wait was interrupted by a signal delivered to the thread, or (in the case of `pthread_mutex_trylock`) the mutex is locked already.

CONFORMANCE

`pthread_mutex_lock` and `pthread_mutex_trylock` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	`pthread_mutex_unlock()`, `pthread_mutex_init()`, `pthread_mutex_destroy()`, `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`
Example -	`pthread_mutex_lock.c`, `pthread_mutex_trylock.c`

NAME

pthread_mutex_unlock – release a mutex lock

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_unlock(m)
pthread_mutex_t *m;
```

DESCRIPTION

pthread_mutex_unlock is used by the owner of a mutex lock to release that lock. The mutex becomes unowned if and only if there are no threads currently awaiting ownership of the mutex; otherwise, the mutex ownership is granted to the thread that is of equal or greater priority to all other waiting threads and which has been waiting the longest for the mutex.

NOTE

Using a mutex that has not been created with pthread_mutex_create, or which has been deleted with pthread_mutex_delete, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

Pthread_mutex_unlock returns 0 when successful, or -1 upon error. If the call fails **errno** will contain one of the following:

EINVAL	**m** does not refer to a valid mutex.
EPERM	The mutex is not owned by the thread.

CONFORMANCE

pthread_mutex_unlock is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions - `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_create()`, `cv_wait()`, `cv_signal()`, `cv_broadcast()`

NAME

pthread_once – do something once and only once

SYNOPSIS

```
#include <pthread.h>

pthread_once_t onetimeonly = pthread_once_init;

int pthread_once(one_time_only, routine)
pthread_once_t *one_time_only;
void (*routine)();
```

DESCRIPTION

pthread_once provides a means of executing **routine** once and only once. It is useful when it's not quite clear which thread will get to an initialization routine first. So long as **one_time_only** is initialized to pthread_once_init, it is guaranteed that only one call to routine will be generated, no matter how many times pthread_once is called with a given **pthread_once_t**.

DIAGNOSTICS

pthread_once always returns 0, whether or not the routine was actually called.

NOTE

pthread_once can be thought of as a wrapper around the typical initialization code for a module:

```
static int module_initialized = 0;
...
randomfunction()
{
if (! module_initialized++) {
initialize_module();
}
 ...
```

```
}
...
```

The code above is roughly analogous to:

```
pthread_once_t module_initialized = pthread_once_init;
(void) pthread_once(&module_initialized,initialize_module)
```

The difference is that `pthread_once` is airtight, even on a multiprocessor.

CONFORMANCE

`pthread_once` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	`pthread_create()`
Example -	`pthread_once.c`

NAME

pthread_self – get system thread ID

SYNOPSIS

```
#include <pthread.h>

pthread_t pthread_self()
```

DESCRIPTION

pthread_self returns the system thread ID for the calling thread. The thread ID is used as a thread identifier in most thread control system and library calls.

DIAGNOSTICS

pthread_self never fails.

NOTE

pthread_self is a macro accessing a piece of thread-specific data which is initialized to contain the thread ID when the thread is first started up. This value can be accidentally overwritten by an errant program, just as any value in the writable address space of the thread can be. However, the macro is an order of magnitude faster than the system call. To use the system call version of pthread_self, define _POSIX_THREADS_CALLS when compiling C code that calls pthread_self.

CONFORMANCE

pthread_self is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	pthread_create(), pthread_exit(), pthread_detach(), pthread_join()
Example -	pthread_self.c

NAME

pthread_setcancel,
pthread_setasynccancel – set calling system thread's cancelability status

SYNOPSIS

```
#include <pthread.h>

int pthread_setcancel(state)
int state;

int pthread_setasynccancel(state)
int state;
```

DESCRIPTION

pthread_setcancel sets the calling pthread's general cancelability status on if state is CANCEL_ON and off if state is CANCEL_OFF. pthread_setasynccancel sets the calling pthread's asynchronous cancelability status on if state is CANCEL_ON and off if state is CANCEL_OFF.

When the pthread's asynchronous cancelability state is TRUE, and the pthread's general cancelability state is TRUE, the pthread will be immediately canceled by pthread_cancels issued against it. When the pthread's cancelability has asynchronous cancelability set to FALSE, and the pthread's general cancelability state is TRUE, then pthread_cancels issued against the pthread will cause it to be canceled the next time it reaches a cancellation point. A pthread is never canceled so long as it has general cancelability set to FALSE. Instead, the cancel request is held pending until the pthread's cancelability changes so that general cancelability is TRUE; at that time the pthread is canceled either synchronously or asynchronously, depending on the pthread's asynchronous cancelability.

System thread cancellation points occur:

when the pthread issues a pthread_cond_wait.

when the pthread issues a pthread_testcancel.

when the pthread issues a pthread_join.

When the pthread is finally terminated, any cleanup routines on the pthread's cleanup stack will be executed in LIFO order by the pthread. A value of `THREAD_CANCELED` is given as the pthread's exit status to any pthreads performing a `pthread_join` with the canceled pthread.

An pthread's initial cancellation state is `CANCEL_GENERAL`.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

DIAGNOSTICS

`pthread_setcancel` returns the pthread's general cancelability status. `pthread_setasynccancel` returns the pthread's asynchronous cancelability status. Both routines return -1 upon error. In the case where a pthread has a pending cancellation request and `pthread_setcancel` causes the `THREAD_CANCELLABLE` attribute of the thread to transition to TRUE, the pthread will be immediately terminated, so in this case the call will never return.

CONFORMANCE

`pthread_setcancel` and `pthread_setasynccancel` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	`pthread_cond_wait()`, `pthread_join()`, `pthread_cancel()`, `pthread_testcancel()`, `pthread_cleanup_push()`, `pthread_cleanup_pop()`, `pthread_create()`
Example -	`pthread_setcancel.c`, `pthread_setasynccancel.c`

NAME

pthread_setprio – set system thread scheduling priority

SYNOPSIS

```
#include <pthread.h>

int pthread_setprio(tid, prio)
pthread_t tid;
int prio;
```

DESCRIPTION

pthread_setprio sets the scheduling priority of the thread identified by **tid** to **prio**.

DIAGNOSTICS

If **tid** refers to a currently existing thread, pthread_setprio returns its priority. Otherwise, pthread_setprio returns -1 and sets **errno** to one of the following values:

ESRCH	The thread referred to by **tid** does not exist.
EINVAL	**prio** is an invalid value.

CONFORMANCE

pthread_setprio is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	pthread_create(), pthread_attr_setprio(), pthread_getprio()
Example -	pthread_setprio.c

NAME

`pthread_setscheduler`,
`pthread_getscheduler` – set/get POSIX thread scheduling algorithm and priority

SYNOPSIS

```
#include <pthread.h>

int pthread_setscheduler(tid, scheduler, prio)
pthread_t tid;
int scheduler;
int prio;

int pthread_getscheduler(tid)
pthread_t tid;
```

DESCRIPTION

`pthread_setscheduler` sets the scheduling algorithm and priority of the thread **tid** to **scheduler** and **prio**, respectively. `pthread_getscheduler` returns the value for the **scheduler** under which the thread **tid** is currently running.

Valid values for **scheduler** are:

SCHED_FIFO	A first in-first out (FIFO) scheduling policy.
SCHED_RR	A round robin scheduling policy.
SCHED_DEFAULT	A priority-based quantum scheduling policy.
SCHED_OTHER	Identical to SCHED_DEFAULT

DIAGNOSTICS

If **tid** refers to a currently existing thread, `pthread_setscheduler` returns the old scheduling policy of the thread, and `pthread_getscheduler` returns the current scheduling policy. Otherwise, both functions return -1 and set **errno** to one of the following values:

`EINVAL`	**`prio`** is outside the range of priorities for the specified scheduling policy, or `alg` is not a valid scheduling policy.
`EPERM`	**`prio`** is greater than the maximum non-privileged priority, and the caller is non-privileged.
`ESRCH`	**`pid`** does not specify a valid thread.

CONFORMANCE

`pthread_setscheduler` and `pthread_getscheduler` are implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	`pthread_create()`, `pthread_setprio()`, `pthread_getprio()`, `pthread_attr_setscheduler()`
Example -	`pthread_getscheduler.c`

NAME

`pthread_testcancel` – allow the calling pthread to be terminated now

SYNOPSIS

```
#include <pthread.h>

void pthread_testcancel()
```

DESCRIPTION

`pthread_testcancel` allows the calling pthread to be terminated if another pthread has issued a `pthread_cancel` against this system thread and this pthread's general cancelability state is `TRUE` and its asynchronous cancelability state is `FALSE`.

If the calling pthread's general cancelability is set to `FALSE`, then the `pthread_testcancel` does nothing. If the calling pthread's general cancelability is `TRUE` and its asynchronous cancelability is `TRUE`, then it cannot have a pending cancellation request —any call to `pthread_cancel` would have already caused the pthread to terminate. The only time this routine might have an effect is when the calling thread's general cancelability is `TRUE` and asynchronous cancelability is `FALSE`.

When the pthread is canceled, any cleanup routines on the pthread's cleanup stack will be executed in LIFO order by the pthread. A value of `THREAD_CANCELED` is given as the pthread's exit status to any pthreads performing a `pthread_join` with the terminated pthread.

The pthread's initial cancellation state is `CANCEL_GENERAL`.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

DIAGNOSTICS

`pthread_testcancel` never fails. It returns the cancelability state of the system thread, or does not return at all.

CONFORMANCE

`pthread_testcancel` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

Library Functions -	`pthread_join()`, `pthread_cond_wait()`, `pthread_create()` `pthread_cancel()`, `pthread_setcancel()`, `pthread_cleanup_pop()`
Example -	`pthread_testcancel.c`

NAME

`pthread_yield` – force a POSIX thread to relinquish the CPU

SYNOPSIS

`void pthread_yield ()`

DESCRIPTION

`pthread_yield` forces the currently running POSIX thread to relinquish the CPU. It shall resume execution after all other runnable threads (POSIX threads, non-POSIX threads, or processes) of equal or greater priority have been scheduled to run.

DIAGNOSTICS

`pthread_yield` is the same as `yield`, and therefore, is also devoid of return value and personality.

CONFORMANCE

`pthread_yield` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

System Calls - `yield()`

NAME

putc, putchar,
fputc, putw,
unlocked_putc,
unlocked_putchar,
unlocked_fputc,
unlocked_putw – put a character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

int putchar(c)
char c;

int fputc(c, stream)
char c;
FILE *stream;

int putw(w, stream)
int w;
FILE *stream;

int unlocked_putc(c, stream)
char c;
FILE *stream;

int unlocked_putchar(c)
char c;

int unlocked_fputc(c, stream)
char c;
FILE *stream;

int unlocked_putw(w, stream)
int w;
FILE *stream;
```

DESCRIPTION

`putc` writes the character **c** to the output stream at the current pointer position. `putchar` is implemented as a macro, in terms of `putc`, and always writes to `stdout`.

`fputc` is similar to `putc` but it is a true function. It is slower but takes less space per invocation.

`putw` writes the word (**int** or **unsigned**) **w** to the output stream at the current pointer position. `putw` does not force even alignment on the file.

`putc`, `fputc`, `putw`, and `putchar` obtain mutually-exclusive access to the relevant stream, and are therefore thread-safe. The unlocked variants of these functions are identical to their thread-safe brethren but perform no locking. If the application wishes to do its own locking, it can do so by using `flockfile` and `funlockfile`. In that case, the unlocked versions will be somewhat faster.

DIAGNOSTICS

If successful, the value written is returned.

If unsuccessful, `EOF` is returned. This can occur if the file is not open for writing or if the output file cannot be grown.

Because `EOF` is a valid **int**, `ferror` should be used to check for error when using `putw`.

SEE ALSO

Library Functions - `fopen()`, `fclose()`, `flockfile()`, `funlockfile()`, `getc()`, `ungetac()`

NAME

`putenv` – change or add environment data

SYNOPSIS

```
int putenv(string)
char *string;
```

DESCRIPTION

`putenv` adds the string pointed to by **`string`** to the environment. **`string`** should point to a string of the form `name=value`. This has the effect of setting the value of the environment variable `name` to `value`. If the environment variable `name` is already defined, then its value is changed. Otherwise it is created. In either case, the string pointed to by **`string`** becomes part of the environment, so modifying the **`string`** (after `putenv` has returned) will modify the environment. When a new string defining the same environment variable `name` as **`string`** is passed to `putenv`, the space used by the **`string`** is no longer used.

DIAGNOSTICS

`putenv` returns non-zero if it was unable to obtain enough space via `malloc` for an expanded environment, otherwise zero.

CONFORMANCE

System V

SEE ALSO

System Calls - `execve()`

Library Functions - `getenv()`, `malloc()`

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent(p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

putpwent writes a password file entry on the stream **f**, derived from the contents of the **passwd** structure pointed to by **p**. The password file entry produced by putpwent matches the format of /etc/passwd.

DIAGNOSTICS

putpwent returns non-zero if it detects an error, otherwise zero.

CONFORMANCE

System V

SEE ALSO

Library Functions - getpwent(), getpw(), getgrent()

File Format - passwd

NAME

`puts`, `fputs` – put a string on a stream

SYNOPSIS

```
include <stdio.h>

int puts(s)
char *s;

int fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

`puts` writes the null-terminated string pointed to by **s** to the standard output stream `stdout`. The string is followed by a new-line character.

`fputs` writes the null-terminated string, pointed to by **s**, to **stream**. The string is not followed by a new-line character.

Neither function writes out the null character which terminates the argument string.

DIAGNOSTICS

`EOF` is returned if an error occurs. This will happen if output is attempted to a file not open for writing.

SEE ALSO

Library Functions - `ferror()`, `fopen()`, `fread()`, `printf()`, `putc()`, `gets()`

NAME

`qsort` – sort contents of an array

SYNOPSIS

```
qsort(base, nel, width, compar)
char *base;
int nel, width;
int(*compar)();
```

DESCRIPTION

`qsort` is a fast internal sort. The first argument, **`base`**, points to the beginning of the data to be sorted; **`nel`** gives the number of elements, each width bytes in size; **`compar`** points to a function which will be called to compare elements during the sort. **`compar`** is invoked as follows:

```
compar(item1, item2)
itemtype *item1, *item2;
```

`compar` must return a number less than, equal to, or greater than zero, reflecting the relationship between `*item1` and `*item2`.

NOTE

This version of `qsort` is not troubled by sorted runs in the input list.

SEE ALSO

Library Functions - `bsearch()`

NAME

`raise` – send signal to self

SYNOPSIS

```
#include <signal.h>

int raise(sig)
int sig;
```

DESCRIPTION

The `raise` function sends the signal **sig** to the calling thread. It uses the `kill` system call and either the `getpid` or `_getstid` system call to send the signal to the caller:

Single-threaded C library:

```
kill(getpid(), sig);
```

Multi-threaded C library:

```
kill(BUILDPID(0, _getstid()), sig);
```

DIAGNOSTICS

The `raise` function returns `0` if successful, or `-1` if not. If the call fails, **errno** will contain one of the following:

`EINVAL` The signal **sig** is invalid.

SEE ALSO

System Calls - `_getstid(), getpid(), kill(), sigaction()`

NAME

`rand`, `srand` – simple random-number generator

SYNOPSIS

```
#include <stdio.h>

int rand()

void srand(seed)
long seed;
```

DESCRIPTION

`rand` uses a multiplicative congruency random-number generator. It returns values which may span the range of positive integers. **`srand`** can be called at any time to reset the random-number generator to a new starting point. The generator is initially seeded with a value of 1.

SEE ALSO

Library Functions - `random()`

NAME

`random, srandom,`
`initrandom, setstate` – extra-random random numbers

SYNOPSIS

```
long random()

void srandom(seed)
long seed;

char *initstate(seed, state, n)
long seed; long *state;
int n;

char *setstate(state)
long *state;
```

DESCRIPTION

`random` returns pseudo-random numbers uniformly distributed over the range of positive long integers. The period of the random sequence (assuming default initialization; see below) is about sixteen times larger than the size of the range. The sequence can be initialized with **`srandom`** by passing some **`long`** value. Each such initialization value results in the same sequence; the default value if **`srandom`** is not called, is 1.

The values produced by `random` are such that all quotients derived from them are random. In other words, the values can be reduced by some factor or restricted modulo some maximum value and the result is still random. For example, if only the least significant bit of each value is used, a random bit string results.

The algorithm used by `random` is such that the state information of the generator is larger than a single **`seed`** value. The state of the generator can be initialized in such a way as to provide a larger or smaller period than the default. The **`initstate`** routine uses the array state as the generator workspace. The size of the array (in bytes, not words), **`n`**, should be a power of two between 8 and 1024. The larger the array, the longer the period. The seed is used to fill the array with random data for the generator algorithm. **`initstate`** returns the address of the previous state array.

The **setstate** routine sets the state of the generator to **state**, and returns the old state. Thus different sequences can be rapidly interchanged.

NOTE

Unlike `rand`, `random` returns a **long value**. Care should be taken to properly declare `random` in programs which may run in environments in which **int** and **long** variables are different in size.

SEE ALSO

Library Functions - `rand()`

NAME

regcmp, regex – compile and execute regular expression

SYNOPSIS

```
char *regcmp(string1, [, string2, ...], (char *)0)
char *string1, *string2, ...;

char *regex(cmpstr, subjstr)
char *cmpstr;
char *subjstr;

extern char *__loc1;
```

DESCRIPTION

regcmp is the regular expression compiling routine. regcmp compiles a regular expression and returns a pointer to the compiled form. The regular expression is interpreted as the concatenation of the arguments **string1**, **string2**, etc. malloc is used to create space for the compiled form, so it is the user's responsibility to free such space when it is no longer needed.

regex executes a compiled form against a **subject string**. **cmpstr** is a compiled form returned by regcmp and **subjstr** is the **subject string** against which the compiled form is executed. regex returns a pointer to the next unmatched character on success or NULL on failure. The global variable **__loc1** points to where the match began.

NOTE

The user program may run out of memory if regcmp is called iteratively without freeing the memory no longer needed.

SEE ALSO

Library Functions - malloc()

NAME

`remove` – remove a file

SYNOPSIS

```
#include <stdio.h>

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

DESCRIPTION

`remove` removes the file named path from the file system. For directory files, `remove` is equivalent to `rmdir`. For all other file types, `remove` is equivalent to `unlink`. `remove` returns 0 if successful, or -1 if not.

SEE ALSO

System Calls - `rmdir()`, `unlink()`

NAME

`rpc` – library routines for remote procedure calls

SYNOPSIS AND DESCRIPTION

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

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

```
void
auth_destroy(auth)
AUTH *auth;
```

A macro that destroys the authentication information associated with `auth`. Destruction usually involves deallocation of private data structures. The use of `auth` is undefined after calling `authdestroy()`

```
AUTH *
authnone_create()
```

Create and returns an `RPC` authentication handle that passes non-usable authentication information with each remote procedure call. This is the default authentication used by `RPC`.

```
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
char *host;
int uid, gid, len, *aup_gids;
```

Create and return an `RPC` authentication handle that contains authentication information. The parameter `host` is the name of the machine on which the information was created; `uid` is the user's user ID; `gid` is the user's current group ID; `len` and `aup_gids` refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

```
AUTH *
authunix_create_default()
```

Calls **authunix_create()** with the appropriate parameters.

```
callrpc(host, prognum, versnum, procnum, inproc, in,
     outproc, out)
char *host;
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
```

Calls the remote procedure associated with `prognum`, `versnum`, and `procnum` on the machine, `host`. The parameter `in` is the address of the procedure's argument(s), and `out` is the address of where to place the result(s); `inproc` is used to encode the procedure's parameters, and outproc is used to decode the procedure's results. This routine returns 0 if it succeeds, or the value of `enum clnt_stat` cast to an integer if it fails. The routine `clnt_perrno()` is handy for translating failure status values into messages.

WARNING: Calling remote procedures with this routine uses UDP/IP as a transport; see `clntudp_create()` for restrictions. You do not have control of timeouts or authentication using this routine.

```
enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc,
     out, eachresult)
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
resultproc_t eachresult;
```

Like `callrpc()`, except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls `eachresult()`, whose form is:

```
eachresult(out, addr)
char *out;
struct sockaddr_in *addr;
```

Where out is the same as out passed to clnt_broadcast(), except that the remote procedure's output is decoded there; addr points to the address of the machine that sent the results. If eachresult() returns 0, clnt_broadcast() waits for more replies; otherwise it returns with appropriate status.

WARNING: Broadcast sockets are limited in size to the maximum transfer unit of the data link. For ethernet, this value is 1500 bytes.

```
enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
CLIENT *clnt;
u_long procnum;
xdrproc_t inproc, outproc;
char *in, *out;
struct timeval tout;
```

A macro that calls the remote procedure procnum associated with the client handle, clnt, which is obtained with an RPC client creation routine such as clnt_create(). The parameter in is the address of the procedure's argument(s), and out is the address of where to place the result(s); inproc is used to encode the procedure's parameters, and outproc is used to decode the procedure's results; tout is the time allowed for results to come back.

```
clnt_destroy(clnt)
CLIENT *clnt;
```

A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including clnt itself. Use of clnt is undefined after calling clnt_destroy(). If the RPC library opened the associated socket, it will close it also. Otherwise, the socket remains open.

```
CLIENT *
clnt_create(host, prog, vers, proto)
char *host;
u_long prog, vers;
char *proto;
```

Generic client creation routine. `host` identifies the name of the remote host where the server is located. `proto` indicates which kind of transport protocol to use. The currently supported values for this field are `udp` and `tcp`. Default timeouts are set, but can be modified using `clnt_control()`.

WARNING: Using UDP has its shortcomings. Since UDP-based RPC messages can only hold up to 8KB of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

```
bool_t
clnt_control(cl, req, info)
CLIENT *cl;
char *info;
```

A macro used to change or retrieve various information about a client object. `req` indicates the type of operation, and `info` is a pointer to the information. For both UDP and TCP, the supported values of `req` and their argument types and what they do are:

```
CLSET_TIMEOUT   struct timeval     set total timeout
CLGET_TIMEOUT    struct timeval       get total timeout
```

NOTE

If you set the timeout using `clnt_control()`, the timeout parameter passed to `clnt_call()` will be ignored in all future calls.

```
CLGET_SERVER_ADDR  struct sockaddrin  get server's address
```

The following operations are valid for UDP only:

```
CLSET_RETRY_TIMEOUT   struct timeval     set the retry timeout
CLGET_RETRY_TIMEOUT    struct timeval     get the retry timeout
```

The retry `timeout` is the time that UDP RPC waits for the server to reply before retransmitting the request.

```
clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an `RPC` call. The parameter `out` is the address of the results, and `outproc` is the XDR routine describing the results. This routine returns 1 if the results were successfully freed, and 0 otherwise.

```
void
clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;
```

A macro that copies the error structure out of the client handle to the structure at address `errp`.

```
void clnt_pcreateerror(s)
char *s;
```

Print a message to standard error indicating why a client `RPC` handle could not be created. The message is prepended with string `s` and a colon. Used when a `clnt_create()`, `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()` call fails.

```
void
clnt_perrno(stat)
enum clnt_stat stat;
```

Print a message to standard error corresponding to the condition indicated by stat. Used after `callrpc()`.

```
clnt_perror(clnt, s)
CLIENT *clnt;
char *s;
```

Print a message to standard error indicating why an `RPC` call failed; `clnt` is the handle used to do the call. The message is prepended with string `s` and a colon. Used after `clnt_call()`.

```
char *
clnt_spcreateerror(s)
char *s;
```

Like `clnt_pcreateerror()`, except that it returns a string instead of printing to the standard error.

BUGS

Returns pointer to static data that is overwritten on each call.

```
char *
clnt_sperrno(stat)
enum clnt_stat stat;
```

Take the same arguments as `clnt_perrno()`, but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message. The string ends with a NEWLINE.

`clnt_sperrno()` is used instead of `clnt_perrno()` if the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with `printf`, or if a message format different than that supported by `clnt_perrno()` is to be used.

NOTE

Unlike `clnt_sperror()` and `clnt_spcreaterror()`, `clnt_sperrno()` returns pointer to static data, but the result will not get overwritten on each call.

