man pages section 3: Basic Library Functions

Beta

Part No: 821–1465–06
November 2010

ORACLE

110425@25097

Contents

Preface

Both novice users and those familar with the SunOS operating system can use online man pages to obtain information about the system and its features. A man page is intended to answer concisely the question "What does it do?" The man pages in general comprise a reference manual. They are not intended to be a tutorial.

Overview

The following contains a brief description of each man page section and the information it references:

- Section 1 describes, in alphabetical order, commands available with the operating system.
- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.
- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.
- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5 contains miscellaneous documentation such as character-set tables.
- Section 7 describes various special files that refer to specific hardware peripherals and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.
- Section 9E describes the DDI (Device Driver Interface)/DKI (Driver/Kernel Interface), DDI-only, and DKI-only entry-point routines a developer can include in a device driver.
- Section 9F describes the kernel functions available for use by device drivers.
- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report,

there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man(1)` for more information about man pages in general.

NAME — This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS — This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full path name is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

- [ ] — Brackets. The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.
- . . . — Ellipses. Several values can be provided for the previous argument, or the previous argument can be specified multiple times, for example, "`filename...`".
- | — Separator. Only one of the arguments separated by this character can be specified at a time.
- { } — Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.

PROTOCOL — This section occurs only in subsection 3R to indicate the protocol description file.

DESCRIPTION — This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES. Interactive commands, subcommands, requests, macros, and functions are described under USAGE.

IOCTL — This section appears on pages in Section 7 only. Only the device class that supplies appropriate parameters to the `ioctl(2)` system call is called `ioctl` and generates its own heading. `ioctl` calls for a specific device are listed alphabetically (on the man page for that specific device).

	`ioctl` calls are used for a particular class of devices all of which have an `io` ending, such as `mtio(7I)`.
OPTIONS	This secton lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output – standard output, standard error, or output files – generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.
USAGE	This section lists special rules, features, and commands that require in-depth explanations. The subsections listed here are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar
EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete

	example including command-line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%`, or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS, and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion, and values other than zero for various error conditions.
FILES	This section lists all file names referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
ATTRIBUTES	This section lists characteristics of commands, utilities, and device drivers by defining the attribute type and its corresponding value. See `attributes(5)` for more information.
SEE ALSO	This section lists references to other man pages, in-house documentation, and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.
BUGS	This section describes known bugs and, wherever possible, suggests workarounds.

REFERENCE

Basic Library Functions

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

Synopsis

```
#include <stdlib.h>

long a64l(const char *s);

char *l64a(long l);
```

Description These functions maintain numbers stored in base-64 ASCII characters that define a notation by which long integers can be represented by up to six characters. Each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are as follows:

Character	Digit
.	0
/	1
0-9	2-11
A-Z	12-37
a-z	38-63

The `a64l()` function takes a pointer to a null-terminated base-64 representation and returns a corresponding `long` value. If the string pointed to by *s* contains more than six characters, `a64l()` uses the first six.

The `a64l()` function scans the character string from left to right with the least significant digit on the left, decoding each character as a 6-bit radix-64 number.

The `l64a()` function takes a `long` argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, `l64a()` returns a pointer to a null string.

The value returned by `l64a()` is a pointer into a static buffer, the contents of which are overwritten by each call. In the case of multithreaded applications, the return value is a pointer to thread specific data.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also attributes(5), standards(5)

Name abort – terminate the process abnormally

Synopsis

```
#include <stdlib.h>

void abort(void);
```

Description The `abort()` function causes abnormal process termination to occur, unless the signal `SIGABRT` is being caught and the signal handler does not return. The abnormal termination processing includes at least the effect of `fclose(3C)` on all open streams and message catalogue descriptors, and the default actions defined for `SIGABRT`. The `SIGABRT` signal is sent to the calling process as if by means of the `raise(3C)` function with the argument `SIGABRT`.