```
char *
clnt_sperror(rpch, s)
CLIENT *rpch;
char *s;
```

Like `clnt_perror()`, except that, like `clnt_sperrno()`, it returns a string instead of printing to standard error.

BUGS

Returns pointer to static data that is overwritten on each call.

```
CLIENT *
clntraw_create(prognum, versnum)
u_long prognum, versnum;
```

This routine creates a toy RPC client for the remote program prognum, version versnum. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see svcraw_create(). This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

```
CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
struct sockaddr_in *addr;
u_long prognum, versnum;
int *sockp;
u_int sendsz, recvsz;
```

This routine creates an RPC client for the remote program prognum, version versnum; the client uses TCP/IP as a transport. The remote program is located at Internet address *addr. If addr->sin_port is 0, then it is set to the actual port that the remote program is listening on (the remote portmap service is consulted for this information). The parameter sockp is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets sockp. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters sendsz and recvsz; values of 0 choose suitable defaults. This routine returns NULL if it fails.

```
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum, versnum;
struct timeval wait;
int *sockp;
```

This routine creates an RPC client for the remote program prognum, version versnum; the client uses use UDP/IP as a transport. The remote program is located at Internet address addr. If addr->sin_port is 0, then it is set to actual port that the remote program is listening on (the remote portmap service is consulted for this information). The parameter sockp is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets sockp. The UDP transport resends the call message in intervals of wait time until a response is received or until the call times out. The total time for the call to time out is specified by clnt_call().

WARNING

Since UDP-based RPC messages can only hold up to 8KB of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

```
CLIENT *
clntudp_bufcreate(addr, prognum, versnum, wait, sockp,
sendsize, recosize)
struct sockaddr_in *addr;
u_long prognum, versnum;
struct timeval wait;
int *sockp;
unsigned int sendsize;
unsigned int recosize;
```

This routine creates an RPC client for the remote program prognum, on versnum; the client uses use UDP/IP as a transport. The remote program is located at Internet address addr. If addr->sin_port is 0, then it is set to actual port that the remote program is listening on (the remote portmap service is consulted for this information). The parameter sockp is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets sockp. The UDP transport resends the call message in intervals of wait time until a response is received or until the call times out.

The total time for the call to time out is specified by clnt_call(). This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

```
void
get_myaddress(addr)
struct sockaddr_in *addr;
```

Stuff the machine’s IP address into *addr, without consulting the library routines that deal with /etc/ hosts. The port number is always set to htons(PMAPPORT).

```
struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;
```

A user interface to the `portmap` service, which returns a list of the current `RPC` program-to-port mappings on the host located at IP address `*addr`. This routine can return NULL. The command `rpcinfo -p` uses this routine.

```
u_short
pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
u_long prognum, versnum, protocol;
```

A user interface to the portmap service, which returns the port number on which waits a service that supports program number `prognum`, version `versnum`, and speaks the transport protocol associated with `protocol`. The value of `protocol` is most likely `IPPROTO_UDP` or `IPPROTO_TCP`. A return value of 0 means that the mapping does not exist or that the `RPC` system failed to contact the remote `portmap` service. In the latter case, the global variable `rpc_createerr` contains the `RPC` status.

```
enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in, outproc,
out, tout, portp)
struct sockaddr_in *addr;
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
struct timeval tout;
u_long *portp;
```

A user interface to the `portmap` service, which instructs `portmap` on the host at IP address `*addr` to make an `RPC` call on your behalf to a procedure on that host. The parameter `*portp` will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in `callrpc()` and `clnt_call()`. This procedure should be used for a “ping” and nothing else. See also `clnt_broadcast()`.

```
pmap_set(prognum, versnum, protocol, port)
u_long prognum, versnum, protocol;
u_short port;
```

A user interface to the portmap service, which establishes a mapping between the triple [prognum, versnum, protocol] and port on the machine's portmap service. The value of protocol is most likely IPPROTO_UDP or IPPROTO_TCP. This routine returns 1 if it succeeds, 0 otherwise. Automatically done by svc_register().

```
pmap_unset(prognum, versnum)
u_long prognum, versnum;
```

A user interface to the portmap service, which destroys all mapping between the triple [prognum, versnum, *] and ports on the machine's portmap service. This routine returns 1 if it succeeds, 0 otherwise.

```
registerrpc(prognum, versnum, procnum, procname, inproc,
outproc)
u_long prognum, versnum, procnum;
char *(*procname) ();
xdrproc_t inproc, outproc;
```

Register procedure procname with the RPC service package. If a request arrives for program prognum, version versnum, and procedure procnum, procname is called with a pointer to its parameter(s); progname should return a pointer to its static result(s); inproc is used to decode the parameters while outproc is used to encode the results. This routine returns 0 if the registration succeeded, -1 otherwise.

NOTE: Remote procedures registered in this form are accessed using the UDP/IP transport; see svcudp_create() for restrictions.

```
struct rpc_createerr rpc_createerr;
```

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine clnt_pcreateerror() to print the reason why.

```
svc_destroy(xprt)
SVCXPRT xprt*
```

A macro that destroys the RPC service transport handle, xprt. Destruction usually involves deallocation of private data structures, including xprt itself. Use of xprt is undefined after calling this routine.

fd_set svc_fdset;

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the `select` system call. This is only of interest if a service implementor does not call `svc_run()`, but rather does his own asynchronous event processing. This variable is read-only (do not pass its address to `select`!), yet it may change after calls to `svc_getreqset()` or any creation routines.

int svc_fds;

Similar to `svc_fdset()`, but limited to 32 descriptors. This interface is obsoleted by `svc_fdset()`.

```
svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using `svc_getargs()`. This routine returns 1 if the results were successfully freed, and 0 otherwise.

```
svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, `xprt`. The parameter `in` is the address where the arguments will be placed; `inproc` is the XDR routine used to decode the arguments. This routine returns 1 if decoding succeeds, and 0 otherwise.

```
struct sockaddr_in *
svc_getcaller(xprt)
SVCXPRT *xprt;
```

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, `xprt`.

```
svc_getreqset(rdfds)
fd_set *rdfds;
```

This routine is only of interest if a service implementor does not call `svc_run()`, but instead implements custom asynchronous event processing. It is called when the select system call has determined that an `RPC` request has arrived on some `RPC` socket(s); `rdfds` is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of `rdfds` have been serviced.

```
svc_getreq(rdfds)
int rdfds;
```

Similar to `svc_getreqset()`, but limited to 32 descriptors. This interface is obsoleted by `svc_getreqset()`.

```
svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
u_long prognum, versnum;
void (*dispatch) ();
u_long protocol;
```

Associates `prognum` and `versnum` with the service dispatch procedure, `dispatch`. If `protocol` is 0, the service is not registered with the `portmap` service. If `protocol` is non-zero, then a mapping of the triple [`prognum`, `versnum`, `protocol`] to `xprt->xp_port` is established with the local portmap service (generally protocol is 0, `IPPROTO_UDP` or `IPPROTO_TCP`). The procedure `dispatch` has the following form:

```
dispatch(request, xprt)
struct svc_req *request;
SVCXPRT *xprt;
```

The `svc_register()` routine returns 1 if it succeeds, and 0 otherwise.

```
svc_run()
```

This routine never returns. It waits for `RPC` requests to arrive, and calls the appropriate service procedure using `svc_getreq()` when one arrives. This procedure is usually waiting for a `select()` system call to return.

```
svc_sendreply(xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;
```

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter `xprt` is the request's associated transport handle; `outproc` is the XDR routine which is used to encode the results; and out is the address of the results. This routine returns 1 if it succeeds, 0 otherwise.

```
void
svc_unregister(prognum, versnum)
u_long prognum, versnum;
```

Remove all mapping of the double [`prognum`, `versnum`] to dispatch routines, and of the triple [`prognum`, `versnum`, `*`] to port number.

```
void
svcerr_auth(xprt, why)
SVCXPRT *xprt;
enum auth_stat why;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

```
void
svcerr_decode(xprt)
SVCXPRT *xprt;
```

Called by a service dispatch routine that cannot successfully decode its parameters. See also `svc_getargs()`.

```
void
svcerr_noproc(xprt)
SVCXPRT *xprt;
```

Called by a service dispatch routine that does not implement the procedure number that the caller requests.

```
void
svcerr_noprog(xprt)
SVCXPRT *xprt;
```

Called when the desired program is not registered with the `RPC` package. Service implementors usually do not need this routine.

```
void
svcerr_progvers(xprt)
SVCXPRT *xprt;
```

Called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

```
void
svcerr_systemerr(xprt)
SVCXPRT *xprt;
```

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

```
void
svcerr_weakauth(xprt)
SVCXPRT *xprt;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient authentication parameters. The routine calls `svcerr_auth(xprt, AUTH_TOOWEAK)`.

```
SVCXPRT *
svcraw_create()
```

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see `clntraw_create()`. This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

```
SVCXPRT *
svctcp_create(sock, send_buf_size,  recv_buf_size)
int sock;
u_int send_buf_size, recv_buf_size;
```

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket `sock`, which may be `RPC_ANYSOCK`, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, xprt->xp_sock is the transport's socket descriptor, and xprt->xp_port is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers; values of 0 choose suitable defaults.

```
SVCXPRT *
svcfd_create(fd, sendsize, recvsize)
int fd;
u_int sendsize;
u_int recvsize;
```

Create a service on top of any open descriptor. Typically, this descriptor is a connected socket for a stream protocol such as TCP. sendsize and recvsize indicate sizes for the send and receive buffers. If they are 0, a reasonable default is chosen.

```
SVCXPRT *
svcudp_bufcreate(sock, sendsize, recosize)
int sock;
```

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket sock, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, xprt->xp_sock is the transport's socket descriptor, and xprt->xp_port is the transport's port number. This routine returns NULL if it fails. This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

```
xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;
```

Used for encoding RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

```
xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;
```

Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

```
void
xdr_callhdr(xdrs, chdr)
XDR *xdrs;
struct rpc_msg  *chdr;
```

Used for describing RPC call header messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

```
xdr_callmsg(xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;
```

Used for describing RPC call messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

```
xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;
```

Used for describing RPC authentication information messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

```
xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;
```

Used for describing parameters to various `portmap` procedures, externally. This routine is useful for users who wish to generate these parameters without using the `pmap` interface.

```
xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;
```

Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these parameters without using the pmap interface.

```
xdr_rejected_reply(xdrs, rr)
XDR *xdrs;
struct rejected_reply *rr;
```

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

```
xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC style messages without using the RPC package.

```
void
xprt_register(xprt)
SVCXPRT *xprt;
```

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable svc_fds(). Service implementors usually do not need this routine.

```
void
xprt_unregister(xprt)
SVCXPRT *xprt;
```

Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable svc_fds(). Service implementors usually do not need this routine.

SEE ALSO

Library Functions – `xdr()`

Documentation - *Remote Procedure Calls: Protocol Specification*

Remote Procedure Call Programming Guide

rpcgen Programming Guide

RPC: Remote Procedure Call Protocol Specification; RFC1050; Sun Microsystems, Inc.; USC-ISI.

NAME

`rtcreate` – create a realtime file

SYNOPSIS

```
#include &lt;types.h&gt; #include &lt;files.h&gt;

int rtcreate(path,mode,reqattr,reqwhich,actattr,actwhich)
char *path ;
mode_t mode ;
struct attrbuf *reqattr ;
int reqwhich ;
struct attrbuf *actattr ;
int *actwhich ;
```

DESCRIPTION

The `rtcreate()` function creates the realtime file named in path. The permissions for the new file are set to the value obtained by the bitwise-AND of **mode** and the complement of the current `umask`. The **reqattr** and **actattr** arguments point to the **attrbuf** structure.

The actual attributes of the created file will be stored in the **attrbuf** structure specified by the **actattr** argument upon successful completion of the `rtcreate()` function.

The **reqwhich** argument indicates which fields of the **reqattr** structure are to be used. The flags are defined in <files.h>

ATB_ALLOC	Preallocation amount.
ATB_ALLOCFLAGS	Preallocation modification
ATB_EXTINCR	Extension size.
ATB_EXTFLAGS	Extension modification.
ATB_CACHEFLAGS	Caching Flags.
ATB_DIOFLAGS	Direct I/O flags.
ATB_DIOSIZE	Direct I/O size.

`ATB_ALL` All of the above.

The **actwhich** argument will contain the set of **attrbuf** fields in **actattr** that do not match those in **reqattr**. In other words **reqattr** is what was requested and **actattr** is what the system actually set up. If a flag is set in the **actwhich** argument then the corresponding element of the **actattr** structure does not match the corresponding element of the **reqattr** structure.

The `rtcreate()` function returns a file descriptor to the calling process if the function is successful; otherwise the `rtcreate()` function returns either `a` or the value -1 and sets **errno** to indicate the error.

The **attrbuf** structure, defined in **<files.h>**, contains at least the following fields:

```
off_t    atb_alloc;
int      atb_allocflags;
off_t    atb_extincr;
int      atb_extflags;
int      atb_cacheflags;
int      atb_dioflags;
unsigned atb_diosize;
```

The `atb_alloc` defines the length in bytes of the spaces to be allocated for future use as part of that file. The allocation has the following characteristics

NOTE

The **atb_alloc** field in **reqattr** is the length in bytes of the space reserved for use the file. It should be a multiple of 512. The reservation has the following characteristics:

`ATB_NOEXTEND` The file cannot be extended when the current allocation is exceeded.

`ATB_CONTIGUOUS`
Allocation is performed in a manner consistent with the performance attributes of a contiguous file.

DIAGNOSTICS

If any of the following conditions occur, the `rtcreate()` function returns -1 and sets **`errno`** to the corresponding value:

`EACCES`	Search permission is denied for a component of the path prefix, or the file exists and write permission is denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
`EEXIST`	The specified file already exists.
`EINTR`	The `rtcreate()` operation was interrupted by a signal.
`EINVAL`	The `rtcreate()` function is not supported on this file system, or one or more of the requested attributes are not valid. Upon return, if one or more of the attributes are not valid, **`actwhich`** will indicate at least one of the parameters found to be in invalid.
`EISDIR`	The named file is a directory.
`EMFILE`	Too many file descriptors are in use by this process.
`ENFILE`	Too many files are currently open in the system.
`ENOENT`	The named file does not exist or the **`path`** argument points to an empty string.
`ENOSPC`	The filesystem that would contain the file has no space.
`ENOTDIR`	A component of the path prefix is not a directory.
`EPERM`	The process does not have appropriate privilege.
`EROFS`	The named file resides on a read-only file system.

CONFORMANCE

`rtcreate` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Library Functions - `creat()`

Example - `rtcreate.c`

NAME

`rtfsync` – synchronous file synchronization

SYNOPSIS

```
#include <fcntl.h>

int rtfsync(fildes,op)
int fildes ;
int op ;
```

DESCRIPTION

The `rtfsync()` function causes all modified data associated with the file indicated by the file descriptor **fildes** to be successfully transferred. If **op** is `O_DSYNC`, all I/O operations are completed to maintain data integrity. If **op** is `O_FSYNC`, all I/O operations are completed to maintain file integrity.

The `rtfsync()` function returns the value zero to the calling process if the function is successful; otherwise, the function returns the value -1.

DIAGNOSTICS

If any of the following conditions occur, the `rtfsync()` function returns -1 and sets **errno** to the corresponding value:

EBADF	The **fildes** argument is not a valid file descriptor open for writing.
EINVAL	A value of **op** other than `O_DSYNC` or `O_FSYNC` was specified.

CONFORMANCE

`rtfsync` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`fcntl()`, `open()`
Library Functions -	`afsynch()`
Example -	`rtfsync.c`

NAME

scandir, alphasort – convenient directory examination

SYNOPSIS

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

scandir(dirname, namelist, select, compar)
char *dirname;
struct direct **namelist;
int(*select)();
int(*compar)();

alphasort(d1, d2)
struct direct **d1, **d2;
```

DESCRIPTION

scandir provides an easy way of reading the contents of a LynxOS directory file. All or part of the directory named by **dirname** is read into an array, automatically allocated with malloc. The address of the array is placed in **namelist**. scandir returns the number of entries in the array.

The **select** function is called once for each directory entry. A pointer to the direct structure is passed to **select**. If **select** returns zero, scandir will skip that entry. Usually, **select** is used to include only files whose names match some pattern. If **select** is NULL, every entry will be included.

The **compar** function is passed directly to qsort to sort the completed array. **alphasort** is a function which compares the names of two directory entries, and can be used when the list of entries is to be sorted alphabetically. If **compar** is NULL, the array is not sorted.

The memory allocated for the array can be released by successively freeing each entry (with free), and then freeing the array itself.

DIAGNOSTICS

scandir returns -1 if the directory cannot be opened or if `malloc` refuses to allocate memory.

SEE ALSO

Library Functions - `directory, malloc(), qsort()`

File Formats - `dir`

NAME

`scanf`, `fscanf`, `sscanf` – convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf(format [, pointer] ... )
char *format;

int fscanf(stream, format [, pointer] ... )
FILE *stream;
char *format;

int sscanf(s, format [, pointer] ... )
char *s, *format;
```

DESCRIPTION

Each function reads characters from somewhere (a file or a string), converts them according to a format, and stores the results in its arguments. The arguments consist of the control string **`format`** and a set of pointer arguments indicating where the converted input should be stored.

`scanf` reads from standard input, `fscanf` from **`stream`**, and `sscanf` from the string **`s`**. `scanf` and `fscanf` obtain mutually-exclusive access to the relevant **`stream`**, and are therefore thread-safe. See `flockfile` and `funlockfile` for more information on the file locking mechanism.

The control string may contain:

- White-space characters (blanks, tabs, and new-lines) which cause input to be read up to the next non white-space character.
- An ordinary character (not `%`), which must match the next character of the input stream.
- Conversion specifications, consisting of the character `%`, an optional assignment suppressing character `*`, an optional numerical maximum field width, an optional l indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by the flag `*` (see below). An input field is defined as a string of non-white-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion code indicates the interpretation of the input field. For a suppressed field, no pointer argument should be given. The following conversion codes are legal:

`%`	A single `%` is expected in the input at this point; no assignment is done.
`d,D`	A decimal integer is expected. The corresponding argument should be a pointer to `int`.
`u`	same as **`d`** above but applies to `unsigned int` values.
`U`	same as **`d`** above but applies to `unsigned long` values.
`i`	same as **`d`** above but applies to an `int` constant as in standard C.
`I`	same as **`d`** above but applies to a `long` constant as in standard C.
`h`	A short decimal integer is expected. The corresponding argument should be a pointer to `short`.
`o,O`	An octal integer is expected. The corresponding argument should be a pointer to **`int`**.
`x,X`	A hexadecimal integer is expected. The corresponding argument should be a pointer to **`int`**.
`e,f,g`	A floating point number is expected. The next field is converted accordingly and stored through the corresponding argument, which should be a pointer to `float`. The input format for floating point numbers is an optionally signed string of digits, possibly with a decimal point, followed by an optional exponent field consisting of an `e`, or an `E` followed by an optionally signed integer.

E, F, G	same as **e**, **f**, **g** options above but they are for signed `double` precision numbers.
s	A character string is expected. The corresponding argument should be a pointer to **char** giving the address of an array of characters large enough to accept the string and a terminating `\0`, which will be added automatically. The input field is terminated by a white-space character.
c	A character is expected. The corresponding argument should be a pointer to **char**. The normal skip over white space is suppressed in this case; to read the next non-space character, use `1s`. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.
n	Does not match anything. Returns the current number of characters read.
[str]	A character string consisting of characters in the set `str` is expected. If the "^" first character of `str` is a string consisting of characters not in the rest of `str` is expected. The corresponding argument should be a pointer to a character array.

The conversion characters **d**, **o**, and **x** may be preceded by `l` to indicate that a pointer to `long` rather than to **int** is in the argument list. Also, the conversion characters **e**, **f**, and **g** may be preceded by `l` to indicate that a pointer to `double` rather than to `float` is in the argument list.

`scanf` conversion terminates at `EOF`, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

`scanf` returns the number of successfully matched and assigned input items. This number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, `EOF` is returned.

DIAGNOSTICS

These functions return `EOF` on end of input and a short count for missing or illegal data items.

NOTE

Trailing white space (including a new-line) is left unread unless matched in the control string.

SEE ALSO

Library Functions - `fopen()`, `printf()`, `getc()`, `flockfile()`, `funlockfile()`

NAME

`sched_getparam` – get scheduling parameters

SYNOPSIS

```
#include <sched.h>

int sched_getparam(pid, param)
pid_t pid;
struct sched_param *param;
```

DESCRIPTION

`sched_getparam` retrieves the scheduling parameters of the process specified by **pid**, in the structure referenced by the **param** argument. The structure `sched_param` is defined as follows:

```
int    sched_priority   # Scheduling priority
```

If **pid** is zero, scheduling parameters for the calling process are returned.

DIAGNOSTICS

`sched_getparam` retrieves the scheduling parameters of the specified process into the structure pointed to by the **param** argument when successful, and returns `0`. Upon failure, it returns `-1` and sets **errno** as follows:

`ESRCH`	The argument **pid** does not correspond to a valid process.

CONFORMANCE

The `sched_getparam` function is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `sched_getscheduler()`, `sched_setscheduler()`, `sched_setparam()`

NAME

sched_get_priority_max,
sched_get_priority_min,
sched_rr_get_interval – get scheduling parameter limits

SYNOPSIS

```
#include <sched.h>

int sched_get_priority_max(policy)
int policy;

int sched_get_priority_min(policy)
int policy;

int sched_rr_get_interval(pid, interval)
pid_t pid;
struct timespec *interval;
```

DESCRIPTION

sched_get_priority_max and sched_get_priority_min return the maximum and minimum scheduling priorities, respectively, for the scheduling policy **policy**. The valid scheduling policies are:

SCHED_FIFO	First in-first out scheduling policy
SCHED_RR	Round robin scheduling policy
SCHED_DEFAULT	Priority based quantum scheduling policy (Lynx proprietary)
SCHED_OTHER	Same as SCHED_DEFAULT

sched_rr_get_interval sets the **timespec** structure referred by the **interval** argument to the current time quantum for the process **pid**, with the SCHED_RR scheduling policy. If the **pid** argument is zero, the time quantum for the calling process is retrieved.

DIAGNOSTICS

The `sched_get_priority_max` and `sched_get_priority_min` functions return the appropriate maximum and minimum values, respectively, when successful. Upon failure, these functions return `-1`, and set **`errno`** to indicate the error condition.

The `sched_rr_get_interval` function returns `0` when successful. On failure it returns `-1`, and sets the **`errno`**.

The **`errno`** set upon failure for these functions are as follows:

`EINVAL`	The **`policy`** argument is not a valid scheduling policy.
`ESRCH`	The argument **`pid`** to `sched_rr_get_interval` does not correspond to a valid process.

CONFORMANCE

The functions `sched_get_priority_max`, `sched_get_priority_min`, and `sched_rr_get_interval` are implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `sched_getparam()`, `sched_setparam()`, `sched_getscheduler()`, `sched_setscheduler()`

NAME

`sched_getscheduler` – get scheduling policy

SYNOPSIS

```
#include <sched.h>

int sched_getscheduler(pid)
pid_t pid;
```

DESCRIPTION

`sched_getscheduler` returns the scheduling policy of the process specified by the **pid** argument. If **pid** is zero, the scheduling policy for the calling process is returned. The scheduling policy may be one of the following valid scheduling policies:

`SCHED_FIFO`	First in-first out scheduling policy
`SCHED_RR`	Round robin scheduling policy
`SCHED_DEFAULT`	Priority based quantum scheduling policy (Lynx proprietary)
`SCHED_OTHER`	Same as `SCHED_DEFAULT`

See the `sched_get_priority_max` manual page for more information.

DIAGNOSTICS

`sched_getscheduler` returns the scheduling policy of the specified process when successful. On failure, it returns `-1`, and sets **errno** as follows:

`ESRCH`	The argument **pid** does not correspond to a valid process.

CONFORMANCE

The `sched_getscheduler` function is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `sched_setscheduler()`, `sched_getparam()`, `sched_setparam()`

NAME

sched_setparam – set scheduling parameters

SYNOPSIS

```
#include <sched.h>

int sched_setparam(pid, param)
pid_t pid;
const struct sched_param *param;
```

DESCRIPTION

sched_setparam sets the scheduling parameters of the process specified by the **pid** argument to the structure referred by the **param** argument. The structure sched_param contains the following member:

```
int     sched_priority   # Scheduling priority
```

The value of the sched_priority member must lie within the minimum and maximum ranges for the scheduling policy of the specified process.

If **pid** is zero, scheduling parameters for the calling process are set.

DIAGNOSTICS

sched_setparam sets the scheduling parameters of the specified process when successful, and returns 0. Upon failure, it returns −1 and sets **errno** as follows:

EINVAL	The scheduling parameters specified by the **param** argument are invalid.
EPERM	Permission to set the scheduling parameters of the requested process is denied.
ESRCH	The argument **pid** does not correspond to a valid process.

CONFORMANCE

The `sched_setparam` function is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `sched_getscheduler()`, `sched_setscheduler()`, `sched_getparam()`

NAME

sched_setscheduler – set scheduling policy and parameters

SYNOPSIS

```
#include <sched.h>

int sched_setscheduler(pid, policy, param)
pid_t pid;
int policy;
const struct sched_param *param;
```

DESCRIPTION

sched_setscheduler sets the scheduling policy for the process specified by **pid** to **policy**, and the scheduling parameters to the structure referenced by **param**. If **pid** is zero, the scheduling policy and scheduling parameters for the calling process are set.

The argument **policy** can be one of the following:

SCHED_FIFO	First in-first out scheduling policy
SCHED_RR	Round robin scheduling policy
SCHED_DEFAULT	Priority based quantum scheduling policy (Lynx proprietary)
SCHED_OTHER	Same as SCHED_DEFAULT

The structure sched_param is used to set the scheduling priority of the process specified by **pid**. It contains the following member:

```
int    sched_priority   # Scheduling priority
```

DIAGNOSTICS

sched_setscheduler sets the scheduling policy of the specified process when successful and returns the previous scheduling policy. On failure, it returns -1, and sets **errno** as follows:

EINVAL	Either the **policy** argument or the scheduling parameters specified by the **param** argument are invalid.
EPERM	Permission to set the scheduling policy and/or the scheduling parameters of the requested process is denied.
ESRCH	The argument **pid** does not correspond to a valid process.

CONFORMANCE

The `sched_setscheduler` function is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `sched_getscheduler()`, `sched_getparam()`, `sched_setparam()`

NAME

`sched_yield` – yield processor

SYNOPSIS

```
#include <sched.h>
int sched_yield(void)
```

DESCRIPTION

`sched_yield` causes the calling process to give up the processor until it again becomes the head of its process list. It takes no arguments.

DIAGNOSTICS

`sched_yield` always completes successfully and returns `0`. It sets no **errno** values.

CONFORMANCE

The `sched_yield` function is implemented according to the POSIX 1003.1b (Real-Time) standard.

NAME

sem_close – close a named semaphore

SYNOPSIS

```
#include <semaphore.h

int sem_close(sem)
sem_t *sem;
```

DESCRIPTION

sem_close closes the named semaphore referenced by **sem**. Any system resources allocated to the process by the system for this semaphore are deallocated. POSIX.1b semaphores are persistent. That is, if the semaphore has not been removed by a call to sem_unlink, then the sem_close call has no effect on the state of the semaphore.

DIAGNOSTICS

sem_close closes a named semaphore when successful, and returns 0. Otherwise, it returns -1, and sets **errno** to one of the following values:

EINVAL	The **sem** argument is not a valid semaphore descriptor.

CONFORMANCE

sem_close is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls -	close()
Library Calls -	sem_open(), sem_wait(), sem_trywait(), sem_post()

NAME

sem_destroy – destroy an unnamed semaphore

SYNOPSIS

```
#include <semaphore.h>

int sem_destroy(sem)
sem_t *sem;
```

DESCRIPTION

sem_destroy destroys an unnamed POSIX.1b semaphore referenced by **sem**. Only a semaphore which was created with the sem_init function may be destroyed with sem_destroy.

DIAGNOSTICS

sem_init destroys the unnamed semaphore and returns 0 if successful. Otherwise, it returns -1, and sets **errno** to one of the following values:

EINVAL	The **sem** argument is not a valid semaphore.

CONFORMANCE

sem_destroy is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - sem_init()

NAME

sem_getvalue – get the value of a semaphore

SYNOPSIS

```
#include <semaphore.h>

int sem_getvalue(sem, sval)
sem_t *sem;
int *sval;
```

DESCRIPTION

sem_getvalue obtains the value of the semaphore referred to by **sem**, and stores it in the location referenced by **sval**. The value in **sval** is a value that occurred some time during the sem_getvalue call, but not necessarily when the call returns.

When **sem** is locked, the value stored in **sval** by sem_getvalue is either zero or a negative number whose absolute value is the number of processes waiting for the semaphore sometime during the call.

DIAGNOSTICS

sem_getvalue obtains the value of a semaphore when successful, and returns 0. Otherwise, it returns -1, and sets **errno** to one of the following values:

EINVAL	The **sem** argument does not refer to a valid semaphore.

CONFORMANCE

sem_getvalue is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - sem_wait(), sem_trywait(), sem_post()

NAME

sem_init – initialize an unnamed semaphore

SYNOPSIS

```
#include <semaphore.h>

int sem_init(sem, pshared, value)
sem_t *sem;
int pshared;
unsigned int value;
```

DESCRIPTION

sem_init initializes an unnamed POSIX.1b semaphore referenced by **sem**. The value of the semaphore is initialized to **value**. The maximum possible value of a semaphore is given by SEM_VALUE_MAX, which is defined in <limits.h>. The semaphore initialized by this call may be used in subsequent calls to sem_post, sem_wait, and sem_trywait. It remains usable until it is destroyed by sem_destroy. The **pshared** argument is ignored by LynxOS.

For POSIX.1b compliance, only **sem** by itself may be used for performing synchronization. Two processes accessing an unnamed semaphore must have it mapped in shared memory.

DIAGNOSTICS

sem_init initializes the unnamed semaphore and returns 0 if successful. Otherwise, it returns -1, and sets **errno** to one of the following values:

EINVAL	The value argument exceeds SEM_VALUE_MAX.
ENOSPC	A resource required to initialize the semaphore has been exhausted. The limit on the number of semaphores in the system (SEM_NSEMS_MAX defined in <limits.h>) has been reached.
EPERM	The process lacks the privileges to initialize the semaphore.

CONFORMANCE

`sem_init` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `sem_destroy(), sem_post(), sem_wait()`

NAME

sem_open – open/initialize a named semaphore

SYNOPSIS

```
#include <semaphore.h>

sem_t *sem_open(name, oflags [,mode, value])
const char *name;
int oflags;
mode_t mode;
unsigned int value;
```

DESCRIPTION

sem_open initializes and opens a named semaphore referenced by the character string **name**, depending on the argument **oflags**. Following a call to sem_open with semaphore name **name**, the process can reference the semaphore using the address returned from the call.

The semaphore can be used in subsequent calls to sem_wait, sem_trywait, and sem_post functions. It remains usable by this process until it is successfully closed by calls to sem_close, exit, or exec.

The **oflags** argument determines whether the semaphore is initialized or merely opened. The following bits may be set in **oflags**:

O_CREAT — If the named semaphore does not exist, create it. If the semaphore exists, there is no effect except as noted under O_EXCL below. This flag requires two additional arguments: **mode**, and **value**. The semaphore is created with an initial value of **value**. A valid semaphore value must be less than or equal to SEM_VALUE_MAX. The semaphore's permission bits are set to the value of the **mode** argument, except those set in the file mode creation mask of the process.

O_EXCL — If O_EXCL and O_CREAT are both set, sem_open fails if the semaphore already exists.

DIAGNOSTICS

sem_open opens a named semaphore when successful, and returns the address of the semaphore. Otherwise, it returns -1, and sets **errno** to one of the following values:

EACCES	There is no valid permission to create or open the named semaphore.
EEXIST	O_CREAT and O_EXCL are set and the named semaphore already exists.
EINTR	The sem_open operation was interrupted by a signal.
EINVAL	Semaphore creation was attempted, and **value** was greater than SEM_VALUE_MAX.
EMFILE	Too many semaphore descriptors are currently in use by this process.
ENAMETOOLONG	The length of the **name** string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.
ENFILE	Too many semaphores are currently open in the system.
ENOENT	O_CREAT is not set and the named semaphore does not exist.
ENOSPC	There is insufficient space for the creation of the newly-named semaphore.

CONFORMANCE

sem_open is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls -	open()
Library Calls -	sem_close(), sem_wait(), sem_trywait(), sem_post()

NAME

sem_post – unlock a semaphore

SYNOPSIS

```
#include <semaphore.h>

int sem_post(sem)
sem_t *sem;
```

DESCRIPTION

sem_post unlocks the semaphore referenced by **sem**.

If no processes are blocked waiting for the semaphore to become unlocked, the semaphore value is simply incremented. The resulting semaphore value is positive.

If one or more processes are blocked waiting for the semaphore, then one of the blocked processes is allowed to lock the semaphore. The value of the semaphore resulting from sem_post under these circumstances is zero. The highest priority blocked process, which had been waiting the longest for the semaphore would be allowed to lock the semaphore.

The sem_post function can be invoked from a signal handler.

DIAGNOSTICS

sem_post unlocks the semaphore when successful, and returns 0. Otherwise, it returns -1, and sets **errno** to one of the following values:

EINVAL	The **sem** argument does not refer to a valid semaphore.

CONFORMANCE

sem_post is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - sem_wait(), sem_trywait()

NAME

`sem_unlink` – remove a named semaphore

SYNOPSIS

```
#include <semaphore.h>

int sem_unlink(name)
const char *name;
```

DESCRIPTION

`sem_unlink` removes the named semaphore named by the string **`name`**. If there are one or more references to the named semaphore when the object is unlinked, only the name is removed when `sem_unlink` returns. The named semaphore is destroyed only when all references to it are closed.

The **`name`** argument must conform to standard pathname construction rules, even though it does not appear in the file system.

DIAGNOSTICS

`sem_unlink` removes the named semaphore when successful, and returns `0`. Otherwise it returns `-1`, and sets **`errno`** to one of the following values:

`EACCES`	There is no permission to unlink the named semaphore.
`ENAMETOOLONG`	The length of the named semaphore is too long, or the length of one of the pathname components is too long.
`ENOENT`	The named semaphore does not exist.

CONFORMANCE

`sem_unlink` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls -	`unlink()`
Library Calls -	`sem_open()`, `sem_close()`

NAME

`sem_wait, sem_trywait` – lock a semaphore

SYNOPSIS

```
#include <semaphore.h>

int sem_wait(sem)
sem_t*sem;

int sem_trywait(sem)
sem_t*sem;
```

DESCRIPTION

`sem_wait` locks the semaphore referenced by **sem**. If the semaphore is currently locked by a process, `sem_wait` blocks until it either locks the semaphore or is interrupted by a signal.

`sem_trywait` locks the semaphore referenced by **sem** only if it is unlocked. If the semaphore is locked, `sem_trywait` does not block and does not lock the semaphore.

DIAGNOSTICS

`sem_wait` and `sem_trywait` functions are successful if the calling process locked the semaphore **sem**, and return `0`. Otherwise, they return `-1`, and set **errno** to one of the following values:

EAGAIN	The semaphore was already locked, and could not be immediately locked by `sem_trywait`.
EINVAL	The **sem** argument does not refer to a valid semaphore.
EINTR	A signal interrupted this function.

CONFORMANCE

`sem_wait` and `sem_trywait` are implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `sem_post()`

NAME

`setattr,fsetattr` – set attributes a realtime file

SYNOPSIS

```
#include <types.h>
#include <files.h>

int setattr(path, reqattr, reqwhich, actattr, actwhich)
char *path;
struct attrbuf *reqattr;
int reqwhich;
struct attrbuf *actattr;
int *actwhich;

int fsetattr(fildes, reqattr, reqwhich, actattr, actwhich)
int fildes;
struct attrbuf *reqattr;
int reqwhich;
struct attrbuf *actattr;
int *actwhich;
```

DESCRIPTION

The `setattr()` function sets one or more attributes associated with the realtime file specified by **`path`**.

The `fsetattr()` function sets one or more attributes associated with the realtime file specified by the file descriptor **`fildes`**.

The **`reqwhich`** arguments specifies which attributes to set. This argument is in the same format as the **`reqwhich`** argument from `rtcreate()`.

The **`reqattr`** specifies the desired attributes.

The resulting attributes for the file will be stored in **`actattr`** upon successful completion of the function. The **`actwhich`** argument will contain the set of **`attrbuf`** fields in **`actattr`** that do not match those in **`reqattr`**.

The above functions return the value zero to the calling process if the new attributes were successfully set. The value -1 is returned if at least one of the requested operation(s) cannot be performed.

DIAGNOSTICS

If any of the following conditions occur, the `setattr()` function returns -1 and sets **errno** to the corresponding value:

EACCES	Search permission is denied for a component of the **path** prefix.
EINVAL	One of the requested attributes is not valid or the set attributes function is not supported for the specified file. Upon return, **actwhich** will indicate which parameter was found to be in error.
ENOENT	The named file does not exist or the **path** argument points to an empty string.
ENOTDIR	A component of the **path** prefix is not a directory.
EPERM	The process does not have appropriate privilege.

If any of the following conditions occur, the `fsetattr()` function returns -1 and sets **errno** to the corresponding value:

EBADF	The **fildes** argument is not a valid file descriptor.
EINVAL	One of the requested attributes is not valid or the set attributes function is not supported for the specified file. Upon return, **actwhich** will indicate which parameter was found to be in error.

CONFORMANCE

`setattr` and `fsetattr` are implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Library Functions - `rtcreate(), getattr()`

Example - `setattr.c`

NAME

setbuf, setvbuf,
unlocked_setbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf(stream, buf)
FILE *stream;
char *buf;

int setvbuf(stream, buf, mode, size)
FILE *stream;
char *buf;
int mode;
size_t size;

int unlocked_setbuf(stream, buf)
FILE *stream;
char *buf;
```

DESCRIPTION

A standard I/O stream may be unbuffered, block buffered, or line buffered. When an output stream is unbuffered, data appears at the destination file or terminal as soon as it is written. When it is block buffered, several characters are saved up and written as a block. When it is line buffered, characters are saved up until a newline is encountered, or input is read from `stdin`. The `fflush` function may be used to force buffered data out early. A buffer is obtained using `malloc` upon the first `getc` or `putc` on the file. If `stdout` refers to a terminal, it is line buffered. The `stderr` channel is unbuffered by default.

The `setbuf` function is used after a stream has been opened but before it is read or written. It causes the character array pointed to by **`buf`**, with size equal to **`size`**, to be used instead of an automatically allocated buffer. If **`buf`** is a NULL pointer, input/output will be completely unbuffered.

There is no limitation on the size of a buffer, but the constant `BUFSIZ`, defined in the `<stdio.h>` header file, is usually a good buffer size:

```
char buf[BUFSIZ];
```

The `setvbuf` function is used after a stream has been opened but before it is read or written. The **`mode`** argument determines how **`stream`** will be buffered, as follows:

`_IOFBF`	causes input/output to be block (i.e. fully) buffered.
`_IOLBF`	causes input/output to be line buffered.
`_IONBF`	causes input/output to be unbuffered.

If **`buf`** is not a NULL pointer, the array to which is points will be used for buffering, instead of an automatically allocated buffer. The **`size`** argument specifies the size of the array.

The `setbuf` function is safe for use by multiple threads at the same time, because it performs implicit locking of the input stream. Such locking requires a certain amount of overhead. If you use `flockfile` and `funlockfile` to manually lock and unlock I/O streams, or if you know that there will be no access by concurrent threads to the input stream, then you can use `unlocked_setbuf`, which does not perform any locking and which is therefore incrementally faster than the thread-safe `setbuf`.

NOTE

It is up to the user to free the space used by **`buf`** (if it was dynamically allocated) before closing the stream.

SEE ALSO

Library Functions - `fclose()`, `flockfile()`, `funlockfile()`, `fopen()`, `fread()`, `getc()`, `malloc()`, `printf()`, `putc()`, `puts()`

NAME

`setbuffer, setlinebuf` – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuffer(stream, buf, size)
FILE *stream;
char *buf;
int size;

int setlinebuf(stream)
FILE *stream;
```

DESCRIPTION

A standard I/O stream may be unbuffered, block buffered, or line buffered. When an output stream is unbuffered, data appears at the destination file or terminal as soon as it is written. When it is block buffered, several characters are saved up and written as a block. When it is line buffered, characters are saved up until a newline is encountered, or input is read from `stdin`. The `fflush` function may be used to force buffered data out early. A buffer is obtained using `malloc` upon the first `getc` or `putc` on the file. If `stdout` refers to a terminal, it is line buffered. The `stderr` channel is unbuffered by default.

The `setbuffer` function is used after a stream has been opened but before it is read or written. It causes the character array pointed to by **`buf`**, with size equal to **`size`**, to be used instead of an automatically allocated buffer. If **`buf`** is a NULL pointer, input/output will be completely unbuffered. There is no limitation on the size of a buffer, but the constant `BUFSIZ`, defined in the `<stdio.h>` header file, is usually a good buffer size:

```
char buf[BUFSIZ];
```

The `setlinebuf` function is used to change the buffering on a stream from block buffered or unbuffered to line buffered. Unlike `setbuffer`, it can be used at any time that stream is active.

A stream can be changed from unbuffered or line buffered to block buffered using `freopen`. A stream can be changed from block buffered or line buffered to unbuffered by using `freopen` followed by `setbuf` with a buffer argument of NULL.

NOTE

Both `setbuffer` and `setlinebuf` are safe for use by multiple threads at the same time, because they perform implicit locking of the stream via `flockfile` and `funlockfile`.

SEE ALSO

Library Functions - `fclose()`, `flockfile()`, `funlockfile()`, `fopen()`, `fread()`, `getc()`, `malloc()`, `printf()`, `putc()`, `puts()`, `setbuf()`

NAME

`setjmp`, `longjmp`,
`_setjmp`, `_longjmp` – non-local goto and primitive versions

SYNOPSIS

```
#include <setjmp.h>

int setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
int val;

int _setjmp(env)
jmp_buf env;

_longjmp(env, val)
jmp_buf env;
int val;
```

DESCRIPTION

`setjmp` saves its stack environment in **env** (whose type, `jmp_buf`, is defined in **setjmp.h**), for later use by `longjmp`. It returns the value 0.

`longjmp` restores the environment saved by the last call of `setjmp` with the corresponding **env** argument. After `longjmp` is completed program execution continues as if the corresponding call to `setjmp` had just returned the value **val**. The routine from which `setjmp` was called must not have returned in the interim; in other words, the invocation of the routine calling `longjmp` must be traceable to a function call within the routine which called `setjmp`.

`longjmp` cannot cause `setjmp` to return the value 0. If `longjmp` is invoked with a second argument of 0, `setjmp` will return 1. All accessible data have values as of the time `setjmp` was called.

`_setjmp` and `_longjmp` are primitive versions of `setjmp` and `longjmp`. `setjmp` and `longjmp` maintain the supervisor stack of the process in case you jump out of a signal handler; `_setjmp` and `_longjmp` do not. Do not use `_longjmp` to leave a signal handler. You may exhaust your supervisor stack, and then the system will not deliver any more signals to you.

NOTE

Unpredictable (probably disastrous) things will happen if `longjmp` is called when **`env`** contains garbage, or the function which called `setjmp` has since returned.

`setjmp` and `longjmp` are defined by ANSI C: ANSI X3.159 - 1989.

SEE ALSO

System Calls -	`sigsetmask(), sigvec()`
Library Functions -	`signal(), sigsetjmp(), siglongjmp(), evtsetjmp(), evtlongjmp()`

NAME

`setlocale` – extensions to `setlocale` functions

SYNOPSIS

```
#include <locale.h>

char *setlocale(category locale)
int category;
char *locale;
```

DESCRIPTION

The `setlocale()` function sets, changes, or queries the locale of the process according to the values of the **`category`** and **`locale`** arguments. The possible values for **`category`** include:

- `LC_CTYPE`
- `LC_COLLATE`
- `LC_TIME`
- `LC_NUMERIC`
- `LC_MONARY`
- Implementation-defined additional categories.

Environment variables are defined that correspond to the named categories above and that have the same spelling.

The value `LC_ALL` for **`category`** names all of the categories of the locale of the process; `LC_ALL` is a special constant, not a category. There is an environment variable `LC_ALL` with the semantics noted below.

The **`locale`** argument is a pointer to a character string that can be an explicit string, a NULL pointer, or a null string.

When **`locale`** is an explicit string, it can be any of the strings “C”, “C;C;C;C;C” or “POSIX”. The value “C” for **`locale`** specifies the minimal environment for C-language translation. If `setlocale()` is not invoked, the “C” locale shall be the locale of the process.

When **locale** is a NULL pointer, the locale of the process is queried according to the value of **category**. The content of the string returned is unspecified.

When **locale** is a null string, the `setlocale()` function takes the name of the new locale for the specified category from the environment as determined by the first condition met below:

1. If `LC_ALL` is defined in the environment and is not null, the value of `LC_ALL` is used.
2. If there is a variable defined in the environment with the same name as the category and that is not null, the value specified by that environment variable is used.
3. If `LANG` is defined in the environment and is not null, the value of `LANG` is used.

If the resulting value is a supported locale, `setlocale()` sets the specified category of the locale of the process to that value and returns the value specified below. If the value does not name a supported locale (and is not null), `setlocale()` returns a NULL pointer, and the local of the process is not changed by this function call. If no nonnull environment variable is present to supply a value, `setlocale()` sets the specified category of the locale of the process to “C”. The possible actual values of the environment variables are as follows: “C”, “C;C;C;C;C” or “POSIX”.

Setting all of the categories of the locale of the process is similar to successively setting each individual category of the locale of the process, except that all error checking is done before any actions are performed. To set all the categories of the locale of the process, `setlocale()` is invoked as:

```
setlocale (LC_ALL, ""):
```

In this case, `setlocale()` first verifies that the values of all the environment variables it needs according to the precedence above indicate supported locales. If the value of any of these environment-variable searches yields a locale that is not supported (and nonnull), the `setlocale()` function returns a NULL pointer and the locale of the process is not changed. If all environment variables name supported locales, `setlocale()` then proceeds as if it had been called for each category, using the appropriate value from the associated environment variable or from the implementation-defined default if there is no such value.

DIAGNOSTICS

A successful call to `setlocale()` returns a string that corresponds to the locale set. The string returned is such that "a subsequent call with that string and its associated category will restore that part of the process' locale" (C Standard {2}). The string returned shall not be modified by the process, but may be overwritten by a subsequent call to the `setlocale()` function. This string is not required to be the value of the environment variable used, if one was used.