The status made available to `wait(3C)` or `waitpid(3C)` by `abort` will be that of a process terminated by the `SIGABRT` signal. `abort` will override blocking or ignoring the `SIGABRT` signal.

Return Values The `abort()` function does not return.

Errors No errors are defined.

Usage Catching the signal is intended to provide the application writer with a portable means to abort processing, free from possible interference from any implementation-provided library functions. If `SIGABRT` is neither caught nor ignored, and the current directory is writable, a core dump may be produced.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `exit(2)`, `getrlimit(2)`, `kill(2)`, `fclose(3C)`, `raise(3C)`, `signal(3C)`, `wait(3C)`, `waitpid(3C)`, `attributes(5)`, `standards(5)`

Name abs, labs, llabs – return absolute value of integer

Synopsis

```
#include <stdlib.h>

int abs(int val);

long labs(long lval);

long long llabs(long long llval);
```

Description The `abs()` function returns the absolute value of its `int` operand.

The `labs()` function returns the absolute value of its `long` operand.

The `llabs()` function returns the absolute value of its `long long` operand.

Usage In 2's-complement representation, the absolute value of the largest magnitude negative integral value is undefined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `attributes(5)`, `standards(5)`

Name addsev – define additional severities

Synopsis

```
#include <pfmt.h>

int addsev(int int_val, const char *string);
```

Description The `addsev()` function defines additional severities for use in subsequent calls to `pfmt`(3C) or `lfmt`(3C). It associates an integer value *int_val* in the range [5-255] with a character *string*, overwriting any previous string association between *int_val* and *string*.

If *int_val* is OR-ed with the *flags* argument passed to subsequent calls to `pfmt()` or `lfmt()`, *string* will be used as severity. Passing a null *string* removes the severity.

Return Values Upon successful completion, `addsev()` returns `0`. Otherwise it returns–`1`.

Usage Only the standard severities are automatically displayed for the locale in effect at runtime. An application must provide the means for displaying locale-specific versions of add-on severities. Add-on severities are only effective within the applications defining them.

Examples **EXAMPLE 1** Example of `addsev()` function.

The following example

```
#define Panic 5
setlabel("APPL");
setcat("my_appl");
addsev(Panic, gettxt(":26", "PANIC"));
/* . . . */
lfmt(stderr, MM_SOFT|MM_APPL|PANIC, ":12:Cannot locate database\n");
```

will display the message to *stderr* and forward to the logging service

```
APPL: PANIC: Cannot locate database
```

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-safe

See Also `gettxt`(3C), `lfmt`(3C), `pfmt`(3C), `attributes`(5)

Name addseverity – build a list of severity levels for an application for use with fmtmsg

Synopsis

```
#include <fmtmsg.h>

int addseverity(int severity, const char *string);
```

Description The `addseverity()` function builds a list of severity levels for an application to be used with the message formatting facility `fmtmsg()`. The *severity* argument is an integer value indicating the seriousness of the condition. The *string* argument is a pointer to a string describing the condition (string is not limited to a specific size).

If `addseverity()` is called with an integer value that has not been previously defined, the function adds that new severity value and print string to the existing set of standard severity levels.

If `addseverity()` is called with an integer value that has been previously defined, the function redefines that value with the new print string. Previously defined severity levels may be removed by supplying the null string. If `addseverity()` is called with a negative number or an integer value of 0, 1, 2, 3, or 4, the function fails and returns –1. The values 0–4 are reserved for the standard severity levels and cannot be modified. Identifiers for the standard levels of severity are:

`MM_HALT`	Indicates that the application has encountered a severe fault and is halting. Produces the print string `HALT`.
`MM_ERROR`	Indicates that the application has detected a fault. Produces the print string `ERROR`.
`MM_WARNING`	Indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the print string `WARNING`.
`MM_INFO`	Provides information about a condition that is not in error. Produces the print string `INFO`.
`MM_NOSEV`	Indicates that no severity level is supplied for the message.

Severity levels may also be defined at run time using the `SEV_LEVEL` environment variable (see `fmtmsg(3C)`).