COPYRIGHT NOTICE

SEE ALSO

Library Functions - `localeconv()`

NAME

setprio – set process priority

SYNOPSIS

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

int setprio(pid, prio)
pid_t pid;
int prio;
```

DESCRIPTION

setprio sets the priority of the process specified by **pid** to **prio**. The value of **prio** shall be any integer in the range PRIO_MIN and PRIO_MAX as defined for the current scheduling policy. If **pid** is 0, the priority shall be set for the calling process.

For a process to have permission to set the priority of a process, it must either have root privileges, or the real or effective user ID of the calling process must match the real or effective user ID of the target process.

The target process shall resume execution after all other runnable processes of equal or greater priority have been scheduled to run. The priority of a process is inherited across fork and exec.

If the priority of the target process is set higher than that of the calling process, and if the target process is ready to run, the target process will preempt the calling process. If the calling process sets its priority lower than that of the target process, and if the target process is ready to run, the target process shall preempt the calling process.

DIAGNOSTICS

setprio returns old priority value if successful, or -1 if not.

If setprio fails, **errno** will contain one of the following:

EINVAL	**prio** is outside the range of priorities for the current scheduling algorithm.

EPERM	The effective or real `uids` for process **pid** do not match the effective or real `uids` for the calling process, and the calling process is not privileged.
EPERM	**prio** is greater than the maximum non-privileged priority, and the caller is non-privileged.
ESRCH	**pid** does not specify a valid process.

CONFORMANCE

`setprio` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`getpid(), getprio(), getscheduler(), setscheduler()`
Example -	`setprio.c`

NAME

`setuid`, `setgid`,
`seteuid`, `setegid`,
`setruid`, `setrgid` – change user or group ID

SYNOPSIS

```
setuid(uid)
int uid;

setgid(gid)
int gid;

setruid(uid)
int uid;

setrgid(gid)
int gid;

seteuid(uid)
int uid;

setegid(gid)
int gid;
```

DESCRIPTION

The routines `setuid` and `setgid` set the real and effective user and group ID to the given value. `seteuid` and `setegid` set only the effective user or group ID, and `setruid` and `setrgid` set only the real user or group ID.

These calls are only available to the super user or to processes whose real or effective ID is the same as the argument.

DIAGNOSTICS

If permission is denied, the routines return -1; otherwise they return 0.

SEE ALSO

Utility Programs -	`who`
System calls -	`setreuid()`, `getuid()`, `geteuid()`, `getgid()`, `getegid()`, `getgroups()`
Library Functions -	`getpass()`, `getlogin()`

NAME

`signal` – simplified signal handling

SYNOPSIS

```
#include <signal.h>

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

DESCRIPTION

`signal` provides a simple interface to the system call `sigvec`. The function referenced by **`func`** is installed as the handler for signal number **`sig`**. The address of the former signal handler is returned.

Two special signal handler names are provided in **`signal.h`**: `SIG_DFL` and `SIG_IGN`. If `SIG_DFL` is passed as **`func`**, the default action for the signal is restored. If `SIG_IGN` is passed, the signal is ignored.

The following is a list of LynxOS signals:

```
NSIG        32        /* number of possible signals */
SIGHUP      1         /* hangup */
SIGINT      2         /* interrupt */
SIGQUIT     3         /* quit */
SIGILL      4         /* illegal instruction */
SIGTRAP     5         /* trace trap */
SIGBRK      6         /* break */
SIGIOT      SIGBRK
SIGABRT     SIGBRK
SIGCORE     7         /* kill with core dump */
SIGEMT      SIGCORE
SIGFPE      8         /* floating point exception */
SIGKILL     9         /* cannot be caught or blocked */
SIGBUS      10        /* bus error */
SIGSEGV     11        /* segment violation */
SIGSYS      12        /* bad argument to system call */
SIGPIPE     13        /* write on a pipe with no one to
                         read it */
SIGALRM     14        /* real-time alarm clock */
SIGTERM     15        /* software termination */
```

```
SIGURG          16          /* urgent condition on socket */
SIGSTOP         17          /* stop the task */
SIGTSTP         18          /* stop the task from a tty */
SIGCONT         19          /* signal to continue a stopped
                               process */
SIGCHLD         20          /* child status changed */
SIGCLD          SIGCHLD
SIGTTIN         21          /* background read attempted from
                               term */
SIGTTOU         22          /* background write attempted from
                               term */
SIGIO23                     /* I/O possible on descriptor */
SIGUDEF24       24          /* user defined */
SIGXCPU         SIGUDEF24
SIGUDEF25       25          /* user defined */
SIGXFSZ         SIGUDEF25
SIGVTALRM       26          /* virtual time alarm */
SIGUSR2         SIGVTALARM
SIGPROF         27          /* profiling alarm */
SIGUDEF28       28          /* user defined */
SIGWINCH        SIGUDEF28
SIGUDEF29       29          /* user defined */
SIGADJ          SIGUDEF29
SIGUDEF30       30          /* user defined */
SIGLOST         SIGUDEF30
SIGUDEF31       31          /* user defined */
SIGUSR1         SIGUDEF31
SIGPRIO         32         /* sent to a process when its process
                              group or its priority is changed.
                              default is to inform its open
                              devices with FIOPRIO ioctl */
```

NOTE

The signals `SIGKILL` and `SIGSTOP` cannot be blocked, ignored, or caught; the signal `SIGCONT` cannot be blocked or ignored; and the signal `SIGPRIO` cannot be blocked or caught.

SEE ALSO

System Calls - `sigvec()`, `sigblock()`, `setsetmask()`, `sigpause()`

Library Functions - `psignal()`

NAME

`sigsetjmp`,
`siglongjmp` – non-local goto with signal mask save and restore

SYNOPSIS

```
#include<setjmp.h>

int sigsetjmp(env, savesigmask)
sigjmp_buf env;
int savesigmask;

int siglongjmp(env, returnval)
sigjmp_buf env;
int returnval;
```

DESCRIPTION

`sigsetjmp` and `siglongjmp` do what `setjmp` and `longjmp` do. That is, `sigsetjmp` is used to record a snapshot of the machine environment at a particular point in time, and `siglongjmp` can then be used, later, to return to that exact machine state. `sigsetjmp` and `siglongjmp` are useful for quickly unwinding long sequences of computations that have gone horribly awry.

`sigsetjmp` and `siglongjmp` do more than `setjmp` and `longjmp`. In particular, if you invoke `sigsetjmp` with the second argument not zero, then the signal mask of the process is recorded in the environment buffer as part of the machine snapshot. Then, when `siglongjmp` is called, the signal mask recorded in the buffer is re-installed as part of the process of rewinding back to where the process was at the time `sigsetjmp` was called.

When `sigsetjmp` is directly called, it returns `0`. When `siglongjmp` is called, it never returns, but rather, execution proceeds as if the corresponding `sigsetjmp` call had just returned. In this case, the `sigsetjmp` call seems to return the value that was passed to `siglongjmp` as the **`returnval`** parameter.

If **`returnval`** is `0`, then `sigsetjmp` will seem to return `1`. It is impossible to fake out `sigsetjmp` so that a return via `siglongjmp` returns `0`.

NOTE

If you call `siglongjmp` with an uninitialized environment snapshot, or one for which the stack no longer exists (i.e., not a call frame that still exists on the stack), then unpredictable (but predictably bad) results will ensue.

`sigsetjmp` and `siglongjmp` are defined by POSIX.1: ISO 9945-1 (1990).

SEE ALSO

System Calls -	`sigblock(), sigsetmask()`
Library Functions -	`signal(), setjmp(), longjmp(),` `evtsetjmp(), evtlongjmp()`

NAME

`sigsetops: sigemptyset, sigfillset, sigaddset, sigdelset, sigismember` – signal set operations signals, sets of signal set manipulation

SYNOPSIS

```
#include <signal.h>
sigemptyset(set)
sigset_t *set;

sigfillset(set)
sigset_t *set;

sigaddset(set, sig)
sigset_t *set;
int sig;

sigdelset(set, sig)
sigset_t *set;
int sig;

sigismember(set, sig)
sigset_t *set;
int sig;
```

DESCRIPTION

These functions manipulate **set**s of signals located in the address space of the calling process. They do not affect any of the signal **set**s managed by the operating system, such as the **set** of blocked signals for the process or the **set** of pending signals for the process. `sigemptyset` initializes the signal **set** pointed to by **set** such that all signals are excluded. `sigfillset` initializes the signal **set** pointed to by **set** such that all signals are included. `sigaddset` adds the individual signal **sig** to the signal **set** pointed to by **set**. `sigdelset` deletes the individual signal **sig** from the signal **set** pointed to by **set**. **sigismember** tests whether the signal **sig** is a member of the signal set pointed to by **set**.

DIAGNOSTICS

sigemptyset and sigfillset return 0 if successful. sigaddset and sigdelset return 0 if successful, or -1 if not. sigismember returns 1 if the specified signal is a member of the specified **set**, 0 if it is not, or -1 if an error is detected. If sigaddset, sigdelset, or sigismember detect an error, **errno** will contain one of the following:

EINVAL	The **sig** argument is not a valid signal number.

SEE ALSO

System Calls -	sigaction(), sigpending(), sigprocmask(), sigsuspend()

NAME

sin, cos, tan, asin,
acos, atan, atan2 – trigonometric functions and their inverses

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double tan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

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

DESCRIPTION

`sin`, `cos`, and `tan` each take an argument in radians and return the respective trigonometric function. The caller should ensure that the argument is within the appropriate bounds in order to get meaningful results.

`asin` returns the arc sin of the argument in the range $-\pi/2$ to $\pi/2$.

`acos` returns the arc cosine of the argument in the range 0 to π.

`atan` returns the arc tangent of the argument in the range $-\pi/2$ to $\pi/2$.

`atan2` returns the arc tangent of **x**/**y** in the range $-\pi$ to π.

DIAGNOSTICS

For arguments greater than 1, `asin` and `acos` return the value 0 and set **`errno`** to `EDOM`.

NAME

`sinh, cosh, tanh` – hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

`sinh`, `cosh`, and `tanh` each take a real argument and return the respective hyperbolic function.

SEE ALSO

Library Functions - `exp()`

NAME

sleep, usleep – suspend execution

SYNOPSIS

```
sleep(seconds)
unsigned seconds;

#include <types.h>

usleep(useconds)
time_t useconds;

susleep(seconds, useconds)
time_t seconds;
time_t useconds;
```

DESCRIPTION

sleep suspends process execution for the given number of seconds by setting the real-time interval timer and then waiting for SIGALRM. The actual duration of the suspension may be different from **seconds** by up to one seconds, due to inherent inaccuracies in the system timing mechanisms and to other system activity.

The previous state of the interval timer is restored after the sleep call. If the previous timer would have fired during the slept-through interval, sleep sets the timer to one second in the future in order to give a user SIGALRM handler access to the alarm.

The **usleep** function allows a finer resolution. The **useconds** argument gives the number of microseconds to sleep.

The **susleep** call is provided for completeness. Two values, seconds and microseconds, are passed instead of just seconds or just microseconds; the routine is thus ideal for those occasions that call for a sleep of 1 hour plus 0.01 seconds.

NOTE

The resolution of the timer is never finer than that of the system tick clock, usually something between 1/60 and 1/128 seconds. A non-zero sleep request smaller than one clock period is rounded up.

SEE ALSO

System Calls - `setitimer()`, `sigpause()`, `nanosleep()`

Library Functions - `signal()`

NAME

`spipe` – create a STREAMS pipe

SYNOPSIS

```
spipe(fd)
int fd[2];
```

DESCRIPTION

`spipe()` function creates a STREAMS pipe which is a full-duplex connection between two file descriptors. The STREAMS pipe is created by opening the STREAMS pipe device `/dev/spx` twice and issuing a `I_FDINSERT` ioctl call to establish the full-duplex connection between those two file descriptors.

RETURN VALUE

If succesful, returns `0` and `fd[0]` and `fd[1]` contain the two file descriptors for the pipe. Returns `-1` if unsuccessful.

FILES

`/dev/spx`

SEE ALSO

Device Facilities - `spx`

NAME

`sputl, sgetl` – write/read long integer in machine independent fashion

SYNOPSIS

```
void sputl(value, buffer)
long value;
char *buffer;

long sgetl(buffer)
char *buffer;
```

DESCRIPTION

`sputl` places the four bytes of the long integer **`value`** in memory starting at the address pointed to by **`buffer`**. The ordering of bytes is the same across all machines.

`sgetl` returns the long integer value of the four bytes in memory starting at the address pointed to by **`buffer`**. The returned long integer value is in the byte ordering of the host machine.

The use of `sputl` and `sgetl` provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

NAME

`ssignal`, `gsignal` – software signals

SYNOPSIS

```
#include <signal.h>

int(*ssignal (sig, action))()
int sig,(*action)();

int gsignal(sig)
int sig;
```

DESCRIPTION

`ssignal` and `gsignal` implement a software facility similar to the library function `signal()`. Software signals are associated with integers in the inclusive range 1 through 16. `ssignal` associates a function, action, with the software signal **sig**. `gsignal` generates a software signal **sig** and thus causes the action associated with that signal to be taken.

The argument action to `ssignal` is either the name of a user defined action function or one of the constants `SIG_DFL` (default) or `SIG_IGN` (ignore). `ssignal` returns the action previously associated with **sig**. If no action has been associated with **sig** or **sig** is illegal, `ssignal` returns `SIG_DFL`.

`gsignal` generates the signal identified by its argument, **sig**:

- If an action function has been associated with **sig**, then that action is reset to `SIG_DFL` and the action function entered with argument **sig**. `gsignal` returns the value returned to it by the **action** function.
- If the action for **sig** is `SIG_IGN`, `gsignal` returns the value 1 and takes no other action.
- If the action for **sig** is `SIG_DFL`, `gsignal` returns the value 0 and takes no other action.
- If **sig** has an illegal value or no action was ever associated with **sig**, `gsignal` returns the value 0 and takes no other action.

SEE ALSO

System Calls - `sigaction()`

Library Functions - `signal()`

NAME

st_cancel – cancel a system thread

SYNOPSIS

```
#include <time.h>

#include <st.h>

int stcancel(tid)
tid_t tid;
```

DESCRIPTION

st_cancel cancels or arranges the cancellation of the st whose ID is given by **tid**. It is not possible to cancel a st in another process.

If the st's cancelability state contains CANCEL_ASYNC and CANCEL_GENERAL, the st will be immediately canceled. If the system thread's cancelability state contains CANCEL_GENERAL but not CANCEL_ASYNC, then the st will be canceled the next time it reaches a cancellation point. If the st's cancelability state does not contain CANCEL_GENERAL, then the cancel request is held pending until the st's cancelability changes.

System thread cancellation points occur:

- when the st issues a cv_wait.
- when the st issues a testcancel.
- when the st issues a st_join.

When the st is canceled, any cleanup routines on the st's cleanup stack will be executed in LIFO order by the st. A value of THREAD_CANCELED is given as the st's exit status to any sts performing a st_join with the canceled st.

An st's initial cancellation state is CANCEL_GENERAL.

NOTE

An st that has called st_exit or already been successfully canceled, but which has not been detached with st_detach, can still be successfully canceled with st_cancel. Such a call to st_cancel will have no effect; but it will also not fail.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, SIGTHREADKILL. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle SIGTHREADKILL.

DIAGNOSTICS

st_cancel 0 if the target thread exists, or -1 upon error. If an error is detected, **errno** will be set to one of the following values:

ESRCH	The value given by **tid** does not refer to any known st.

SEE ALSO

System Calls -	st_exit(), st_build(), st_join(), st_detach(), cv_wait()
Library functions -	st_pushrecover(), st_poprecover(), st_testcancel(), st_setcancel(), st_new()

NAME

`st_default_attr` – initialize a `st_attr_t` structure to default values

SYNOPSIS

```
#include <time.h>

#include <st.h>

void st_default_attr(attr)
st_attr_t *attr;
```

DESCRIPTION

`st_default_attr` initializes a **`st_attr_t`** structure at address **`attr`** to system default values.

System Thread Creation Attributes

The **`attr`** argument refers to a structure of type **`st_attr_t`**, which contains the following members:

`stksize`	minimum `st` stack size in bytes
`prio`	`st` scheduling priority
`inheritsched`	inherit scheduling parameters from creating `st`.
`state`	initial `st` schedulability state

`stksize` determines the `st`'s stack size in bytes. This number is a minimum; the stack of the `st` will probably be rounded up to an integral number of pages.

`prio` sets the `st`'s scheduling priority, if the `st` is not inheriting its scheduling parameters from the creating thread. `prio` must be a number between 0 and 255.

state determines whether the st is immediately made runnable or not. state can have one of the following values:

THREAD_RUNNING	The st is created ready and willing to run and will begin execution as soon as it can, according to its scheduling algorithm.
THREAD_STOPPED	The st is created in the stopped state and will not be made runnable until it is started explicitly by another st using st_start.

inheritsched determines whether the st created with these attributes will inherit its scheduling algorithm and priority from its parent or not. If inheritsched is nonzero, then any st created with these attributes will be initialized with the scheduling priority and policy of the st executing st_new. Otherwise, the new st will have its scheduling policy and priority initialized according to the priority and scheduler attributes.

The default values for these attributes is:

stksize	THREAD_DEFAULT_STACK
priority	0
inherit	1
state	THREAD_RUNNING

NOTE

st_default_attr may be implemented as a macro, so do not perform operations on the attr argument within the call. Specifically, avoid calls like the following:

```
st_attr_t bunch[10];   /* A bunch of attribute structures */
st_attr_t *t;
t = bunch;
for (i=0; i<10; i++)
  st_default_attr(t++);
```

Incrementing will cause problems if encountered in the scope of a macro.

DIAGNOSTICS

`st_default_attr` doesn't return anything and doesn't fail. If the address passed to `st_default_attr` is bad, then the `st` may receive a `SIGSEGV` or a `SIGBUS` signal.

SEE ALSO

Library Functions - `st_new()`

NAME

`st_getspecific`,
`st_setspecific` – retrieve, set a thread- specific data value

SYNOPSIS

```
#include <st.h>
#include <sem.h>

int st_setspecific(key, value)
int key;
void *value;

int st_getspecific(key, value)
int key;
void **value;
```

DESCRIPTION

`st_getspecific` retrieves the data value associated with **key** for the calling thread, storing the value in **value**. `st_setspecific` sets the data value associated with **key** to **value**. Different threads have different values associated with a given **key**, allowing a limited form of private thread storage.

In the case of `st_setspecific`, if this thread has previously set a value for **key**, and the destructor for this **key** is not NULL, then the destructor function will be called with the old value before the new value is saved. The destructor will not be called until a value has been set with `st_setspecific`.

DIAGNOSTICS

`st_getspecific` stores the data value in **value** and returns 0 if it successfully retrieved the data value associated with **key**. `st_setspecific` returns 0 if it successfully stored the new value associated with **key**. If either call fails, it will return -1.

If the call fails, **errno** will contain one of the following:

`EINVAL`	Invalid key value. The key was not created using `st_keycreate`.

NOTE

If the destructor function address supplied to `st_keycreate` is bogus, then subsequent calls to `st_keycreate` may cause the program to receive a signal. The destructor address is not checked for validity; only for NULL or non-NULL.

SEE ALSO

Library Functions - `st_keycreate()`

NAME

`st_keycreate` – create a thread-specific data key

SYNOPSIS

```
#include <time.h>

#include <st.h>

#include <sem.h>

int st_keycreate(destructor, key)
void(*destructor)();
int *key;
```

DESCRIPTION

`st_keycreate` creates a new key for thread-specific data. This key value can be used with `st_setspecific` and `st_getspecific` to maintain data values that are specific to each thread within a process. The new key value is stored in **`key`**. The **`destructor`** argument refers to a function that should be called, implicitly by `st_setspecific`, with the value of the previous thread-specific data value for that key. If **`destructor`** is NULL, no destructor function is registered for the new key value. If **`destructor`** is not a valid function address, the program will probably get a signal when the bogus destructor is called.

The destructor will not be called until a value has been set with `st_setspecific`.

DIAGNOSTICS

`st_keycreate` stores the new key if **`key`** and returns 0 if it successfully creates a new `thread-specific` data key. If it fails, `st_keycreate` will return -1. If the call fails, **`errno`** will contain one of the following:

`ENOSPC`	No more thread-specific data keys are available.

NOTE

If the destructor function address supplied to `st_keycreate` is bogus, then subsequent calls to `st_keycreate` may cause the program to receive a signal. The destructor address is not checked for validity; only for NULL or non-NULL.

SEE ALSO

Library Functions - `st_setspecific()`, `st_getspecific()`

NAME

`st_new` – create a new system thread

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_new(start_routine, arg, attr, tidp)
void *(*start_routine)();
void *arg;
st_attr_t *attr;
tid_t *tidp;
```

DESCRIPTION

`stone` creates a new system **st** in the current process context. The **st** is initialized to begin running the routine at **start_routine**, which is passed the single parameter **arg**. The various creation characteristics of the newly- created **st** are determined by the contents of **attr**. Upon successful **st** creation, the ID of the new system thread is stored in **tidp**.

System Thread Creation Attributes

The **attr** argument refers to a structure of type **st_attr_t**, which contains the following members:

`stksize`	minimum **st** stack size in bytes
`prio`	**st** scheduling priority
`inheritsched`	Whether to inherit scheduling parameters from creating thread or not.
`state`	initial **st** schedulability state

`stksize` determines the **st**'s stack size in bytes. This number is a minimum; the stack of the **st** will probably be rounded up to an integral number of pages.

prio sets the **st**'s scheduling priority if the **st** is not inheriting its scheduling state from the creating thread. prio must be a number between 0 and 255.

inheritsched determines whether the **st**'s scheduling state (priority) is inherited from the creating thread, or set according to the prio field.

state determines whether the **st** is immediately made runnable or not. state can have one of the following values:

THREAD_RUNNING	The **st** is created ready and willing to run and will begin execution as soon as it can, according to its scheduling algorithm.
THREAD_STOPPED	The **st** is created in the stopped state and will not be made runnable until it is started explicitly by another **st** using **st_start**.

attr can be initialized to default values using **st_default_attr**.

The **st'**s initial cancellation state is CANCEL_GENERAL.

DIAGNOSTICS

st_new returns 0 when successful, or -1 upon error. If the call fails, **errno** will contain one of the following:

EINVAL	The stack size given is bogus.
EINVAL	The priority value is not a valid priority.
EINVAL	The state value is not THREAD_STOPPED or THREAD_RUNNABLE.
EFAULT	**start_routine** does not point to a location in the process's address space.
ENOMEM	Out of room for **sts**.

SEE ALSO

System Calls -	`st_build()`, `st_exit()`, `st_join()`, `fork()`
Library Functions -	`st_default_attr()`, `st_setcancel()`, `st_testcancel()`, `st_cancel()`

NAME

st_pushrecover,
st_poprecover – push and pop recovery routines on a thread's cancellation stack

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_pushrecover(func, arg)
void(*func)();
void *arg;

int st_poprecover(execute)
int execute;
```

DESCRIPTION

st_pushrecover and st_poprecover manage a stack of cleanup routines to be executed if the calling thread is canceled via st_cancel or st_testcancel, or if the thread itself exits (either prematurely, by using pthread_exit, or by returning from the function at which the thread began executing). The cleanup stack is initially empty; cleanup routines (**func**) and their arguments (**arg**) are pushed, as single stack elements, onto the cleanup stack using st_pushrecover. The stack is popped using st_poprecover; if the **execute** argument is nonzero, then the function popped is called with the argument popped. Otherwise, the popped stack element is simply discarded. In the event that the thread exits, (due either to being canceled, to calling st_exit itself, or to returning from its initial function), the cleanup stack is popped until empty, each function being called with the pushed parameter in LIFO order.

_ _st_pushrecover and _ _st_poprecover are macro implementations of the functions above. They provide a certain speedup in pushing and popping cleanup routines, but must be nested within the same lexical scope of a single C routine.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread- specific signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

DIAGNOSTICS

`st_pushrecover` returns 0 on successful completion of the push. If the function fails, it returns -1 and sets **errno** to one of the following values:

`ENOSPC`	Insufficient memory to push the new stack element.

`st_poprecover` returns 0 on successful completion of the pop. If the function fails, it returns -1 and sets **errno** to one of the following values:

`EAGAIN`	The cleanup stack is empty.

`_ _st_pushrecover` and `_ _st_poprecover` never fail at runtime; however, each `_ _st_pushrecover` must have a matching `_ _st_poprecover` in the same lexical scope or the function will not compile.