Return Values Upon successful completion, `addseverity()` returns `MM_OK`. Otherwise it returns `MM_NOTOK`.

Examples **EXAMPLE 1** Example of `addseverity()` function.

When the function call

```
addseverity(7,"ALERT")
```

is followed by the call

```
fmtmsg(MM_PRINT, "UX:cat", 7, "invalid syntax", "refer to manual",
"UX:cat:001")
```

EXAMPLE 1 Example of `addseverity()` function. *(Continued)*

the resulting output is

```
UX:cat: ALERT: invalid syntax
TO FIX: refer to manual   UX:cat:001
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also

`fmtmsg(1)`, `fmtmsg(3C)`, `gettxt(3C)`, `printf(3C)`, `attributes(5)`

Name aio_cancel – cancel asynchronous I/O request

Synopsis

```
#include <aio.h>

int aio_cancel(int fildes, struct aiocb *aiocbp);
```

Description The `aio_cancel()` function attempts to cancel one or more asynchronous I/O requests currently outstanding against file descriptor *fildes*. The *aiocbp* argument points to the asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is `NULL`, then all outstanding cancelable asynchronous I/O requests against *fildes* are canceled.

Normal asynchronous notification occurs for asynchronous I/O operations that are successfully canceled. If there are requests that cannot be canceled, then the normal asynchronous completion process takes place for those requests when they are completed.

For requested operations that are successfully canceled, the associated error status is set to `ECANCELED` and the return status is −1. For requested operations that are not successfully canceled, the *aiocbp* is not modified by `aio_cancel()`.

If *aiocbp* is not `NULL`, then if *fildes* does not have the same value as the file descriptor with which the asynchronous operation was initiated, unspecified results occur.

Return Values The `aio_cancel()` function returns the value `AIO_CANCELED` to the calling process if the requested operation(s) were canceled. The value `AIO_NOTCANCELED` is returned if at least one of the requested operation(s) cannot be canceled because it is in progress. In this case, the state of the other operations, if any, referenced in the call to `aio_cancel()` is not indicated by the return value of `aio_cancel()`. The application may determine the state of affairs for these operations by using `aio_error(3C)`. The value `AIO_ALLDONE` is returned if all of the operations have already completed. Otherwise, the function returns −1 and sets `errno` to indicate the error.

Errors The `aio_cancel()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

`ENOSYS` The `aio_cancel()` function is not supported.

Usage The `aio_cancel()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also aio.h(3HEAD), signal.h(3HEAD), aio_read(3C), aio_return(3C), attributes(5), lf64(5), standards(5)

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set errno to ENOSYS.

Name aiocancel – cancel an asynchronous operation

Synopsis

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

int aiocancel(aio_result_t *resultp);
```

Description `aiocancel()` cancels the asynchronous operation associated with the result buffer pointed to by *resultp*. It may not be possible to immediately cancel an operation which is in progress and in this case, `aiocancel()` will not wait to cancel it.

Upon successful completion, `aiocancel()` returns `0` and the requested operation is cancelled. The application will not receive the `SIGIO` completion signal for an asynchronous operation that is successfully cancelled.

Return Values Upon successful completion, `aiocancel()` returns `0`. Upon failure, `aiocancel()` returns `−1` and sets `errno` to indicate the error.

Errors `aiocancel()` will fail if any of the following are true:

`EACCES` The parameter *resultp* does not correspond to any outstanding asynchronous operation, although there is at least one currently outstanding.

`EFAULT` *resultp* points to an address outside the address space of the requesting process. See `NOTES`.

`EINVAL` There are not any outstanding requests to cancel.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `aioread(3C)`, `aiowait(3C)`, `attributes(5)`

Notes Passing an illegal address as *resultp* will result in setting `errno` to `EFAULT` *only* if it is detected by the application process.