SEE ALSO

Library Functions - `st_cancel()`, `st_setcancel()`, `st_testcancel()`

NAME

`strcoll` – string collation

SYNOPSIS

```
#include <string.h>

int strcoll(s1, s2)
const char *s1;
const char *s2;
```

DESCRIPTION

The `strcoll` function compares the string pointed to by **s1** to the string pointed to by **s2**, both interpreted as appropriate to the `LC_COLLATE` category of the current locale. It returns an integer greater than, equal to, or less than zero, accordingly as string **s1** is greater than, equal to, or less than string **s2** when both are interpreted as appropriate to the current locale.

The `strcoll` and `strxfrm` functions provide for locale-specific string sorting. In situations where the number of comparisons per string is small, `strcoll` is sufficient. When the number of comparisons per string is large, it is better to use `strxfrm` and `strcmp` because the transformation process occurs only once per string.

SEE ALSO

Library Functions - `localeconv()`, `setlocale()`, `string()`, `strxfrm()`

NAME

`strerror` – get system error message

SYNOPSIS

```
#include <string.h>

char *strerror(errnum)
int errnum;
```

DESCRIPTION

`strerror` maps the error number in **errnum** to an error message string and returns a pointer to the string. The string returned by `strerror` is identical to that printed by `perror`. If *errnum* is not a valid value for **errno**, `strerror` will return NULL.

The string returned by `strerror` should not be modified by the program. This string may be overwritten by a subsequent call to `strerror`.

SEE ALSO

Library Function - `perror()`

NAME

`string: strcat, strdup, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, strtok_r, strstr, index, rindex` – string operations

SYNOPSIS

```
#include <string.h>

char *strcat(s1, s2)
char *s1, *s2;

char *strdup(s1)
char *s1;

char *strncat(s1, s2, n)
char *s1, *s2;
int n;

int strcmp(s1, s2)
char *s1, *s2;

int strncmp(s1, s2, n)
char *s1, *s2;
int n;

char *strcpy(s1, s2)
char *s1, *s2;

char strncpy(s1, s2, n)
char *s1, *s2;
int n;

int strlen(s)
char *s;

char *strchr(s, c)
char *s;
int c;
```

```
char *strrchr(s, c)
char *s;
int c;

char *strpbrk(s1, s2)
char *s1, *s2;

int strspn(s1, s2)
char *s1, *s2;

int strcspn(s1, s2)
char *s1, *s2;

char *strtok(s1, s2)
char *s1, *s2;

char *strtok_r(s1, s2, lasts)
char *s1, *s2, **lasts;

char *strstr(s1, s2)
char *s1, *s2;

char *index(s, c)
char *s;
int c;

char *rindex(s, c)
char *s;
int c;
```

DESCRIPTION

The arguments **s1**, **s2**, and **s** point to strings (arrays of characters terminated by a null character). The functions `strcat`, `strncat`, `strcpy`, and `strncpy` all alter **s1**; they do not check for overflow.

`strcat` Appends a copy of string **s2** to the end of string **s1**. Returns a pointer to the null- terminated result. `strncat` similar to `strcat`, but appends at most n characters. Returns a pointer to the null-terminated result.

`strdup`	Returns a pointer to a new string which is a duplicate of the string pointed to by **s1**. The space for the new string is obtained using `malloc`. If the new string cannot be created, NULL is returned.
`strcmp`	Compares its arguments and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**.
`strncmp`	Similar to `strcmp`, but examines at most **n** characters.
`strcpy`	Copies string **s2** to **s1**, stopping after the null character has been copied. Returns **s1**.
`strncpy`	Copies exactly **n** characters from **s2** to **s1**, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null- terminated if the length of **s2** is **n** or more. Returns **s1**.
`strlen`	Returns the number of characters in s, not including the terminating null character.
`strchr(strrchr)`	Returns a pointer to the first (last) occurrence of character **c** in string **s**, or a NULL pointer if **c** does not occur in the string. The null character terminating a string is considered to be part of the string.
`strpbrk`	Returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a NULL pointer if no character from **s2** exists in **s1**.
`strspn (strcspn)`	Returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.
`strtok`	Tokenizes the string **s1**—one token per call—using the separator characters from the string **s2**. The first call is made with **s1** specified, and returns a pointer to the first character of the first token. Later calls are made with **s1** a NULL pointer, and return pointers to subsequent tokens. Every call writes a null character into **s1** immediately following the last character of the returned token. A span of two or more consecutive characters

	from **s2** is considered a single separator. The string **s2** can even be different for each call. When no more tokens remain in **s1**, a NULL pointer is returned.
`strtok_r`	Is the thread-reentrant version of strtok. It performs the same function as strtok, but maintains its state in the pointer pointed to by **lasts**, rather than in static storage between calls.
`strstr`	Returns a pointer to the first occurrence of string **s2** within string **s1**, or a NULL pointer if string **s2** does not occur in string **s1**. If **s2** points to a string with zero length, strstr returns **s1**.
`index (rindex)`	Returns a pointer to the first (last) occurrence of **c** in string **s**. NULL is returned if **c** is not in **s**.

NOTE

All of the string functions are declared with appropriate types in `string.h`.

SEE ALSO

Library Functions - `ctype()`, `xstring()`, `bstring()`, `strod()`, `strol()`

NAME

`strtod` – convert ASCII string to double-precision number

SYNOPSIS

```
double strtod(str, ptr)
char *str, **ptr;
```

DESCRIPTION

`strtod` converts the character string pointed to by **`str`** to a `double` value. The first unrecognized character ends the conversion.

`strtod` recognizes an optional string of white-space characters, then an optional sign, then a string of digits optionally containing a decimal point, then an optional `E` or `e` followed by an optionally signed integer.

If **`ptr`** is not `(`**`char`**` **)NULL`, then `strtod` stores the address of the first unrecognized character in the location pointed to by **`ptr`**. If conversion is not possible, then `strtod` returns 0 and stores the value of `str` in the location pointed to by **`ptr`**.

NOTE

The function `atof(`**`str`**`)` is equivalent to
`strtod(`**`str,`**` (`**`char`**` **)NULL)`.

DIAGNOSTICS

`strtod` uses `IEEEerr` to determine the validity of the converted number. If the conversion would cause positive overflow, `strtod` returns `HUGE_VAL` (defined in `math.h`). If the conversion would cause negative overflow, `strtod` returns `-HUGE_VAL`. If the conversion would cause underflow, `strtod` returns 0. In all error cases, `strtod` sets **`errno`** to `ERANGE`.

SEE ALSO

Library Functions - `atof()`, `atoi()`, `atol()`, `catoi()`, `catol()`, `IEEEerr()`, `strtol()`

NAME

`strtol` – convert ASCII string to integer with base

SYNOPSIS

```
long strtol(str, ptr, base)
char *str, **ptr;
int base;
```

DESCRIPTION

`strtol` converts the character string pointed to by **`str`** to a **`long`** integer. The first unrecognized character (according to the base) ends the conversion. Leading white space characters are ignored.

If **`ptr`** is not `(`**`char`**` **)NULL`, then `strtol` stores the address of the first unrecognized character in the location pointed to by **`ptr`**. If conversion is not possible, then `strtol` returns 0 and stores the value of **`str`** in the location pointed to by **`ptr`**.

`base` is the conversion base, as long as it is between 2 and 36, inclusive. In this case, `strtol` accepts an optional leading sign character and ignores leading zeroes. If **`base`** is 16, then a leading `0x` or `0X` is also ignored.

If **`base`** is 0, then `strtol` determines the base from the string itself: After an optional leading sign character, a leading 0 indicates octal conversion, and a leading `0X`or `0X` indicates hexadecimal (base 16) conversion. In all other cases, `strtol` performs decimal conversion.

NOTE

The function `atol(str)` is equivalent to

```
strtol(str,(char **)NULL, 10)
```

The function `atoi(`**`str`**`)` is equivalent to

```
(int) strtol(str, (char **)NULL, 10)
```

DIAGNOSTICS

`strtol` returns the conversion of the string **`str`** as a long integer if successful, and 0 if not.

If **`ptr`** is not `(char **)NULL`, and the conversion was successful, then `strtol` stores the address of the first unrecognized character in the location pointed to by **`ptr`**.

If **`ptr`** is not `(char **)NULL`, and the conversion was not successful, then `strtol` stores the value of **`str`** in the location pointed to by **`ptr`**.

SEE ALSO

Library Functions - `atof()`, `atoi()`, `atol()`, `catoi()`, `catol()`, `IEEEerr()`, `strtod()`

NAME

`strxfrm` – string transformation

SYNOPSIS

```
#include <string.h>

size_t strxfrm(s1, s2, n)
char *s1;
const char *s2;
size_t n;
```

DESCRIPTION

The `strxfrm` function transforms the string pointed to by **s2** and places the resulting string into the array pointed to by **s1**. The transformation is such that if `strcmp` is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of `strcoll` applied to the same two original strings. No more than **n** characters are placed into the resulting array pointed to by **s1**, including the terminating NULL character. If **n** is 0, and **s1** is a NULL pointer, `strxfrm` returns the number of bytes required for the transformed string. If copying takes place between objects that overlap, the behavior is undefined.

The `strxfrm` function returns the length of the transformed string (not including the terminating NULL character). If the value returned is **n** or more, the contents of the array pointed to by **s1** are indeterminate.

EXAMPLES

The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by **s**:

```
1 + strxfrm(NULL, s, 0);
```

SEE ALSO

Library Functions - `localeconv()`, `setlocale()`, `string()`, `strcoll()`

NAME

`st_setcancel` – set calling system thread's cancelability status.

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_setcancel(state)
int state;
```

DESCRIPTION

`st_setcancel` sets the calling **st**'s cancelability status to the status given by **state**. **state** is the `bitwise-OR` of the following values:

`CANCEL_GENERAL`	Whether the **st** is cancelable at all.
`CANCEL_ASYNC`	Whether a cancelable **st** will be canceled immediately or only at synchronous cancellation points, described below.

When the **st**'s cancelability has `CANCEL_ASYNC` set to `TRUE`, the **st** will immediately be canceled by `st_cancels` issued against it. When the **st**'s cancelability has `CANCEL_ASYNC` set to FALSE, then `st_cancels` issued against the **st** will cause it to be canceled the next time it reaches a cancellation point. A **st** is never canceled so long as it has `CANCEL_GENERAL` set to `FALSE`. Instead, the cancel request is held pending until the **st**'s cancelability changes so that `CANCEL_GENERAL` is `TRUE`; at that time the **st** is canceled either synchronously or asynchronously, depending on the value of `CANCEL_ASYNC`. `CANCEL_GENERAL` overrides `CANCEL_ASYNC`.

System thread cancellation points occur:

when the **st** issues a `cv_wait`.

when the **st** issues a `st_testcancel`.

when the **st** issues a `st_join`.

When the **st** is finally terminated, any cleanup routines on the **st**'s cleanup stack will be executed in LIFO order by the **st**. A value of THREAD_CANCELED is given as the **st**'s exit status to any **st**'s performing a st_join with the canceled **st**.

An **st**'s initial cancellation state is CANCEL_GENERAL.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, SIGTHREADKILL. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle SIGTHREADKILL.

DIAGNOSTICS

st_setcancel returns the **st**'s cancelability status, or -1 upon error. In the case where a **st** has a pending cancellation request and st_setcancel causes THREAD_CANCELABLE to transition to TRUE, the **st** will be immediately terminated, so in this case the call will never return.

SEE ALSO

System Calls -	cv_wait(), st_join(), st_build()
Library Functions -	st_cancel(), st_testcancel(), st_pushrecover(), st_poprecover(), st_new()

NAME

st_testcancel – allow the calling system thread to be terminated now

SYNOPSIS

```
#include <time.h>

#include <st.h>

void st_testcancel()
```

DESCRIPTION

st_testcancel allows the calling **st** to be terminated if another **st** has issued a st_cancel against this system thread and this **st'**s cancelability state has CANCEL_GENERAL set to TRUE and CANCEL_ASYNC set to FALSE.

If the calling **st'**s cancelability status has CANCEL_GENERAL set to FALSE, then the st_testcancel does nothing. If the calling **st'**s cancelability has CANCEL_GENERAL set to TRUE and CANCEL_ASYNC set to TRUE, then it cannot have a pending cancellation request - any call to st_cancel would have already caused the **st** to terminate.

When the **st** is canceled, any cleanup routines on the **st'**s cleanup stack will be executed in LIFO order by the **st**. A value of THREAD_CANCELED is given as the **st'**s exit status to any **sts** performing a st_join with the terminated **st**.

The **st'**s initial cancellation state is CANCEL_GENERAL.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, SIGTHREADKILL. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle SIGTHREADKILL.

DIAGNOSTICS

st_testcancel never fails. It returns the cancelability state of the system thread, or does not return at all.

SEE ALSO

System Calls - `st_join()`, `cv_wait()`, `st_build()`

Library Functions - `st_cancel()`, `st_setcancel()`, `st_testcancel()`, `st_pushrecover()`, `st_poprecover()`, `st_new()`

NAME

`swab` – swap adjacent even and odd bytes

SYNOPSIS

```
void swab(from, to, nbytes)
char *from;
char *to;
int nbytes;
```

DESCRIPTION

`swab` exchanges adjacent even and odd bytes of the array pointed to by **`from`** and puts the result in the array pointed to by **`to`**. If **`nbytes`** is positive and even, `swab` exchanges **`nbytes`** bytes. If **`nbytes`** is positive and odd, `swab` exchanges **`nbytes-1`** bytes. If **`nbytes`** is not positive, `swab` does nothing.

NAME

`sysconf` – get configurable system variables

SYNOPSIS

#include <unistd.h>

```
long sysconf(name)
int name
```

DESCRIPTION

The `sysconf()` function provides a method for the application to determine the current value of a configurable system limit or option (variable).

The **name** argument represents the system variable to be queried. All of the variables listed in the following table are supported. The variables in the table "Mapping of System Variables to sysconf() Arguments" come from `<limits.h>` or **<unistd.h>** and the symbolic constants, defined in **<unistd.h>**, that are the corresponding values used for **name**.

Table 1: Mapping of System Variables to sysconf() Arguments

Variable	**name** Value
ARG_MAX	_SC_ARG_MAX
CHILD_MAX	_SC_CHILD_MAX
clock ticks/second	_SC_CLK_TCK
NGROUPS_MAX	_SC_NGROUPS_MAX
OPEN_MAX	_SC_OPEN_MAX
STREAM_MAX	_SC_STREAM_MAX
TZNAME_MAX	_SC_TZNAME_MAX
_POSIC_JOB_CONTROL	_SC_JOB_CONTROL
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_VERSION	_SC_VERSION

DIAGNOSTICS

If **name** is an invalid value, `sysconf()` returns `-1`. If the variable corresponding to **name** is associated with functionality that is not in the above table, `sysconf()` returns `-1` without changing the value of **errno**.

Otherwise, the `sysconf()` function returns the current variable value on the system. The value returned will be no more restrictive than the corresponding value described to the application when it was compiled with `<limits.h>` or **`<unistd.h>`**. The value will not change during the lifetime of the calling process.

If any of the following conditions occur, the `sysconf()` function returns -1 and set **errno** to the corresponding value:

`EINVAL`	The value of the **name** argument is invalid.

SPECIAL SYMBOL

The special symbol `CLK_TCK` shall yield the same result as `sysconf(_SC_CLK_TCK)`. It shall be defined in `<time.h>` The symbol `CLK_TCK` is a constant. This special symbol is obsolescent.

COPYRIGHT NOTICE

NAME

`syslog`, `vsyslog`, `openlog`,
`closelog`, `setlogmask` – control system log

SYNOPSIS

#include <syslog.h>

#include <varargs.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 the system console, log files, logged-in users, or forwarded to other machines as appropriate. (See `syslogd(1)`).

The message is identical to a `printf(3)` format string, except that `%m` is replaced by the current error message. (As denoted by the global variable `errno`; see `strerror(3)`). 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 `varargs(3)`.

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 OR'ing one or more of the following values:

LOG_CONS	If `syslog()` cannot pass the message to `syslogd` it will attempt to write the message to the console (`/dev/con`).
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(1)`, `su(1)`, `getty(1)`, 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 `ftpd(1)`, `routed(1)`, etc., 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_LPR`	The line printer spooling system: `lpr(1)`, `lpc(1)`, `lpd(1)`, etc.
`LOG_MAIL`	The mail system.
`LOG_NEWS`	The network news system.
`LOG_SYSLOG`	Messages generated internally by `syslogd(1)`.
`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.

EXAMPLE

```
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

Utility Programs - `syslogd`

HISTORY

These functions appeared in 4.2BSD.

COPYRIGHT

NAME

`system` – execute a program via the shell

SYNOPSIS

system(command)

char *command;

DESCRIPTION

`system` creates a new process and executes the shell, passing to it the string **command**. The string is parsed by the shell and the shell performs any commands contained therein. The parent process (the one which called `system`) waits for the shell to terminate. The parent also ignores `SIGINT` and `SIGQUIT` signals while the shell runs.

The child process inherits the standard file descriptors (and all others) from the parent process.

RETURN

`system` returns the exit status of the shell. If the shell couldn't be executed, `system` returns 127.

SEE ALSO

System Calls - `fork()`, `execve()`, `wait()`, `sigvec()`

Library Functions - `execl()`, `signal()`

NAME

`t_accept` – accept a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_accept(fd, resfd, call)
int fd;
int resfd;
struct t_call *call;
```

DESCRIPTION

A transport user issues this function to accept a connect request. **`fd`** identifies the local transport endpoint where the connect request arrived, while **`resfd`** is the local transport endpoint where the connection is to be established. **`call`** points to a **`t_call`** structure which contains information that the transport provider requires in order to complete the connection. A **`t_call`** structure has the following members:

```
strut    netbuf    addr;
struct   netbuf    opt;
struct   netbuf    udata;
int      sequence;
```

The `netbuf` structure is used in various transport interface functions to send and receive data. It contains:

```
unsigned   int   maxlen;
unsigned   int   len;
char       *buf
```

`buf` points to a user input/output buffer. `len` specifies the number of bytes within the buffer. If the structure is used for both input and output, `t_accept` will replace the user value of `len` at completion.

`maxlen` has significance only when `buf` receives output from the TI function. In this case, it specifies the physical size of the buffer (the maximum value of `len`). If `maxlen` is too small to hold the returned information, a `TBUFOVFLW` error will result. Certain functions, however, may return part of the data without generating an error.

In the **`call`** structure, `addr` is the caller's address; `opt` associates protocol-specific parameters with the connection; `udata` points to user data which is to be returned; and `sequence` is the value returned by `t_listen` which uniquely associates the response with a previously-received connect indication.

A connection can be accepted at either the local transport endpoint where the connect request arrived or at a different local transport endpoint. If the same endpoint is specified (`resfd=fd`), the connection can be accepted, unless other requests are pending which require a response. If this error occurs, then `t_accept` will fail with **`t_errno`** set to `TBADF`.

If a different local transport endpoint is specified by the transport user (`resfd!=fd`), that endpoint must be linked to a protocol address in the `T_IDLE` state before the `t_accept` is issued (see `t_getstate`).

In either case, `t_accept` will fail and set **`t_errno`** to `TLOOK` if there are connect or disconnect indications waiting on that endpoint.

The values of parameters specified by `opt` and the syntax of those values are protocol-specific. The `udata` argument enables the called user to send user data to the caller. The amount of user data must not exceed the limits supported by the transport provider as returned by `t_open` or `t_getinfo`. If the `len` field of `udata` is zero, no data will be sent to the caller.

`t_errno` may be set to one of the following:

`TBADF`	File descriptor does not refer to a transport endpoint; illegal use of the same endpoint on which the connect indication arrived.
`TOUTSTATE`	Wrong sequence issued on the transport endpoint referenced by **`fd`**; the transport endpoint referred to by **`resfd`** is not in the `T_IDLE` state.

TACCES	User does not have permission to accept a connection or use specified options.
TBADOPT	Options are incorrectly formatted or contain illegal information.
TBADDATA	Amount of user data specified is not within the bounds allowed by the transport provider.
TBADSEQ	Invalid sequence number.
TLOOK	An asynchronous event occurred on the transport endpoint referenced by **fd** and requires immediate attention.
TNOTSUPPORT	Function is not supported by the underlying transport provider.
TSYSERR	A system error has occurred during execution.

DIAGNOSTICS

The following values are returned:

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - t_connect(), t_getstate(), t_listen(), t_open(), t_rcvconnect()

NAME

t_alloc – allocate a library structure

SYNOPSIS

```
#include <tiuser.h>

char *t_alloc(fd, struct_type, fields)
int fd;
int struct_type;
int fields;
```

DESCRIPTION

t_alloc dynamically allocates memory for the various transport function argument structures (see below). t_alloc will allocate memory for the specified structure and buffers referenced by the structure.

struct_type specifies the structure to allocate, and can be:

```
T_BIND     struct t_bind
T_CALL     struct t_call
T_OPTMGMT  struct t_optmgmt
T_DIS      struct t_discon
T_UNITDATA struct t_unitdata
T_UDERROR  struct t_uderr
T_INFO     struct t_info
```

Each of these structures may be used as an argument to one or more transport functions.

Except T_INFO, each of the above structures contains at least one field of type struct netbuf. netbuf is described in t_accept. For each struct netbuf field, the user may specify that that specific buffers be allocated as well. **fields** specifies the buffers to be allocated with a bitwise-OR of any of the following:

T_ADDR	The addr field of t_bind, t_call, t_unitdata, or t_uderr.

`T_OPT`	The opt field of `t_optmgmt`, `t_call`, `t_unitdata`, or `t_uderr`.
`T_UDATA`	The udata field of `t_call`, `t_discon`, or `t_unidata`.
`T_ALL`	All relevant fields of the given structure.

Each buffer specified in **`fields`** will receive an allocation from `t_alloc`. `t_alloc` will then initialize the `buf` pointer and `maxlen` field accordingly. The length of the allocated buffer will be based on the size information returned to the user by `t_open` and `t_getinfo`. Thus, **`fd`** must refer to the local transport endpoint through which the just-allocated structure will pass, so that the size information can be accessed.

When the value associated with a specified field is -1 or -2, `t_alloc` will be unable to determine the size of the buffer to allocate and will fail, setting **`t_errno`** to `TSYSERR` and **`errno`** to `EINVAL`. For unspecified buffers in **`fields`**, `buf` will set to NULL and `maxlen` to zero.

Allocating structures with `t_alloc` will ensure user program compatibility with future releases of the transport interface.

`t_errno` may be set to:

`TBADF`	File descriptor does not refer to a transport endpoint.
`TSYSERR`	A system error occurred during execution.

DIAGNOSTICS

The following values are returned:

0	Successful completion.
-1	System call failure; **`t_errno`** is set.

SEE ALSO

Library Functions - `t_accept(), t_free(), t_getinfo(), t_open()`

NAME

`t_bind` – bind an address to a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_bind(fd, req, ret)
int fd;
struct t_bind *req;
struct t_bind *ret;
```

DESCRIPTION

`t_bind` associates a protocol address with a transport endpoint and activates the endpoint. The transport endpoint is specified by **`fd`**. The transport provider may accept or request connections on the transport endpoint in connection mode. The transport user may send or receive data units through the transport endpoint in connectionless mode.

`req` and **`ret`** point to a `t_bind` structure containing:

```
struct  netbuf  addr;
unsigned  qlen;
```

The `netbuf` structures are described in `t_accept`. In the `t_bind` structures, `addr` specifies a protocol address and `qlen` indicates the maximum number of incomplete connect indications.

The **`t_bind`** structure **`req`** requests that an address be linked to a given transport endpoint. `len`, `buf`, and `maxlen` of the `netbuf` structure `addr` are used in the following manner: `len` specifies the number of bytes in the address, while `buf` points to the address buffer, and `maxlen` has no significance.

In **`ret`**, the user specifies `netbuf maxlen` fields (maximum size of the address buffer) and `buf` (address buffer pointer). At completion, **`ret`** stores the address that the transport provider linked to the transport endpoint; this may differ from the address specified by the user in **`req`**.

On return, `len` dictates the number of bytes in the linked address and `buf` points to it. An error will occur if `maxlen` size is insufficient to hold the returned address.

If the requested address is unavailable or unspecified in **req**, the transport provider will assign an address to be linked, returning that address in the `addr` field of **ret**. To determine whether the transport provider linked the transport endpoint as requested, the user can compare **req** and **ret**.

If the user does not wish to specify an address to be linked, **req** may be NULL. The value of `qlen` is assumed to be zero, and an address must be assigned to the transport endpoint by the transport provider. **ret** may also be NULL if the user does not care what address is linked by the provider or the negotiated value of `qlen`.

req and **ret** can be set to NULL for the same call. In this case the provider chooses the address to link and does not return that information to the user.

The `qlen` field has significance only when a connection-mode service is initialized. `qlen` specifies how many outstanding connect indications the transport provider should support for the transport endpoint.

A transport user who expects only incoming calls (a passive transport user) can issue a `qlen` value >0. The value of `qlen`, can be changed by the transport provider, if the transport provider cannot support the specified number of incomplete connect indications. At completion, the value to be used will reside in the `qlen` field in **ret**.

`t_bind` allows more than one transport endpoint to be linked to the same protocol address; however, the transport provider must support this capability. Linking more than one protocol address to the same transport endpoint is not permitted. If multiple transport endpoints are linked to the same protocol address, only one will listen for the connect indications which are associated with that protocol address. Only one `t_bind` per protocol address will specify a value of `qlen` >0. As a result, the transport provider knows which transport endpoint to notify of an incoming connect indication.

If a protocol address is linked to a second transport endpoint with a value of `qlen` >0, the transport provider assigns a different address to be linked to that second endpoint. If a user accepts a connection on the listening transport endpoint, the linked protocol address will be busy for the duration of that connection.

Other transport endpoints may not be linked for listening as long as the initial listening endpoint is doing the data transfer. Thus, only one transport endpoint, linked to a protocol address, can accept connect indications at any one time.

t_errno may be set to:

TBADF	File descriptor does not refer to a transport endpoint.
TOUTSTATE	Function issued in wrong sequence.
TBADADDR	Protocol address incorrectly formatted or has illegal information.
TNOADDR	Transport provider cannot allocate an address.
TACCES	User does not have permission to use address.
TBUFAVFLW	The allocated number of bytes for incoming argument is insufficient to store the value. The provider's state will change to `T_IDLE` and returned information will be discarded.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

The following values are returned:

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - `t_open(), t_optmgmt(), t_unbind()`

NAME

`tcgetattr`, `tcsetattr` – get and set state

SYNOPSIS

```
#include <termios.h>

int tcgetattr(fd, termios_p)
int fd;
struct termios *termios_p;

int tcsetattr(fd, optional_actions, termios_p)
int fd;
int optional_actions;
struct termios *termios_p;
```

DESCRIPTION

The `tcgetattr()` function gets the parameters associated with the object referred to by **fd** and store them in the **termios** structure referenced by **termios_p**. This function is allowed from a background process; however, the terminal attributes may be subsequently changed by a foreground process. If the terminal device supports different input and output baud rates, the baud rates stored in the **termios** structure returned by `tcgetattr()` will reflect the actual baud rates, even if they are equal. If differing baud rates are not supported, the rate returned as the output baud rate will be the actual baud rate. The rate returned as the input baud rate will be either the number zero or the output rate (as one of the symbolic values).

The `tcsetattr()` function sets the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the **termios** structure referenced by **termios_p** as follows:

1. If **optional_actions** is `TCSANOW`, the change occurs immediately.
2. If **optional_actions** is `TCSADRAIN`, the change occurs after all output written to **fd** has been transmitted. This function should be used when changing parameters that affect output.

3. If **optional_actions** is TCSAFLUSH, the change occurs after all output written to the object referred to by **fd** has been transmitted, and all input that has been received, but not read, is discarded before the change is made.

The symbolic constants for the values of **optional_actions** are defined in **<termios.h>**.

The zero baud rate, B0, is used to terminate the connection. If B0 is specified as the output baud rate when tcsetattr() is called, the modem control lines will no longer be asserted. Normally, this will disconnect the line.

If the input baud rate is equal to the numeral zero in the **termios** structure when tcsetattr() is called, the input baud rate will be changed by tcsetattr() to the same value as that specified by the value of the output baud rate, exactly as if the input rate has been set to the output rate by cfsetispeed(). This usage of zero is obsolescent.

The tcsetattr() function returns success if it was able to perform any of the requested actions, even if some of the requested actions could not be performed. It sets all the attributes that the implementation does support as requested and leave all the attributes not supported by the hardware unchanged. If no part of the request can be honored, it returns -1 and sets **errno** to EINVAL. If the input and output baud rates differ and are a combination that is not supported, neither baud rate is changed. A subsequent call to tcgetattr() returns the actual state of the terminal device [reflecting both the changes made and not made in the previous tcsetattr() call]. The tcsetattr() function does not change the values in the **termios** structure whether or not it actually accepts them.

The effect of the tcsetattr() function is undefined if the value of the **termios** structure pointed to by **termios_p** was not derived from the result of a call to tcgetattr() on **fd**.

DIAGNOSTICS

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

If any of the following conditions occur, the tcgetattr() function returns -1 and sets **errno** to the corresponding value:

`EBADF`	The **`fd`** argument is not a valid file descriptor.
`ENOTTY`	The file associated with **`fd`** is not a terminal.

If any of the following conditions occur, the `tcsetattr()` function returns `-1` and sets **`errno`** to the corresponding value:

`EBADF`	The **`fd`** argument is not a valid file descriptor.
`EINTR`	A signal interrupted the `tcsetattr()` function.
`EINVAL`	The **`optional_actions`** argument is not a proper value, or an attempt was made to change an attribute represented in the **`termios`** structure to an unsupported value.
`ENOTTY`	The file associated with **`fd`** is not a terminal.

COPYRIGHT NOTICE

NAME

`tcgetpgrp` – get foreground process group ID

SYNOPSIS

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

pid_t tcgetpgrp(fildes)
int fildes;
```

DESCRIPTION

The `tcgetpgrp()` function returns the value of the process group ID of the foreground process group associated with the terminal specified by **fildes**.

The `tcgetpgrp()` function is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

DIAGNOSTICS

Upon successful completion, `tcgetpgrp()` returns the process group ID of the foreground process group associated with the terminal. If there is no foreground process group, `tcgetpgrp()` returns the value `NPROC`, as defined in `<param.h>`. Otherwise, a value of `-1` is returned and **errno** is set to indicate the error.

If any of the following conditions occur, the `tcgetpgrp()` function shall return `-1` and set **errno** to the corresponding value;

`EBADF`	The **fildes** argument is not a valid file descriptor.
`ENOSYS`	The `tcgetpgrp()` function is not supported in this implementation.
`ENOTTY`	The calling process does not have a controlling terminal, or the file is not the controlling terminal.

COPYRIGHT NOTICE

SEE ALSO

System Calls - `setsid()`, `setpgid()`

Library Functions - `tcsetpgrp()`

NAME

t_close – close a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_close(fd)
int fd;
```

DESCRIPTION

t_close informs the transport provider when a user is finished with a transport endpoint and frees the local library resources associated with the endpoint. The transport endpoint is specified by **fd**. t_close also closes the file associated with the transport endpoint.

t_close should only be called from the T_UNBND state. Since this function does not check state information, it may be called from any state; in which case, the endpoint's local library resources will be freed automatically. Also, a close system call will be issued for that file descriptor. close will be aborted if no other process has that file open; breaking any transport connection that may be associated with that endpoint.

t_errno may be set to:

TBADF File descriptor does not refer to a transport endpoint.

DIAGNOSTICS

The following values are returned:

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - t_getstate(), t_open(), t_unbind()

NAME

t_connect – set up a connection with another transport user

SYNOPSIS

```
#include <tiuser.h>

int t_connect(fd, sndcall, rcvcall)
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;
```

DESCRIPTION

t_connect allows a local transport user to establish a connection to a destination transport user. The local transport endpoint is identified by **fd**, and **sndcall** and **rcvcall** point to **t_call** structures. A **t-call** structure has the following members::

```
struct  netbuf  addr;
struct  netbuf  opt;
struct  netbuf  udata;
int  sequence
```

sndcall defines information the transport provider needs to establish a connection, while **rcvcall** defines information about the newly-established connection.

netbuf is described in t_accept. In **sndcall**, the protocol address of the destination transport user is specified in addr; protocol-specific information needed by the transport provider is specified in opt; optional user data which is to be passed to the destination transport user while connection is being established is specified in udata; and sequence has no significance.

At completion, **rcvcall** returns the protocol address which is associated with the responding transport endpoint in addr; the protocol-specific information in opt; the optional user data that may be returned during connection establishment in udata; and sequence has no significance.

The structure for the options of the underlying protocol passed in the opt argument is set by the transport provider. To use the default options for the underlying protocol, the len field in the opt structure is set to zero.

udata passes user data to the destination transport user and it receives user data from the destination transport user during the establishment of the connection. In **sndcall**, if the field len of the udata structure is zero, no data is passed to the destination transport. The maximum length of user data is limited by the transport provider

On return, the **rcvcall** structures addr, opt, and udata are updated to contain information about the connection. Therefore, before issuing t_connect, the maxlen field of each argument must be set in order to indicate the maximum size of the buffer for each. If rcvcall is NULL, no information is given to the user on return from t_connect.

t_connect waits for the destination user's response before returning control to the local user, unless O_NDELAY is set. A successful return indicates a successfully established connection. If O_NDELAY is set, t_connect executes asynchronously and the call does not wait for the remote user's response, but returns control immediately.

To the local user, the return code is set to -1 and **t_errno** is set to TNODATA, indicating that a connection has not been established. In asynchronous mode, t_connect initiates a connection establishment procedure.

t_errno may be set to:

TBADF	File descriptor does not refer to a transport endpoint.
TOUTSTATE	Function issued in wrong sequence.
TNODATA	O_NDELAY was set. Function initiated the connection establishment procedure, but did not wait for remote user response.
TBADADDR	Protocol address is formatted incorrectly.
TBADOPT	Options are incorrectly formatted or contain illegal information.

TBADDATA	Amount of user data is not within the bounds allowed by the transport provider.
TACCES	User does not have permission to use address or options.
TBUFAVFLW	The allocated number of bytes for incoming argument is insufficient to store the value. If in synchronous mode, the provider's state changes to T_DATAF and the connect indication information, returned in rcvcall, is discarded.
TLOOK	An asynchronous event occurred on transport endpoint; requires immediate attention.
TNOTSUPPORT	Function is not supported by transport provider.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

The following values are returned:

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - t_accept(), t_alloc(), t_getinfo(), t_listen(), t_open(), t_optmgmt(), t_rcvconnect()

NAME

`tcsendbreak, tcdrain, tcflush, tcflow` – line control functions

SYNOPSIS

```
#include <termios.h>

int tcsendbreak(fd duration)
int fd;
int duration;

int tcdrain(fd)
int fd;

int tcflush(fd, queue_selector)
int fd;
int queue_selector;

int tcflow(fd, action)
int fd;
int action;
```

DESCRIPTION

If the terminal is using asynchronous serial data transmission, the `tcsendbreak()` function causes transmission of a continuous stream of zero-valued bits for a specific duration. If **duration** is zero, it causes transmission of zero-valued bits for at least 0.25 seconds and not more than 0.5 seconds. If **duration** is not zero, it sends zero-valued bits for.

If the terminal is not using asynchronous serial data transmission, it is implementation defined whether the `tcsendbreak()` function sends data to generate a break condition (as defined by the implementation) or returns without taking any action.

The `tcdrain()` function waits until all output written to the object referred to by *fd* has been transmitted.

Upon successful completion, the `tcflush()` function has discarded any data written to the object referred to by *fd* but not transmitted, or data received, but not read, depending on the value of *queue_selector*:

1. If *queue_selector* is `TCIFLUSH`, it flushs data received, but not read.
2. If *queue_selector* is `TCOFLUSH`, it flushs data written, but not transmitted.
3. If *queue_selector* is `TCIOFLUSH`, it flushs both data received but not read and data written but not transmitted.

The `tcflow()` function suspends transmission or reception of data on the object referred to by *fd*, depending on the value of *action*:

1. If *action* is `TCOOFF`, it suspends output.
2. If *action* is `TOON`, it restarts suspended output.
3. If *action* is `TCIOFF`, the system transmits a `STOP` character, which is intended to cause the terminal device to stop transmitting data to the system.
4. If *action* is `TCION`, the system transmits a `START` character, which is intended to cause the terminal device to start transmitting data to the system.

The symbolic constants for the values of *queue_selector* and *action* are defined in **<termios.h>.**

The default on the opening of a terminal file is that neither its input nor its output is suspended.

DIAGNOSTICS

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

If any of the following conditions occur, the `tcsendbreak()` function returns `-1` and sets **errno** to the corresponding value;

`EBADF`	The *fd* argument is not a valid file descriptor.
`ENOTTY`	The file associated with *fd* is not a terminal.

If any of the following conditions occur, the `tcdrain()` function returns `-1` and sets **errno** to the corresponding value:

`EBADF`	The *fd* argument is not a valid file descriptor.
`EINTR`	A signal interrupted the `tcdrain()` function.
`ENOTTY`	The file associated with *fd* is not a terminal.

If any of the following conditions occur, the `tcflush()` function returns `-1` and sets **`errno`** to the corresponding value:

`EBADF`	The *fd* argument is not a valid file descriptor.
`EINVAL`	The *queue_selector* argument is not a proper value.
`ENOTTY`	The file associated with *fd* is not a terminal.

If any of the following conditions occur, the `tcflow()` function returns `-1` and sets **`errno`** to the corresponding value:

`EBADF`	The *fd* argument is not a valid file descriptor.
`EINVAL`	The *action* argument is not a proper value.
`ENOTTY`	The file associated with *fd* is not a terminal.

COPYRIGHT NOTICE

NAME

tcsetpgrp – set foreground process group ID

SYNOPSIS

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

tcsetpgrp(fd, pgrp)
int fd;
pid_t pgrp;
```

DESCRIPTION

If the process has a controlling terminal, the `tcsetpgrp()` function sets the foreground process group ID associated with the terminal to **pgrp**. The file 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** must match a process group ID of a process in the same session as the calling process.

DIAGNOSTICS

Upon successful completion, `tcsetpgrp()` returns a value of zero. Otherwise, a value of `-1` is returned and **errno** is set to indicate the error.

If any of the following conditions occur, the `tcsetpgrp()` function returns `-1` and sets **errno** to the corresponding value;

EBADF	The **fd** argument is not a valid file descriptor.
EINVAL	the value of the **pgrp_id** argument is not supported by the implementation.
ENOSYS	The `tcsetpgrp()` function is not supported in this implementation.
ENOTTY	The calling process does not have a controlling terminal, or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

EPERM	The value of **pgrp** is a value supported by the implementation, but does not match the process group ID of a process in the same session as the calling process.

COPYRIGHT NOTICE

SEE ALSO

System Calls - `setsid()`, `setpgid()`

Library Functions - `tcgetpgrp()`

NAME

termcap: tgetent,
tgetnum, tgetflag,
tgetstr, tgoto,
tputs, fgetent – terminal-independent display control

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp;
char *name;

int tgetnum(cap)
char *cap;

int tgetflag(cap)
char *cap;

char *tgetstr(cap, bufp)
char *cap;
char **bufp;

char *tgoto(cm, column, line)
char *cm;
int column, line;

tputs(s, lines, outfunc)
char *s;
int lines;
int(*outfunc)();

fgetent(bp, objectname, evarname, deffn)
char *bp;
char *objectname;
char *evarname;
char *deffn;
```

DESCRIPTION

These functions allow the terminal capability database (usually `/etc/termcap`) to be accessed and its descriptions used. Once the name of the database entry is determined, the `tgetent` routine should be called to initialize the system. The `name` of the database entry and a pointer, `bp`, to a buffer of at least 1024 bytes are passed to `tgetent`, which fills the buffer with the entry. The buffer must remain in existence so long as its contents are needed for calls to `tgetnum`, `tgetflag`, or `tgetstr`. If `tgetent` fails because the database could not be opened, it returns -1; if the requested entry cannot be found, it returns 0. `tgetent` returns 1 if all went well.

If the environment variable `TERMCAP` is present, `tgetent` examines it. If the value of `TERMCAP` begins with a slash (/), `tgetent` assumes it is the name of a file to search instead of `/etc/termcap`. If the value does not begin with a slash, `tgetent` assumes it is the database entry itself, and verifies that the name field matches the requested **`name`**. If not, `tgetent` fails; thus it is important that `TERMCAP` contain meaningful information.

Once a database entry has been loaded, calls to `tgetnum`, `tgetflag`, and `tgetstr` retrieve values from it. `tgetnum` retrieves the numeric value of capability `cap`. `tgetflag` returns 1 if `cap` is present in the database entry, or 0 if not. `tgetstr` copies the string value of `cap` into memory at the location pointed to by the pointer at `bufp`; `bufp` is advanced by the length of the string, including the terminating null. This strange behavior facilitates collection of all string-valued capabilities into one buffer. As an example, to load the string for the capabilities `cm`, `cl`, and `is` into a common pool of memory:

```
char common[512];
char *commonp;
char *cm_value;
char *cl_value;
char *is_value;
. . .
commonp = common;
cm_value = tgetstr("cm", &commonp);
cl_value = tgetstr("cl", &commonp);
is_value = tgetstr("is", &commonp);
```

If `tgetnum` cannot locate `cap`, it returns -1. If `tgetstr` cannot locate `cap`, it returns 0, and `bufp` remains unchanged. The `tgoto` routine uses the cursor positioning string `cm` to produce a character sequence that when written will move the cursor to position (`line`, `column`) on the screen. The string `cm` should have been obtained by a prior `tgetstr`. Note that `tgoto` returns the movement string, but `tgoto` does not print the movement string. If the **`line`** and **`column`** values when formatted according to the `cm` string result in a string that contains ASCII code 0, 4, 9, or 10 (null, EOF, tab, or line feed), the **`line`** and **`column`** values are amended appropriately and a number of corrective single position movement commands are appended. The global variables `UP` (from the "up" capability) and `BC` (from the "bs" capability, or simply `ASCII 8` if BC is zero) are used to correct the position. These variables must be set by the user software.

`tputs` sends each character in the string `s` to the function **`outfunc`**, one at a time. `tputs` also interprets padding codes at the beginning of `s`, and uses the `lines` parameter as a scaling factor on the delay time. The **`lines`** value should be 1 unless the amount of delay should be proportional to the number of lines affected by the sequence (see the `termcap` description in the File Formats chapter of this manual). The global variable `ospeed` should be set to a line speed value as per the `tty` driver's understanding thereof. The `ospeed` value determines how many pad characters to transmit. The global variable `PC` should be set to an appropriate pad character (usually null is sufficient).

The `fgetent` routine is a more flexible version of `tgetent`. The first two parameters, `bp` and `objectname`, serve the same function as the `bp` and `name` parameters of `tgetent`. The `evarname` parameter gives a default environment value to examine for a database file override or cached entry. The `deffn` parameter names the default database file. Thus a call to `tgetent`:

```
tgetent(mybuf, "myterm");
```

can also be expressed as a call to `fgetent`:

```
fgetent(mybuf, "myterm", "TERMCAP", "/etc/termcap");
```

The `fgetent` call is supplied so that other `termcap`-like databases for other purposes can be created and used easily.

SEE ALSO

Utility Programs -	`termcap`
Library Functions -	`ttyconfig()`
Device Facilities -	`tty`
File Formats -	`termcap`

NAME

t_error – produce error message

SYNOPSIS

```
#include <tiuser.h>

void t_error(errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
```

DESCRIPTION

t_error builds a message describing the most recent error encountered during a call to a transport function and outputs it on standard error. The string **errmsg** is supplied by the user in order to provide context to the error. The user's error message is followed by a colon, followed by the standard transport function error message for the current value contained in **t_errno**. If **t_errno** is TSYSERR, t_error will print the standard error message for the current value in **t_errno**.

t_errlist is the array of message strings; **t_nerr** is the maximum index value for the **t_errlist** array. **t_errno** can be used to index this array to retrieve the error message string. The user can then format the message for output.

t_errno is set when an error occurs and is not cleared on future successful calls.

NAME

`t_free` – free a library structure

SYNOPSIS

```
#include <tiuser.h>

int t_free(ptr, struct_type)
char *ptr;
int struct_type;
```

DESCRIPTION

`t_free` frees memory allocatd by `t_alloc`. `t_free` frees memory for the specified structure and for the buffers referenced by the structure.

`ptr` points to a structure allocated by `t_alloc`; **`struct_type`** identifies the type structure. Valid values for **`struct_type`** are:

T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UNDERROR	struct t_uderr
T_INFO	struct t_info

If **`ptr`** or any of the `buf` pointers points to a block of memory that was not previously allocated by `t_alloc`, undefined results will occur.

`t_errno` may set to:

[TSYSERR]	A system error occurred during execution.

DIAGNOSTICS

0	Successful completion.

-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - `t_alloc()`

NAME

t_getinfo – get protocol-specific service information

SYNOPSIS

```
#include <tiuser.h>

int t_getinfo(fd, info)
int fd;
struct t_info *info;
```

DESCRIPTION

t_getinfo returns the characteristics of the underlying transport protocol which is associated with the **fd** file descriptor. The **info** structure returns the same information returned by t_open. t_getinfo provides the transport user with access to this information throughout the communication process.

info points to a **t_info** structure which contains:

```
long addr;
/* max size of the transport protocol address */

long options;
/* max number of bytes of protocol-specific options */

long tsdu;
/* max size of transport service data unit (TSDU) */

long etsdu;
/* max size of an expedited transport service data unit (ETSDU) */

long connect;
/* max amount of data allowed on connection establishment
functions */

long discon;
/* max amount of data allowed on t_snddis and t_rcvdis functions
*/

long servtype;
/* service type supported by the transport provider */
```

The significance of field values is:

addr

	≥ 0	Max size of transport protocol address.
	-1	No address size limit.
	-2	Transport provider does not provide access to transport protocol addresses.

options

	≥ 0	Max number of protocol-specific option bytes supported by the provider.
	-1	No limit on option size.

	-2	Transport provider does not support user-settable options.
`tsdu`		
	> 0	Max size of a transport service data unit.
	0	Transport provider does not support `TSDU`, but it does support ending data without logical boundaries across a connection.
	-1	No limit on the size of a `TSDU`.
	-2	Transport provider does not support transfer of normal data.
`etsdu`		
	> 0	Max size of an expedited transport service data unit.
	0	Transport provider does not support `ETSDU`; it does support sending expedited data without logical boundaries across a connection
	-1	No limit on the `ETSDU` size.
	-2	Transfer of expedited data is not supported by the transport provider.
`connect`		
	≥ 0	Max amount of data that may be associated with connection establishment functions.
	-1	No limit on amount of data sent during connection.
	-2	Transport provider does not allow data to be sent with connection establishment functions.
`discon`		
	≥ 0	Max amount of data that may be associated with `t_snddis` and `t_rcvdis`.
	-1	No limit on the amount of data sent with abortive release functions.

	-2	Transport provider does not allow data to be sent with abortive release functions.
servtype	Service type supported by the transport provider. This field may display one of the following values:	
	T_COTS	Transport provider supports a connection-mode service, but not the optional systematic release facility.
	T_COS_ORD	Transport provider supports a connection-mode service with a optional systematic release facility.
	T_CLTS	Transport provider supports a connectionless-mode service. t_open will return -2 for etsdu, connect, and discon.