Name aio_error – retrieve errors status for an asynchronous I/O operation

Synopsis

```
#include <aio.h>

int aio_error(const struct aiocb *aiocbp);
```

Description The `aio_error()` function returns the error status associated with the `aiocb` structure referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the `errno` value that would be set by the corresponding `read(2)`, `write(2)`, or `fsync(3C)` operation. If the operation has not yet completed, then the error status will be equal to `EINPROGRESS`.

Return Values If the asynchronous I/O operation has completed successfully, then `0` is returned. If the asynchronous operation has completed unsuccessfully, then the error status, as described for `read(2)`, `write(2)`, and `fsync(3C)`, is returned. If the asynchronous I/O operation has not yet completed, then `EINPROGRESS` is returned.

Errors The `aio_error()` function may fail if:

`EINVAL` The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

Usage The `aio_error()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Examples **EXAMPLE 1** The following is an example of an error handling routine using the `aio_error()` function.

```
#include <aio.h>
#include <errno.h>
#include <signal.h>
struct aiocb      my_aiocb;
struct sigaction  my_sigaction;
void              my_aio_handler(int, siginfo_t *, void *);
 . . .
my_sigaction.sa_flags = SA_SIGINFO;
my_sigaction.sa_sigaction = my_aio_handler;
sigemptyset(&my_sigaction.sa_mask);
(void) sigaction(SIGRTMIN, &my_sigaction, NULL);
 . . .
my_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
my_aiocb.aio_sigevent.sigev_signo = SIGRTMIN;
my_aiocb.aio_sigevent.sigev_value.sival_ptr = &myaiocb;
 . . .
(void) aio_read(&my_aiocb);
 . . .
void
my_aio_handler(int signo, siginfo_t *siginfo, void *context) {
int    my_errno;
struct aiocb    *my_aiocbp;
```

EXAMPLE 1 The following is an example of an error handling routine using the `aio_error()` function. *(Continued)*

```
my_aiocbp = siginfo->si_value.sival_ptr;
    if ((my_errno = aio_error(my_aiocb)) != EINPROGRESS) {
        int my_status = aio_return(my_aiocb);
            if (my_status >= 0){ /* start another operation */
                     . . .
            } else        { /* handle I/O error */
                      . . .
            }
    }
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also

`_Exit(2)`, `close(2)`, `fork(2)`, `lseek(2)`, `read(2)`, `write(2)`, `aio.h(3HEAD)`, `aio_cancel(3C)`, `aio_fsync(3C)`, `aio_read(3C)`, `aio_return(3C)`, `aio_write(3C)`, `lio_listio(3C)`, `signal.h(3HEAD)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Name aio_fsync – asynchronous file synchronization

Synopsis