For protocol independence, the above sizes may be accessed to retrieve information on how large the buffers must be, or the t_alloc function may be used to allocate the buffers. Exceeding the allowed data size on any function will result in an error. Option negotiation can change the value of each field.

t_errno may be set to:

TBADF	File descriptor does not refer to a transport endpoint.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - t_open()

NAME

t_getstate – get the current state of the transport provider

SYNOPSIS

```
#include <tiuser.h>

int t_getstate(fd)
int fd;
```

DESCRIPTION

t_getstate returns the current state of the transport provider of the specified transport endpoint. The transport endpoint is specified by **fd**.

t_errno may be set to if t_getstate fails:

TBADF	File descriptor does not refer to a transport endpoint.
TSTATECHNG	The transport provider is in the middle of a state change.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

The following states are returned:

T_UNBND	Successful completion, unbound.
T_IDLE	Successful completion, idle.
T_OUTCON	Successful completion, outgoing connection pending.
T_INCON	Successful completion, incoming connection pending.
T_DATAXFER	Successful completion, data transfer.
T_OUTREL	Successful completion, outgoing release.
T_INREL	Successful completion, incoming release.
-1	t_getstate failure; **t_errno** is set.

SEE ALSO

Library Functions - `t_open()`

NAME

`time, ftime` – get system clock value

SYNOPSIS

```
long time(clock)
long *clock;

#include <types.h>
#include <timeb.h>

ftime(tp)
struct timeb *tp;
```

DESCRIPTION

`time` is a simplified interface to `gettimeofday`. It returns the second-count portion of the current system clock. The return value is also stored into the `long` pointed to by **`clock`**, unless **`clock`** is NULL. The system clock represents the number of seconds since the beginning of time, which to LynxOS is usually midnight GMT, 1 Jan 1970.

`ftime` fills in a structure defined as follows:

```
struct timeb {
  time_t  time;
  int     millitm;
  int     timezone;
  int     dstflag;
};
```

The structure contains the time (as per `time`), an extra word of resolution of this time (at millisecond resolution; not guaranteed accurate), the timezone expressed as a number of minutes west of GMT, and a flag which is set to a non- zero value if daylight saving time should be applied at appropriate times.

NOTE

The `dstflag` field of the `timeb` structure tells whether daylight saving time should be applied, not whether daylight saving time is currently in effect.

SEE ALSO

Utility Programs - `date`

System Calls - `gettimeofday()`

Library Functions - `ctime()`

NAME

`t_listen` – listen for a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_connect(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

`t_listen` allows a local transport user to listen for a connect request from a calling transport user. The local transport endpoint where the connection indication will arrive is identified by **fd**, and **call** points to **t_call** structure when `t_listen` returns. A **t_call** structure has the following members:

```
struct  netbuf  addr;
struct  netbuf  opt;
struct  netbuf  udata;
int  sequence
```

`netbuf` is described in `t_accept`. **call** returns the protocol address of the calling transport user in `addr`; the protocol-specific information of the connection in `opt`; the optional user data that the caller may send with the connection request in `udata`; and a sequence number in `sequence`. `sequence` allows the listening user to examine multiple connection indications before responding.

On return, the **call** structures `addr`, `opt`, and `udata` are updated to contain information about the connection. Therefore, before issuing `t_listen`, the `maxlen` field of each `netbuf` structure must be set in order to indicate the maximum size of the buffer for each.

`t_listen` works synchronously and waits for a connection indication before returning control to the local user, unless `O_NDELAY` is set. If `O_NDELAY` is set, `t_connect` executes asynchronously and the call does not wait, but returns control immediately. In asynchronous mode if no connections exist `t_listen` returns -1 and sets **t_errno** to `TNODATA`.

t_errno may be set to:

TBADF	File descriptor does not refer to a transport endpoint.
TNODATA	O_NDELAY was set and no connection indications were queued.
TBUFAVFLW	The allocated number of bytes for incoming argument is insufficient to store the value. The provider's state changes to T_INCON and the connect indication information, returned in **call**, is discarded.
TLOOK	An asynchronous event occurred on transport endpoint; requires immediate attention.
TNOTSUPPORT	Function is not supported by transport provider.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

The following values are returned:

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - `t_accept(), t_alloc(), t_bind(), t_connect(), t_open(), t_rcvconnect()`

NAME

`t_look` – look at the current event at a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_look(fd)
int fd;
```

DESCRIPTION

`t_look` returns the current event on the transport endpoint. The transport endpoint is specified by **fd**.

`t_look` allows the transport provider to inform the transport user of an asynchronous event while the user is operating in synchronous mode. Events that require immediate notification of the user cause the current or next TLI function to return the error `TLOOK`. Upon receipt of a `TLOOK` error the function `t_look` should be called.

`t_look` can also be used to pool the transport endpoint for asynchronous events.

t_errno may be set if `t_getstate` fails:

`TBADF`	File descriptor does not refer to a transport endpoint.
`TSYSERR`	A System error occurred during execution.

DIAGNOSTICS

The following events are returned:

`T_LISTEN`	Successful completion, connection indication received.
`T_CONNECT`	Successful completion, connect confirmation received.
`T_DATA`	Successful completion, normal data received.
`T_EXDATA`	Successful completion, expidited data received.
`T_DISCONNECT`	Successful completion, disconnect received.

`T_ERROR`	Successful completion, fatal error indication.
`T_UDERR`	Successful completion, datagram error indication.
`T_ORDREL`	Successful completion, orderly release indication.
-1	`t_getstate` failure; **`t_errno`** is set.

SEE ALSO

Library Functions - `t_open()`

NAME

`tmpfile` – create a temporary file

SYNOPSIS

`#include <stdio.h>`

`FILE *tmpfile()`

DESCRIPTION

`tmpfile` creates a temporary file using a name generated by `tmpnam`, and returns a corresponding **`FILE`** pointer. If the file cannot be opened, a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update (`w+`).

SEE ALSO

System Calls - `unlink()`

Library Functions - `tmpnam()`

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam(s)
char *s;

char *tempnam(dir, pfx)
char *dir, *pfx;
```

DESCRIPTION

tmpnam and tempnam generate filenames for temporary files.

tmpnam generates a file name using the path-prefix defined as P_tmpdir in **stdio.h**. If **s** is NULL, tmpnam leaves its result in an internal static area and returns a pointer to that area. The next call to tmpnam will destroy the contents of the area. If **s** is not NULL, it is assumed to be the address of an array of at least L_tmpnam bytes, where L_tmpnam is a constant defined in **stdio.h**; tmpnam places its result in that array and returns **s**.

tempnam allows the user to control the choice of a directory. The argument **dir** points to the name of the directory in which the file is to be created. If **dir** is NULL or points to a string that is not a name for an appropriate directory, the path-prefix defined as P_tmpdir in **stdio.h** is used. If that directory is not accessible, /tmp will be used. As an alternative, if the environment variable TMPDIR is defined and has a value, then its value takes precedence over both **dir** and P_tmpdir as the name of the desired temporary-file directory.

The **pfx** argument can be used to give a prefix to the temporary-file name. **pfx** may be NULL or point to a string of up to five characters.

tempnam uses malloc to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from tempnam may serve as an argument to free. tempnam returns NULL if malloc fails, or if all attempts to locate an accessible directory fail.

NOTE

These functions generate a different file name each time they are called. If called more than 456,976 (26^4) times in a single process, these functions will start recycling previously-used names.

SEE ALSO

Library Functions - `tmpfile()`, `mktemp()`, `malloc()`, `free()`

NAME

`t_open` – open a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_open(path, oflags, info)
char *path;
int oflags;
struct t_info *info;
```

DESCRIPTION

`t_open` is the first step in the setup of a transport endpoint. A transport endpoint is established by opening a UNIX system file that identifies a transport protocol. The file descriptor that is returned by `t_open` identifies the local transport endpoint. The full path name of the file to be opened is pointed to by **path**. The open flags that are to be used are found in **oflags**. See manual page for `open` System Call for **oflags** definitions.

The default characteristics of the transport endpoint's underlying transport protocol is returned in a **t_info** structure pointed to by **info**. A **t_info** structure contains:

```
long addr;
    /* max size of the transport protocol address */
long options;
    /* max number of bytes of protocol-specific options */
long tsdu;
    /* max size of transport service data unit (TSDU) */
long etsdu;
    /* max size expedited transport service data unit (ETSDU) */
long connect;
    /* max data allowed on connection establishment functions */
long discon;
    /* max data allowed on t_snddis and t_rcvdis functions */
long servtype;
    /* service type supported by the transport provider */
```

The significance of field values is:

`addr`

≥0	Max size of transport protocol address.
-1	No address size limit.
-2	Transport provider does not provide access to transport protocol addresses.

`options`

≥0	Max number of protocol-specific option bytes supported by the provider.
-1	No limit on option size.
-2	Transport provider does not support user-settable options.

`tsdu`

>0	Max size of a transport service data unit.
0	Transport provider does not support `TSDU`, but it does support sending data without logical boundaries across a connection.
-1	No limit on the size of a `TSDU`.
-2	Transport provider does not support transfer of normal data.

`etsdu`

>0	Max size of an expedited transport service data unit.
0	Transport provider does not support `ETSDU`; it does support sending expedited data without logical boundaries across a connection.
-1	No limit on the `ETSDU` size.
-2	Transfer of expedited data is not supported by the transport provider.

`connect`

	≥0	Max amount of data that may be associated with connection establishment functions.
	-1	No limit on amount of data sent during connection.
	-2	Transport provider does not allow data to be sent with connection establishment functions.
`discon`		
	≥0	Max amount of data that may be associated with `t_snddis` and `t_rcvdis`.
	-1	No limit on the amount of data sent with abortive release functions.
	-2	Transport provider does not allow data to be sent with abortive release functions.
`servtype`	The service type supported by the transport provider.	
	`T_COTS`	Transport provider supports a connection-mode service, but not the optional systematic release facility.
	`T_COS_ORD`	Transport provider supports a connection-mode service with a optional systematic release facility.
	`T_CLTS`	Transport provider supports a connectionless-mode service. `t_open` will return -2 for `etsdu`, `connect`, and `discon`.

For protocol independence, the above sizes may be accessed to retrieve information on how large the buffers must be, or the `t_alloc` function may be used to allocate the buffers. Exceeding the allowed data size on any function will result in an error.

`info` should be set to NULL if no protocol information is to be returned by `t_open`.

`t_errno` may be set to:

`TSYSERR`	A system error occurred during execution.

DIAGNOSTICS

>0	Successful completion; return value is a valid file descriptor.
-1	System call failure; **`t_errno`** is set.

SEE ALSO

System Call - `open()`

NAME

`t_optmgmt` – manage options for a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

cint t_optmgmt(fd, req, ret)
int fd;
struct t_optmgmt *req;
struct t_optmgmt *ret;
```

DESCRIPTION

`t_optmgmt` is used by the transport user to retrieve, verify or negotiate the protocol options of the underlying transport protocol of the specified transport endpoint. The transport endpoint is specified by **fd**.

req and **ret** point to a **t_optmgmt** structure containing:

```
struct  netbuf  opt;
long  flags;
```

The `netbuf` structures are described in `t_accept`. In the **t_optmgmt** structures, `opt` specifies the protocol options and `flags` defines what should be done with the options.

Actions are requested of the transport provider and protocol options are sent to the transport provider in **req**. In **req**, the number of bytes for the options is specified by `len` and the buffer containing the options is pointed to by `buf`. `maxlen` has no meaning in **req**.

The transport provider returns option and flag information in **ret**. In **ret**, the size of the options buffer is specified in `maxlen` and the buffer where the option information is to be placed is pointed to by `buf` before `t_optmgmt` is called. Upon return, the buffer will be filled and the length to the option information returned will be specified in `len`.

The transport provider determines the structure and content of the options.

flags can specify one of the floowing actions.

T_NEGOTIATE	Enable the transport user to negotiate with the transport provider the values of the options specified in **req**. The transport provider evaluates the options passed in **req**, negotiates the values and returns the values in **ret**.
T_CHECK	Enables the transport user to determine if the transport provider supports the options specified in **req**. The transport provider will set the flag field of **ret** with either T_SUCCESS or T_FAILURE to indicate whether or not the option is supported.
T_DEFAULT	Enables the transport user to retrieve the default options of the transport provider. The default options are returned in the opt field of **ret**. In **req**, the len field of opt must be 0.

t_optmgmt may block because of flow control constraints when it is issued as part of a connectionless-mode service. t_optmgmt will not return until all previously-sent data units are processed by the transport provider.

t_errno may be set to:

TBADF	File descriptor does not refer to a transport endpoint.
TOUTSTATE	Function issued in wrong sequence.
TBADOPT	The format of the specified protocol options was incorrect.
TBADFLAG	Invalid flag.
TACCES	User does not have permission to negotiate the options specified.
TBUFAVFLW	The allocated number of bytes for incoming argument is insufficient to store the value. The information returned in **ret** will be discarded.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

The following values are returned:

0	Successful completion.
-1	System call failure; **`t_errno`** is set.

SEE ALSO

Library Functions - `t_getinfo(), t_open()`

NAME

t_rcv – receive data sent over a connection

SYNOPSIS

```
#include <tiuser.h>

int t_rcv(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int *flags;
```

DESCRIPTION

t_rcv receives normal or expedited data. **fd** identifies the local transport endpoint. **buf** points to the user data receive buffer; **nbytes** specifies the size of the receive buffer; **flags** points to optional flags and is set by t_rcv upon return.

t_rcv operates in synchronous mode unless O_NDELAY is set via t_open or fcntl. In synchronous mode, t_rcv will wait for data to arrive if none is immediately available. If O_NDELAY is set, t_rcv will execute asynchronously and fail immediately if there is no data available.

If the current transfer data unit (TSDU) or expedited transport data unit (ETSDU) contains more than **nbytes** the T_MORE flag will be set. The T_MORE indicates to the transport user that t_rcv must be immediately called again to retrieve the rest of the TSDU. T_MORE will continue to be set until the complete TSDU has been received. T_MORE is not meaningful and should be ignored if the transport provider does not support the TSDU concept. The tsdu field of the info structure returned by t_open or t_getinfo will specify whether or not the TSDU concept is supported.

If t_rcv returns with T_EXPEDITED set, then the data returned is an ETSDU. If T_MORE is also set, then t_rcv must be called immediately to retrieve the rest of the ETSDU. The T_EXPEDITED flag will not be set upon the return of any subsequent t_rcv call. When t_rcv returns and the T_MORE flag is not set, the end of the ETSDU has been read.

An ETSDU can arrive while the transport user is receiving a TSDU. When this happens, the receipt of the TSDU is suspended while the ESTDU is received. The receipt of the TSDU will continue after ETSDU receipt is complete.

t_errno may set to:

TBADF	File descriptor does not refer to a transport endpoint.
TNODATA	O_NDELAY was set and no data was available.
TLOOK	An asynchronous event that requires immediate attention occurred on this transport endpoint.
TNOTSUPPORT	Function not supported by the underlying transport provider.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

Positive value	Successful completion, the number of bytes received.
-1	Failure; **t_errno** is set.

SEE ALSO

Library Functions - t_open(), t_snd()

NAME

`t_rcvconnect` – receive confirmation from a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_rcvconnect(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

`t_rcvconnect` allows a transport user to check the status of a previously-sent connect request. It is used in conjunction with `t_connect` to set up a connection while in asynchronous mode. If `t_rcvconnect` returns successfully, then the connection has been established.

`fd` identifies the connection's local transport endpoint; **`call`** is a pointer to a **`t_call`** structure that contains information about the new connection. A **`t_call`** structure contains:

```
struct  netbuf  addr;
struct  netbuf  opt;
struct  netbuf  udata;
int  sequence
```

`netbuf` is described in `t_accept`. The protocol address for the responding transport endpoint is returned in `addr`. Protocol-specific information for the connection is returned in `opt`. Optional user data from the destination transport user may be returned in `udata`. `t_rcvconnect` does not use `sequence`. Before `t_rcvconnect` can be called, the `maxlen` field of `addr`, `opt` and `udata` must be set to the maximum size of the buffer associated with `addr`, `opt` and `udata`.

If **`call`** is NULL, no information is returned to the user.

`t_rcvconnect` works synchronously and waits for a connection indication before returning control to the local user, unless `O_NDELAY` is set. If `O_NDELAY` is set, `t_rcvconnect` executes asynchronously and does not wait, but returns control immediately. In asynchronous mode if no connection is available, `t_rcvconnect` returns -1 and sets **`t_errno`** to `TNODATA`. In this case `t_rcvconnect` must be issued again at a later time to complete the connection.

`t_errno` may set to:

`TBADF`	File descriptor does not refer to a transport endpoint.
`TBUFOVFLW`	The allocated number of bytes for incoming argument is insufficient to store the value. The provider's state changes to `DATAXFER`, and the connect indication information, returned in **`call`**, is discarded.
`TNODATA`	`O_NDELAY` is set, but no connection is available yet.
`TLOOK`	An asynchronous event has occurred on this transport requires immediate attention.
`TNOTSUPPORT`	Function not supported by the underlying transport provider.
`TSYSERR`	A system error occurred during function's execution.

DIAGNOSTICS

0	Successful completion.
-1	Failure; **`t_errno`** is set.

SEE ALSO

Library Functions – `t_accept()`, `t_alloc()`, `t_bind()`, `t_connect()`, `t_listen()`, `t_open()`

NAME

t_rcvdis – retrieve information from disconnect

SYNOPSIS

```
#include <tiuser.h>

int t_rcvdis(fd, discon)
int fd;
struct t_discon *discon;
```

DESCRIPTION

t_rcvdis identifies the cause of a disconnect and retrieves any user data set with the disconnect. **fd** identifies the connection's local transport endpoint; **discon** is a pointer to a t_discon structure that contains information about the disconnect and any user data. A t_discon structure contains:

```
struct  netbuf  udata;
int  reason;
int  sequence;
```

netbuf is described in t_accept. A protocol-dependent reason code explaining the reason for the disconnect is returned in reason. User data, if any, that was sent with the disconnect is returned in udata. Any outstanding connect indications that are associated with the disconnect will be identified by a value returned in sequence. The value in sequence is only meaningful for a passive transport user who has previously called t_listen and is processing the connect information that was returned when t_rcvdis is called.

discon can be NULL. In this case any information or user data associated with the disconnect is lost. This should only be done if the user he does not need any of the information in reason, sequence, or udata.

t_errno may set to:

TBADF	File descriptor does not refer to a transport endpoint.
TNODIS	No indication of disconnect currently exists.

TBUFOVFLW	The allocated number of bytes for incoming data is insufficient to store data. The provider's state changes to T_IDLE and the disconnect indication information, returned in **discon**, is discarded.
TNOTSUPPORT	Function not supported by the underlying transport provider.
TSYSERR	A system error occurred during function's execution.

DIAGNOSTICS

0	Successful completion.
-1	Failure; **t_errno** is set.

SEE ALSO

Library Functions – t_alloc(), t_connect(), t_listen(), t_open() t_snddis()

NAME

`t_rcvrel` – initiate an methodical release

SYNOPSIS

```
#include <tiuser.h>

int t_rcvrel(fd)
int fd;
```

DESCRIPTION

`t_rcvrel` acknowledges the receipt of an orderly release indication. **fd** identifies the local transport endpoint of the connection. Once `t_rcvrel` has returned, no further data may be received over the connection by the user. Any attempt will block forever. However, a user may continue to send data, assuming a methodical release indication has been sent.

`t_rcvrel` is an optional service of the transport provider. It is only supported if the transport provider returned service type `T_COTS_ORD` on `t_open` or `t_getinfo`

t_errno may set to:

`TBADF`	File descriptor does not refer to a transport endpoint.
`TNOREL`	No orderly release indication currently exists.
`TLOOK`	An asynchronous event occurred on transport endpoint; requires immediate attention.
`TNOTSUPPORT`	Not supported by the underlying transport provider.
`TSYSERR`	A system error occurred during execution.

DIAGNOSTICS

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - `t_open(), t_sndrel()`

NAME

t_rcvudata – receive a data unit

SYNOPSIS

```
#include <tiuser.h>

int t_rcvudata(fd, untdata, flags)
int fd;
struct t_unitdata *unitdata;
int flags;
```

DESCRIPTION

t_rcvudata receives a data unit from another transport user while in connectionless mode. **fd** identifies the local transport endpoint for data transmission. **unitdata** points to a **t_unitdata** structure containing:

```
struct  netbuf  addr;
struct   netbuf  opt;
struct  netbuf  udata;
```

netbuf is described in t_accept. The maxlen field of the netbuf structures addr, opt, and udata must be set before calling t_rcvudata. t_rcvudata returns the protocol address of the sending user is specified in addr; the protocol-specific options to be associated with this data unit are returned in opt. The user data received is returned in udata.

flags is set to indicate that the complete data unit was not received.

t_rcvudata operates in synchronous mode unless O_NDELAY is set via t_open or fcntl. In synchronous mode, t_rcvudata may wait for a data unit if none is immediately available. If O_NDELAY is set, t_rcvudata will execute asynchronously and fail immediately if there is no data unit available.

T_MORE will be set in **flags** if the current data unit is larger than the buffer defined in the udata field of **unitdata**. To retrieve the rest of the data unit another call to t_rcvudata must be made. Until the complete data unit is received all subsequent calls to t_rcvudata will return 0 in the length field of addr and opt.

t_errno may set to:

TBADF	File descriptor does not refer to a transport endpoint.
TNODATA	O_NDELAY was set, but no data units were available.
TBUFOVFLW	The allocated number of bytes for incoming protocol address is insufficient to store information. Information returned in **unitdata** is discarded.
TLOOK	An asynchronous event occurred on transport endpoint; requires immediate attention.
TNOTSUPPORT	Not supported by the underlying transport provider.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

0	Successful completion.
-1	Failure; **t_errno** is set.

SEE ALSO

Library Functions - `t_rcvuderr(), t_sndudata()`

NAME

`t_rcvuderr` – receive a data unit error indication

SYNOPSIS

```
#include <tiuser.h>

int t_rcvuderr(fd, uderr)
int fd;
struct t_uderr *uderr;
```

DESCRIPTION

`t_rcvuderr` returns information about an error on a previously sent data unit while in connectionless mode. `t_rcvuderr` should only be sent after a data unit error indication. **`fd`** identifies the local transport endpoint where the error report will be received. **`uderr`** points to a **`t_uderr`** structure containing:

```
struct  netbuf  addr;
struct   netbuf  opt;
long  error;
```

`netbuf` is described in `t_accept`. The `maxlen` field of the `netbuf` structures `addr`, and `opt` must be set before calling `t_rcvuderr`. `t_rcvuderr` returns the protocol address of the erroneous data unit in `addr`; the protocol-specific options to be associated with this data unit are returned in `opt`; the protocol-dependent error code is returned in `error`.

`t_rcvuderr` will clear any error indicationa and return without reporting the reason for the error if **`uderr`** is set to NULL.

`t_errno` may set to:

`TBADF`	File descriptor does not refer to a transport endpoint.
`TNOUDERR`	No error indication currently exists.
`TBUFOVFLW`	The allocated number of bytes for incoming protocol address or options is insufficient to store information. Information returned in **`uderr`** is discarded.

`TNOTSUPPORT`	Not supported by the underlying transport provider.
`TSYSERR`	A system error occurred during execution.

DIAGNOSTICS

0	Successful completion.
-1	Failure; **`t_errno`** is set.

SEE ALSO

Library Functions - `t_rcvudata()`, `t_sndudata()`

NAME

`tsearch, tfind,`
`tdelete, twalk` – manipulate binary search trees

SYNOPSIS

```
#include <search.h>

char *tsearch(key, rootp, compar)
char *key;
char **rootp;
int(*compar)();

char *tfind(key, rootp, compar)
char *key;
char **rootp;
int(*compar)();

char *tdelete(key, rootp, compar)
char *key;
char **rootp;
int(*compar)();

void twalk(root, action)
char *root;
void(*action)();
```

DESCRIPTION

`tsearch` accesses and builds a binary search tree. **`key`** points to a data item that is to be accessed or stored. **`rootp`** points to a variable that points to the root of the tree. **`compar`** points to a function which will be called to compare elements during the search. **`compar`** is invoked as follows:

```
compar(item1, item2)
itemtype *item1, *item2;
```

`compar` must return a number less than, equal to, or greater than zero, reflecting the relationship between `*item1` and `*item2`. If there is a data item in the tree equal to **`key`**, `tsearch` returns a pointer to that item. Otherwise, the data item pointed to by **`key`** is added to the tree and a pointer to it is returned.