```
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocbp);
```

Description The `aio_fsync()` function asynchronously forces all I/O operations associated with the file indicated by the file descriptor `aio_fildes` member of the `aiocb` structure referenced by the *aiocbp* argument and queued at the time of the call to `aio_fsync()` to the synchronized I/O completion state. The function call returns when the synchronization request has been initiated or queued to the file or device (even when the data cannot be synchronized immediately).

If *op* is `O_DSYNC`, all currently queued I/O operations are completed as if by a call to `fdatasync(3C)`; that is, as defined for synchronized I/O data integrity completion. If *op* is `O_SYNC`, all currently queued I/O operations are completed as if by a call to `fsync(3C)`; that is, as defined for synchronized I/O file integrity completion. If the `aio_fsync()` function fails, or if the operation queued by `aio_fsync()` fails, then, as for `fsync(3C)` and `fdatasync(3C)`, outstanding I/O operations are not guaranteed to have been completed.

If `aio_fsync()` succeeds, then it is only the I/O that was queued at the time of the call to `aio_fsync()` that is guaranteed to be forced to the relevant completion state. The completion of subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized fashion.

The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used as an argument to `aio_error(3C)` and `aio_return(3C)` in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. When the request is queued, the error status for the operation is `EINPROGRESS`. When all data has been successfully transferred, the error status will be reset to reflect the success or failure of the operation. If the operation does not complete successfully, the error status for the operation will be set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to occur when all operations have achieved synchronized I/O completion (see `signal.h(3HEAD)`). All other members of the structure referenced by *aiocbp* are ignored. If the control block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

If the `aio_fsync()` function fails or the *aiocbp* indicates an error condition, data is not guaranteed to have been successfully transferred.

If *aiocbp* is `NULL`, then no status is returned in *aiocbp*, and no signal is generated upon completion of the operation.

Return Values The `aio_fsync()` function returns `0` to the calling process if the I/O operation is successfully queued; otherwise, the function returns `−1` and sets `errno` to indicate the error.

Errors The `aio_fsync()` function will fail if:

`EAGAIN` The requested asynchronous operation was not queued due to temporary resource limitations.

`EBADF` The `aio_fildes` member of the `aiocb` structure referenced by the *aiocbp* argument is not a valid file descriptor open for writing.

`EINVAL` The system does not support synchronized I/O for this file.

`EINVAL` A value of *op* other than `O_DSYNC` or `O_SYNC` was specified.

In the event that any of the queued I/O operations fail, `aio_fsync()` returns the error condition defined for `read(2)` and `write(2)`. The error will be returned in the error status for the asynchronous `fsync(3C)` operation, which can be retrieved using `aio_error(3C)`.

Usage The `aio_fsync()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `fcntl(2)`, `open(2)`, `read(2)`, `write(2)`, `aio_error(3C)`, `aio_return(3C)`, `aio.h(3HEAD)`, `fcntl.h(3HEAD)`, `fdatasync(3C)`, `fsync(3C)`, `signal.h(3HEAD)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Name aio_read – asynchronous read from a file

Synopsis

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

Description The `aio_read()` function allows the calling process to read *aiocbp*->`aio_nbytes` from the file associated with *aiocbp*->`aio_fildes` into the buffer pointed to by *aiocbp*->`aio_buf`. The function call returns when the read request has been initiated or queued to the file or device (even when the data cannot be delivered immediately). If `_POSIX_PRIORITIZED_IO` is defined and prioritized I/O is supported for this file, then the asynchronous operation is submitted at a priority equal to the scheduling priority of the process minus *aiocbp*->`aio_reqprio`. The *aiocbp* value may be used as an argument to `aio_error(3C)` and `aio_return(3C)` in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. If an error condition is encountered during queuing, the function call returns without having initiated or queued the request. The requested operation takes place at the absolute position in the file as given by *aio_offset*, as if `lseek(2)` were called immediately prior to the operation with an *offset* equal to *aio_offset* and a `whence` equal to `SEEK_SET`. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aiocbp*->`aio_sigevent` structure defines what asynchronous notification is to occur when the asynchronous operation completes, as specified in `signal.h(3HEAD)`.

The *aiocbp*->`aio_lio_opcode` field is ignored by `aio_read()`.

The *aiocbp* argument points to an `aiocb` structure. If the buffer pointed to by *aiocbp*->`aio_buf` or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

If `_POSIX_SYNCHRONIZED_IO` is defined and synchronized I/O is enabled on the file associated with *aiocbp*->`aio_fildes`, the behavior of this function is according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding to the address range being changed, the result of that action is undefined.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *aiocbp*->`aio_fildes`.

Return Values The `aio_read()` function returns `0` to the calling process if the I/O operation is successfully queued; otherwise, the function returns `−1` and sets `errno` to indicate the error.

Errors The `aio_read()` function will fail if:

`EAGAIN` The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to `aio_read()`, or asynchronously. If any of the conditions below are detected synchronously, the `aio_read()` function returns –1 and sets `errno` to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to –1, and the error status of the asynchronous operation will be set to the corresponding value.

`EBADF` The *aiocbp*->`aio_fildes` argument is not a valid file descriptor open for reading.

`EINVAL` The file offset value implied by *aiocbp*->`aio_offset` would be invalid, *aiocbp*->`aio_reqprio` is not a valid value, or *aiocbp*->`aio_nbytes` is an invalid value.

In the case that the `aio_read()` successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values normally returned by the `read(2)` function call. In addition, the error status of the asynchronous operation will be set to one of the error statuses normally set by the `read()` function call, or one of the following values:

`EBADF` The *aiocbp*->`aio_fildes` argument is not a valid file descriptor open for reading.

`ECANCELED` The requested I/O was canceled before the I/O completed due to an explicit `aio_cancel(3C)` request.

`EINVAL` The file offset value implied by *aiocbp*->`aio_offset` would be invalid.

The following condition may be detected synchronously or asynchronously:

`EOVERFLOW` The file is a regular file, *aiobcp*->`aio_nbytes` is greater than 0 and the starting offset in *aiobcp*->`aio_offset` is before the end-of-file and is at or beyond the offset maximum in the open file description associated with *aiocbp*->`aio_fildes`.

Usage For portability, the application should set *aiocb*->`aio_reqprio` to `0`.

The `aio_read()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Standard	See standards(5).

See Also close(2), exec(2), exit(2), fork(2), lseek(2), read(2), write(2), aio_cancel(3C), aio_return(3C), aio.h(3HEAD), lio_listio(3C), siginfo.h(3HEAD), signal.h(3HEAD), attributes(5), lf64(5), standards(5)

Name aioread, aiowrite – read or write asynchronous I/O operations

Synopsis

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

int aioread(int fildes, char *bufp, int bufs, off_t offset,
     int whence, aio_result_t *resultp);

int aiowrite(int fildes, const char *bufp, int bufs, off_t offset,
     int whence, aio_result_t *resultp);
```

Description The `aioread()` function initiates one asynchronous `read(2)` and returns control to the calling program. The read continues concurrently with other activity of the process. An attempt is made to read *bufs* bytes of data from the object referenced by the descriptor *fildes* into the buffer pointed to by *bufp*.

The `aiowrite()` function initiates one asynchronous `write(2)` and returns control to the calling program. The write continues concurrently with other activity of the process. An attempt is made to write *bufs* bytes of data from the buffer pointed to by *bufp* to the object referenced by the descriptor *fildes*.

On objects capable of seeking, the I/O operation starts at the position specified by *whence* and *offset*. These parameters have the same meaning as the corresponding parameters to the `llseek(2)` function. On objects not capable of seeking the I/O operation always start from the current position and the parameters `whence` and *offset* are ignored. The seek pointer for objects capable of seeking is not updated by `aioread()` or `aiowrite()`. Sequential asynchronous operations on these devices must be managed by the application using the *whence* and *offset* parameters.

The result of the asynchronous operation is stored in the structure pointed to by *resultp*:

```
int aio_return;          /* return value of read() or write() */
int aio_errno;          /* value of errno for read() or write() */
```

Upon completion of the operation both `aio_return` and `aio_errno` are set to reflect the result of the operation. Since `AIO_INPROGRESS` is not a value used by the system, the client can detect a change in state by initializing `aio_return` to this value.

The application-supplied buffer *bufp* should not be referenced by the application until after the operation has completed. While the operation is in progress, this buffer is in use by the operating system.

Notification of the completion of an asynchronous I/O operation can be obtained synchronously through the `aiowait(3C)` function, or asynchronously by installing a signal handler for the `SIGIO` signal. Asynchronous notification is accomplished by sending the process a `SIGIO` signal. If a signal handler is not installed for the `SIGIO` signal, asynchronous notification is disabled. The delivery of this instance of the `SIGIO` signal is reliable in that a signal delivered while the handler is executing is not lost. If the client ensures that `aiowait()`

returns nothing (using a polling timeout) before returning from the signal handler, no asynchronous I/O notifications are lost. The `aiowait()` function is the only way to dequeue an asynchronous notification. The `SIGIO` signal can have several meanings simultaneously. For example, it can signify that a descriptor generated `SIGIO` and an asynchronous operation completed. Further, issuing an asynchronous request successfully guarantees that space exists to queue the completion notification.