`tfind` performs the same function as `tsearch` except that if the data item is not found, it is not added to the tree, and a NULL pointer is returned.

`tdelete` deletes a node from a binary search tree. If the deleted node is the root node of the binary search tree, the variable pointed to by **`rootp`** will be changed. `tdelete` returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

`twalk` traverses a binary search tree. **`Root`** is the root of the tree (or subtree) that is to be traversed. **`action`** points to a function that is to be invoked at each node during the traversal. **`action`** is invoked as follows:

```
action(node, order, level)
nodetype *node;
VISIT order;
int level;
```

The first argument (`node`) is the address of the `node` being traversed. The second argument (`order`) is a value from the `VISIT` enumeration data type, defined in **`search.h`** as follows:

```
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

depending on whether this is the first, second, or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a `leaf`. The third argument (`level`) is the level of the node in the tree, with the root being level zero.

DIAGNOSTICS

If a match is found, both `tsearch` and `tfind` return a pointer to the matched item. Otherwise, `tsearch` returns a pointer to the inserted item, and `tfind` returns NULL. `tsearch` returns a NULL pointer if there is not enough space available to create a new node. `tfind` and `tdelete` return a NULL pointer if **`rootp`** is NULL.

`tdelete` returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

NOTE

Severely bad things happen if the calling function alters the pointer to the root.

SEE ALSO

Library Functions - `bsearch()`, `hsearch()`, `lsearch()`, `qsort()`

NAME

`t_snd` – send data or expedited data

SYNOPSIS

```
#include <tiuser.h>

int t_snd(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;
```

DESCRIPTION

`t_snd` sends normal or expedited data. **`fd`** identifies the local transport endpoint. **`buf`** points to the user data; **`nbytes`** specifies the number of bytes of user data to be sent; **`flags`** specifies optional flags.

`t_snd` operates in synchronous mode unless `O_NDELAY` is set via `t_open` or `fcntl`. In synchronous mode, `t_snd` may wait if flow control restrictions prevent data acceptance by the local transport provider as the call is made. If `O_NDELAY` is set, `t_snd` will execute asynchronously and fail immediately if there are flow control restrictions. `t_snd` will wait if STREAMS internal resources are unavailable, even if `O_NDELAY` is set.

`t_snd` returns the number of bytes accepted by the transport provider. Normally this equals the number of bytes in **`nbytes`**. If `O_NDELAY` is set. There is a chance that only partial data will be accepted by the transport provider. `t_snd` will set `T_MORE` for the data that was sent and return less than **`nbytes`**. If **`nbytes`** is 0, no data will pass to the provider and `t_snd` will return 0.

If `T_EXPEDITED` is set in **`flags`**, the data will transmit as expedited data, subject to the transport provider's interpretation. If `T_MORE` is set in **`flags`**, or is set by `t_snd` for the data that was sent, an indication is passed to the transport provider that the transport service data unit (`TSDU`) or expedited transport service data unit (`ETSDU`) will arrive via multiple `t_snd` calls.

The use of `T_MORE` permits large logical data units to be broken up without a loss of boundaries at the receiving end.

T_MORE does not state how the data is packaged for transfer below the transport interface. The T_MORE flag has no meaning if the transport provider does not support TSDU as shown in the info argument on return from t_open or t_getinfo.

The size limits of the transport provider, as returned by t_open or t_getinfo, cannot be exceeded by TSDU or ETSDU. If exceeded, a TSYSERR with system error EPROTO will occur. Since EPROTO errors may not be reported immediately, the t_snd may not fail. But, a succeeding call that enters the transport endpoint will fail with the related TSYSERR.

When t_snd is issued from the T_IDLE state, the provider may discard the data silently. When t_snd is issued from any state other than T_DATAXFER, T_INREL, or T_IDLE, the provider will generate a TSYSERR with system error EPROTO.

t_errno may set to:

TBADF	File descriptor does not refer to a transport endpoint.
TFLOW	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data.
TNOTSUPPORT	Function not supported by the underlying transport provider.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

Positive value	Successful completion, the number of bytes accepted by the transport provider are returned.
-1	Failed transmission; **t_errno** is set.

SEE ALSO

Library Functions - t_open(), t_rcv()

NAME

t_snddis – send request for user-initiated disconnect

SYNOPSIS

```
#include <tiuser.h>

int t_snddis(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

t_snddis initiates an abortive release on an existing connection or it declines a connect request. **fd** identifies the connection's local transport endpoint; **call** is a pointer to a t_call structure that contains information about the abortive release. A t_call structure contains:

```
struct  netbuf  addr;
struct  netbuf  opt;
struct  netbuf  udata;
int  sequence
```

netbuf is described in t_accept. Depending on the context of the invocation of t_snddis, the semantics for the values in **call** differ. If t_snddis is rejecting a connect request, then call must be non-NULL and sequence must contain a value that uniquely identifies the rejected connect indication to the transport provider. In this case, the t_call fields addr and opt are ignored.

In any other case, **call** can be NULL unless data is to be sent with the disconnect request. If call is non-NULL, the t_call fields addr, opt, and sequence are ignored.

udata contains the user data to be sent to the remote user. The amount of user data must not exceed the limits of the transport provider. If len in udata is 0, no data will be sent.

t_errno may set to:

TBADF File descriptor does not refer to a transport endpoint.

TOUTSTATE	Function issued in the wrong sequence. If transport provider's outgoing queue is flushed, data may be lost.
TBADDATA	Amount of user data is not within the limits of the transport provider. If transport provider's outgoing queue is flushed, data may be lost.
TBADSEQ	Invalid sequence number specified, or a NULL **call** structure was passed when rejecting a connect request. Transport provider's outgoing queue is flushed, data is lost.
TLOOK	An asynchronous event has occurred on this transport endpoint requires immediate attention.
TNOTSUPPORT	Function not supported by the underlying transport provider.
TSYSERR	A system error occurred during function's execution.

DIAGNOSTICS

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions – t_connect(), t_getinfo(), t_listen(), t_open()

NAME

`t_sndrel` – initiate an methodical release

SYNOPSIS

```
#include <tiuser.h>

int t_sndrel(fd)
int fd;
```

DESCRIPTION

`t_sndrel` initiates an methodical release of a transport connection and tells the transport provider that no more data is expected from the transport user. **fd** identifies the local transport endpoint of the connection. Once `t_sndrel` has been issued, no further data may be sent over the connection by the user. However, a user may continue to receive data, assuming a methodical release indication has been received.

`t_sndrel` is an optional service of the transport provider. It is only supported if the transport provider returned service type `T_COTS_ORD` on `t_open` or `t_getinfo`

t_errno may set to:

TBADF	File descriptor does not refer to a transport endpoint.
TFLOW	`O_NDELAY` was set, but the flow control mechanism prevented the transport provider from accepting `t_sndrel`.
TNOTSUPPORT	Not supported by the underlying transport provider.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - `t_open()`, `t_rcvrel()`

NAME

t_sndudata – transmit a data unit

SYNOPSIS

```
#include <tiuser.h>

int t_sndudata(fd, untdata)
int fd;
struct t_unitdata *unitdata;
```

DESCRIPTION

t_sndudata transmits a data unit to another transport user while in connectionless mode. **fd** identifies the local transport endpoint for data transmission. **unitdata** points to **t_unitdata** structure containing:

```
struct  netbuf   addr;
struct  netbuf   opt;
struct  netbuf    udata;
```

netbuf is described in t_accept. The protocol address of the destination user is specified in addr. The protocol-specific options to be associated with this request are specified in opt. The user data to be sent is specified in udata. By setting the len field of opt to zero, the user informs the transport provider to use the default protocol options.

No data units will be passed to the transport provider if the len field of udata is zero.

t_sndudata operates in synchronous mode unless O_NDELAY is set via t_open or fcntl. In synchronous mode, t_sndudata may wait if flow control restrictions prevent data acceptance by the local transport provider at the time the call is made. If O_NDELAY is set, t_sndudata will execute asynchronously and fail immediately if there are flow control restrictions.

When t_sndudata is issued from an invalid state, or the amount of user data exceeds the TSDU size (as returned by t_open or t_getinfo), the provider will generate an EPROTO protocol error.

t_errno may set to:

TBADF	File descriptor does not refer to a transport endpoint.
TFLOW	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data.
TNOTSUPPORT	Not supported by the underlying transport provider.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - `t_rcvudata(), t_rcvuderr()`

NAME

`t_sync` – synchronize transport library

SYNOPSIS

```
#include <tiuser.h>

int t_sync(fd)
int fd;
```

DESCRIPTION

`t_sync` synchronizes the data structures managed by the transport library with information from the underlying transport provider. **fd** is the file descriptor for the transport endpoint whose data structures need to be managed. `t_sync` can convert a raw file descriptor to an initialized transport endpoint. This assumes that the file descriptor referenced a transport provider. `t_sync` also allows two cooperating processes to synchronize with a transport provider.

All users of a transport endpoint are treated as a single user. Multiple processes using the same endpoint should be coordinated so as not to violate the state of the provider. `t_sync` returns the current state of the provider to the user, enabling the user to verify the state. This coordination is only necessary among cooperating processes. A process or an incoming event could change the provider's state after a `t_sync` is issued.

If the provider is in the process of a state transition at the same time as `t_sync` is being called, `t_sync` will fail.

t_errno may set to:

TBADF	File descriptor is valid, but does not refer to a transport endpoint.
TSTATECHNG	The transport provider is undergoing a state transition.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

Upon successful completion, `t_sync` returns a state of the transport provider:

`T_UNBND`	Unbound.
`T_IDLE`	Idle.
`T_OUTCON`	Outgoing connection pending.
`T_INCON`	Incoming connection pending.
`T_DATAXFER`	Data transfer.
`T_OUTREL`	Outgoing orderly release (awaiting orderly release indication).
`T_INREL`	Incoming orderly release (awaiting orderly release request).
-1	Failure; **`t_errno`** is set.

SEE ALSO

System Calls - `dup(), execve(), fork(), open()`

NAME

`ttyconfig` – set terminal characteristics

SYNOPSIS

```
#include <ioctl.h>

struct sgttyb *ttyconfig(config, fd)
char *config;
int fd;
```

DESCRIPTION

`ttyconfig` provides a simplified means of configuring terminals, a process normally accomplished with several **ioctl** calls. Using a database similar to the `termcap` file, `ttyconfig` sets all parameters associated with the given configuration type, **config**, for the terminal associated with file descriptor **fd**. A pointer to the former value of the **sgttyb** structure is returned.

DIAGNOSTICS

If any **ioctl** calls fail, or if **config** is not found in `/etc/tconfig`, `ttyconfig` returns NULL.

NOTE

The value returned upon successful completion points to static data, overwritten by each call.

SEE ALSO

Utility Programs -	`stty`, `tset`
System Calls -	`ioctl()`
Library Functions -	`termcap()`
Device Facilities -	`tty`
File Formats -	`tconfig`

NAME

`t_unbind` – disable a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_unbind(fd)
int fd;
```

DESCRIPTION

t_unbind disables a transport endpoint. **fd** is the file descriptor for the transport endpoint that is to be unbound. After `t_unbind` completes the transport provider will not accept any more data or events destined for this transport endpoint.

t_errno may be set to:

TBADF	File descriptor does not refer to a transport endpoint.
TLOOK	An asynchronous event occurred on this transport endpoint.
TOUTSTATE	Function issued in wrong sequence.
TSYSERR	A system error occurred during execution.

DIAGNOSTICS

The following values are returned:

0	Successful completion.
-1	System call failure; **t_errno** is set.

SEE ALSO

Library Functions - `t_bind()`

NAME

`tzset` – set time zone

SYNOPSIS

```
#include <time.h>

void tzset();
```

DESCRIPTION

The `tzset()` function uses the value of the environment variable `TZ` to set time conversion information used by `localtime()`, `ctime()`, `strftime()`, and `mktime()`. If `TZ` is absent from the environment, implementation-defined default time-zone information shall be used.

The `tzset()` function shall set the external variable `tzname`:

```
extern char *tzname[2] = {"std", "dst"};
```

COPYRIGHT NOTICE

SEE ALSO

Library Functions - `ctime()`, `getenv()`, `localtime()`, `mktime()`, `strftime()`

NAME

`ungetc`,
`unlocked_ungetc` – push a character back into the input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc(c, stream)
int c;
FILE *stream;

int unlocked_ungetc(c, stream)
int c;
FILE *stream;
```

DESCRIPTION

`ungetc` inserts the character **c** into the buffer associated with an input stream. The character will be returned by the next read from that stream. `ungetc` returns **c**.

The character **c** can later be read by `getc`, `fread`, `gets`, `fgets`, `fgetc`, `fscanf`, and `scanf`.

One character pushback is guaranteed provided that something has been read from the stream prior to the `ungetc` call.

If **c** is `EOF`, `ungetc` does nothing and returns `EOF`. `fseek` erases all memory of inserted characters.

The unlocked variant of this function is identical to its thread-safe brethren but performs no locking. If the application wishes to do its own locking, it can do so by using `flockfile` and `funlockfile`. In that case, the unlocked version will be somewhat faster.

DIAGNOSTICS

A read of some kind must be performed prior to the `ungetc`.

An `EOF` is returned if `ungetc` cannot insert the character.

SEE ALSO

Library Functions - `flockfile( )`, `funlockfile( )`, `fseek()`, `getc()`, `setbuf()`

NAME

`utime` – set file times files, times

SYNOPSIS

```
#include <types.h>

utime(name, newtime)
char *name;
time_t newtime[2];
```

DESCRIPTION

`utime` sets the access and modify times for the file **name** to the values given in the **newtime** array (in that order).

DIAGNOSTICS

`utime` returns 0 if successful, -1 if not.

SEE ALSO

Utility Programs - `ls`

System Calls - `utimes()`, `stat()`

NAME

`vprintf`, `vfprintf`,
`vsprintf` – print formatted output of `varargs` argument list

SYNOPSIS

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

int vprintf(format, ap)
char *format;
va_list ap;

int vfprintf(stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf(s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

`vprintf`, `vfprintf` and `vsprintf` are the same as `printf`, `fprintf`, and `sprintf`, respectively, except that instead of being called with a variable number of arguments, they are called with a variable argument list as defined in **`varargs.h`**.

SEE ALSO

Library Functions - `printf()`

NAME

`xdr` – library routines for external data representation

SYNOPSIS AND DESCRIPTION

These routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls are transmitted using these routines.

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
XDR *xdrs;
char **arrp;
u_int *sizep, maxsize, elsize;
xdrproc_t elproc;
```

A filter primitive that translates between variable-length arrays and their corresponding external representations. The parameter `arrp` is the address of the pointer to the array, while `sizep` is the address of the element count of the array; this element count cannot exceed `maxsize`. The parameter `elsize` is the size of each of the array's elements, and `elproc` is an `XDR` filter that translates between the array elements' C form, and their external representation. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_bool(xdrs, bp)
XDR *xdrs;
bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either 1 or 0. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_bytes(xdrs, sp, sizep, maxsize)
XDR *xdrs;
char **sp;
u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter `sp` is the address of the string pointer. The length of the string is located at address `sizep`; strings cannot be longer than `maxsize`. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_char(xdrs, cp)
XDR *xdrs;
char *cp;
```

A filter primitive that translates between C characters and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

NOTE

Encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider `xdr_bytes()`, `xdr_opaque()`, or `xdr_string()`.

```
void
xdr_destroy(xdrs)
XDR *xdrs;
```

A macro that invokes the destroy routine associated with the `XDR` stream, `xdrs`. Destruction usually involves freeing private data structures associated with the stream. Using `xdrs` after invoking `xdr_destroy()` is undefined.

```
xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;
```

A filter primitive that translates between C `double` precision numbers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;
```

A filter primitive that translates between C `enums` (actually integers) and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_float(xdrs, fp)
XDR *xdrs;
float *fp;
```

A filter primitive that translates between C `floats` and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

```
void
xdr_free(proc, objp)
xdrproc_t proc;
char *objp;
```

Generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself.

NOTE

The pointer passed to this routine is not freed, but what it points to is freed (recursively).

```
u_int
xdr_getpos(xdrs)
XDR *xdrs;
```

A macro that invokes the get-position routine associated with the XDR stream, `xdrs`. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

```
long *
xdr_inline(xdrs, len)
XDR *xdrs;
int len;
```

A macro that invokes the in-line routine associated with the XDR stream, `xdrs`. The routine returns a pointer to a contiguous piece of the stream's buffer; `len` is the byte length of the desired buffer.

NOTE

Pointer is cast to `long *`.

WARNING

`xdr_inline()` may return NULL (0) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

```
xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;
```

A filter primitive that translates between C long integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

```
void
xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
u_int size;
enum xdr_op op;
```

This routine initializes the `XDR` stream object pointed to by `xdrs`. The stream's data is written to, or read from, a chunk of memory at location `addr` whose length is no more than `size` bytes long. The `op` determines the direction of the `XDR` stream (either `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`).

```
xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
u_int cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter `cp` is the address of the opaque object, and `cnt` is its size in bytes. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_pointer(xdrs, objpp, objsize, xdrobj)
XDR *xdrs;
char **objpp;
u_int objsize;
xdrproc_t xdrobj;
```

Like `xdr_reference()` except that it serializes NULL pointers, whereas `xdr_reference()` does not. Thus, `xdr_pointer()` can represent recursive data structures, such as binary trees or linked lists.

```
void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit,
writeit)
XDR *xdrs;
u_int sendsize, recvsize;
char *handle;
int (*readit) (), (*writeit) ();
```

This routine initializes the `XDR` stream object pointed to by `xdrs`. The stream's data is written to a buffer of size `sendsize`; a value of 0 indicates the system should use a suitable default. The stream's data is read from a buffer of size `recvsize`; it too can be set to a suitable default by passing a 0 value. When a stream's output buffer is full, `writeit` is called. Similarly, when a stream's input buffer is empty, `readit` is called. The behavior of these two routines is similar to the system calls `read` and `write`, except that handle is passed to the former routines as the first parameter.

NOTE

The `XDR` stream's `op` field must be set by the caller.

WARNING

This `XDR` stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

```
xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;
```

This routine can be invoked only on streams created by `xdrrec_create()`. The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if `sendnow` is non-zero. This routine returns 1 if it succeeds, 0 otherwise.

```
xdrrec_eof(xdrs)
XDR *xdrs;
```

This routine can be invoked only on streams created by `xdrrec_create()`. After consuming the rest of the current record in the stream, this routine returns 1 if the stream has no more input, 0 otherwise.

```
xdrrec_skiprecord(xdrs)
XDR *xdrs;
```

This routine can be invoked only on streams created by `xdrrec_create()`. It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter `pp` is the address of the pointer; `size` is the size of the structure that `*pp` points to; and `proc` is an XDR procedure that filters the structure between its C form and its external representation. This routine returns 1 if it succeeds, 0 otherwise.

WARNING

This routine does not understand NULL pointers. Use `xdr_pointer()` instead.

```
xdr_setpos(xdrs, pos)
XDR *xdrs;
u_int pos;
```

A macro that invokes the set position routine associated with the XDR stream xdrs. The parameter pos is a position value obtained from xdr_getpos(). This routine returns 1 if the XDR stream could be repositioned, and 0 otherwise.

WARNING

It is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

```
xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;
```

A filter primitive that translates between C short integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

```
void
xdrstdio_create(xdrs, file, op)
XDR *xdrs;
FILE *file;
enum xdr_op op;
```

This routine initializes the XDR stream object pointed to by xdrs. The XDR stream data is written to, or read from, the Standard I/O stream file. The parameter op determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

WARNING

The destroy routine associated with such XDR streams calls fflush() on the file stream, but never fclose().

```
xdr_string(xdrs, sp, maxsize)
XDR *xdrs;
char **sp;
u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than maxsize.

NOTE

`sp` is the address of the string's pointer. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_u_char(xdrs, ucp)
XDR *xdrs;
unsigned char *ucp;
```

A filter primitive that translates between unsigned C characters and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_u_int(xdrs, up)
XDR *xdrs;
unsigned *up;
```

A filter primitive that translates between C unsigned integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_u_long(xdrs, ulp)
XDR *xdrs;
unsigned long *ulp;
```

A filter primitive that translates between C unsigned long integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_u_short(xdrs, usp)
XDR *xdrs;
unsigned short *usp;
```

A filter primitive that translates between C unsigned short integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
int *dscmp;
char *unp;
struct xdr_discrim *choices;
```

bool_t (*defaultarm) (); /* may equal NULL */

A filter primitive that translates between a discriminated C union and its corresponding external representation. It first translates the discriminant of the union located at `dscmp`. This discriminant is always an `enum_t`. Next the union located at unp is translated. The parameter choices is a pointer to an array of `xdr_discrim()` structures. Each structure contains an ordered pair of [`value`, `proc`]. If the union's discriminant is equal to the associated value, then the `proc` is called to translate the union. The end of the `xdr_discrim()` structure array is denoted by a routine of value NULL. If the discriminant is not found in the choices array, then the defaultarm procedure is called (if it is not NULL). returns 1 if it succeeds, 0 otherwise.

```
xdr_vector(xdrs, arrp, size, elsize, elproc)
XDR *xdrs;
char *arrp;
u_int size, elsize;
xdrproc_t elproc;
```

A filter primitive that translates between fixed-length arrays and their corresponding external representations. The parameter `arrp` is the address of the pointer to the array, while `size` is the element count of the array. The parameter `elsize` is the size of each of the array's elements, and `elproc` is an `XDR` filter that translates between the array elements' C form, and their external representation. This routine returns 1 if it succeeds, 0 otherwise.

xdr_void()

This routine always returns 1. It may be passed to `RPC` routines that require a function parameter, where nothing is to be done.

```
xdr_wrapstring(xdrs, sp)
XDR *xdrs;
char **sp;
```

A primitive that calls `xdr_string` (`xdrs`, `sp`, `MAXUN.UNSIGNED`); where `MAXUN.UNSIGNED` is the maximum value of an unsigned integer. `xdr_wrapstring()` is handy because the `RPC` package passes a maximum of two `XDR` routines as parameters, and `xdr_string()`, one of the most frequently used primitives, requires three. Returns 1 if it succeeds, 0 otherwise.

SEE ALSO

Library Function - `rpc()`

Documents: *External Data Representation Standard: Protocol Specification*

External Data Representation: Sun Technical Notes

XDR: External Data Representation Standard, RFC1014, Sun Microsystems, Inc., USC-ISI.

NAME

`xstring: xtrcpy,`
`xtrncpy, xtrcat,`
`basename, isdotdir` – extensions to string library

SYNOPSIS

```
#include <string.h>

char *xtrcpy(d, s)
char *d, *s;

char *xtrncpy(d, s, n)
char *d, *s; int n;

char *xtrcat(d, s)
char *d, *s;

char *basename(s)
char *s;

isdotdir(s)
char *s;
```

DESCRIPTION

The `xstring` routines provide some extra functionality beyond that available with the `string` routines. The functions `xtrcpy`, `xtrncpy`, and `xtrcat` work just like `strcpy`, `strncpy`, and `strcat`, except instead of returning a pointer to the beginning of the destination string they return a pointer to the last byte copied. Thus a string can be built up from several strings by composing these functions appropriately:

```
xtrcpy(xtrcpy(xtrcpy(buffer, dirs), "/"), basename);
```

This method is much faster than a comparable sequence of `strcpy` and `strcat`.

The functions `basename` and `isdotdir` perform common operations on file names. `basename` returns a pointer to the first character after the last "`/`" in the argument string. If no "`/`" is found, `basename` returns its argument. `isdotdir` returns 1 if the argument string is either "`.`" or "`..`", 0 otherwise. It is much faster than a pair of `strcmp` calls.

SEE ALSO

Library Functions - `string(), bstring(), ctype()`

Master Index

N

O

P

T

U

V

W

X-Z