The `close(2)`, `exit(2)` and `execve(2)`) functions block until all pending asynchronous I/O operations can be canceled by the system.

It is an error to use the same result buffer in more than one outstanding request. These structures can be reused only after the system has completed the operation.

Return Values Upon successful completion, `aioread()` and `aiowrite()` return `0`. Upon failure, `aioread()` and `aiowrite()` return `−1` and set `errno` to indicate the error.

Errors The `aioread()` and `aiowrite()` functions will fail if:

`EAGAIN` The number of asynchronous requests that the system can handle at any one time has been exceeded

`EBADF` The *fildes* argument is not a valid file descriptor open for reading.

`EFAULT` At least one of *bufp* or *resultp* points to an address outside the address space of the requesting process. This condition is reported only if detected by the application process.

`EINVAL` The *resultp* argument is currently being used by an outstanding asynchronous request.

`EINVAL` The *offset* argument is not a valid offset for this file system type.

`ENOMEM` Memory resources are unavailable to initiate request.

Usage The `aioread()` and `aiowrite()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `close(2)`, `execve(2)`, `exit(2)`, `llseek(2)`, `lseek(2)`, `open(2)`, `read(2)`, `write(2)`, `aiocancel(3C)`, `aiowait(3C)`, `attributes(5)`, `lf64(5)`

Name aio_return – retrieve return status of an asynchronous I/O operation

Synopsis

```
#include <aio.h>

ssize_t aio_return(struct aiocb *aiocbp);
```

Description The `aio_return()` function returns the return status associated with the `aiocb` structure referenced by the *aiocbp* argument. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding `read(2)`, `write(2)`, or `fsync(3C)` function call. If the error status for the operation is equal to `EINPROGRESS`, then the return status for the operation is undefined. The `aio_return()` function may be called exactly once to retrieve the return status of a given asynchronous operation; thereafter, if the same `aiocb` structure is used in a call to `aio_return()` or `aio_error(3C)`, an error may be returned. When the `aiocb` structure referred to by *aiocbp* is used to submit another asynchronous operation, then `aio_return()` may be successfully used to retrieve the return status of that operation.

Return Values If the asynchronous I/O operation has completed, then the return status, as described for `read(2)`, `write(2)`, and `fsync(3C)`, is returned. If the asynchronous I/O operation has not yet completed, the results of `aio_return()` are undefined.

Errors The `aio_return()` function will fail if:

`EINVAL` The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

`ENOSYS` The `aio_return()` function is not supported by the system.

Usage The `aio_return()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `close(2)`, `exec(2)`, `exit(2)`, `fork(2)`, `lseek(2)`, `read(2)`, `write(2)`, `fsync(3C)`, `aio.h(3HEAD)`, `signal.h(3HEAD)`, `aio_cancel(3C)`, `aio_fsync(3C)`, `aio_read(3C)`, `lio_listio(3C)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name aio_suspend – wait for asynchronous I/O request

Synopsis

```
#include <aio.h>

int aio_suspend(const struct aiocb * const list[], int nent,
     const struct timespec *timeout);
```

Description The `aio_suspend()` function suspends the calling thread until at least one of the asynchronous I/O operations referenced by the *list* argument has completed, until a signal interrupts the function, or, if *timeout* is not `NULL`, until the time interval specified by *timeout* has passed. If any of the `aiocb` structures in the list correspond to completed asynchronous I/O operations (that is, the error status for the operation is not equal to `EINPROGRESS`) at the time of the call, the function returns without suspending the calling thread. The *list* argument is an array of pointers to asynchronous I/O control blocks. The *nent* argument indicates the number of elements in the array and is limited to `_AIO_LISTIO_MAX` = 4096. Each `aiocb` structure pointed to will have been used in initiating an asynchronous I/O request via `aio_read(3C)`, `aio_write(3C)`, or `lio_listio(3C)`. This array may contain null pointers, which are ignored. If this array contains pointers that refer to `aiocb` structures that have not been used in submitting asynchronous I/O, the effect is undefined.

If the time interval indicated in the `timespec` structure pointed to by *timeout* passes before any of the I/O operations referenced by *list* are completed, then `aio_suspend()` returns with an error.

Return Values If `aio_suspend()` returns after one or more asynchronous I/O operations have completed, it returns `0`. Otherwise, it returns `−1`, and sets `errno` to indicate the error.

The application may determine which asynchronous I/O completed by scanning the associated error and return status using `aio_error(3C)` and `aio_return(3C)`, respectively.

Errors The `aio_suspend()` function will fail if:

`EAGAIN` No asynchronous I/O indicated in the list referenced by *list* completed in the time interval indicated by *timeout*.

`EINTR` A signal interrupted the `aio_suspend()` function. Since each asynchronous I/O operation might provoke a signal when it completes, this error return can be caused by the completion of one or more of the very I/O operations being awaited.

`EINVAL` The *nent* argument is less than or equal to 0 or greater than `_AIO_LISTIO_MAX`, or the `timespec` structure pointed to by *timeout* is not properly set because *tv_sec* is less than 0 or *tv_nsec* is either less than 0 or greater than 10^9.

`ENOMEM` There is currently not enough available memory; the application can try again later.

`ENOSYS` The `aio_suspend()` function is not supported by the system.

Usage The `aio_suspend()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `aio.h(3HEAD)`, `aio_fsync(3C)`, `aio_read(3C)`, `aio_return(3C)`, `aio_write(3C)`, `lio_listio(3C)`, `signal.h(3HEAD)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name aiowait – wait for completion of asynchronous I/O operation

Synopsis

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

aio_result_t *aiowait(const struct timeval *timeout);
```

Description The `aiowait()` function suspends the calling process until one of its outstanding asynchronous I/O operations completes, providing a synchronous method of notification.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the completion of an asynchronous I/O operation. If *timeout* is a zero pointer, `aiowait()` blocks indefinitely. To effect a poll, the *timeout* parameter should be non-zero, pointing to a zero-valued *timeval* structure.

The `timeval` structure is defined in <`sys/time.h`> and contains the following members:

```
long  tv_sec;           /* seconds */
long  tv_usec;          /* and microseconds */
```

Return Values Upon successful completion, `aiowait()` returns a pointer to the result structure used when the completed asynchronous I/O operation was requested. Upon failure, `aiowait()` returns −`1` and sets `errno` to indicate the error. `aiowait()` returns `0` if the time limit expires.

Errors The `aiowait()` function will fail if:

`EFAULT` The *timeout* argument points to an address outside the address space of the requesting process. See `NOTES`.

`EINTR` The execution of `aiowait()` was interrupted by a signal.

`EINVAL` There are no outstanding asynchronous I/O requests.

`EINVAL` The `tv_secs` member of the `timeval` structure pointed to by *timeout* is less than 0 or the `tv_usecs` member is greater than the number of seconds in a microsecond.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `aiocancel(3C)`, `aioread(3C)`, `attributes(5)`

Notes The `aiowait()` function is the only way to dequeue an asynchronous notification. It can be used either inside a `SIGIO` signal handler or in the main program. One `SIGIO` signal can represent several queued events.

Passing an illegal address as *timeout* will result in setting `errno` to `EFAULT` only if detected by the application process.

Name aio_waitn – wait for completion of asynchronous I/O operations

Synopsis

```
#include <aio.h>

int aio_waitn(struct aiocb *list[], uint_t nent,
     uint_t *nwait, const struct timespec *timeout);
```

Description The `aio_waitn()` function suspends the calling thread until at least the number of requests specified by *nwait* have completed, until a signal interrupts the function, or if *timeout* is not `NULL`, until the time interval specified by *timeout* has passed.

To effect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued `timespec` structure.

The *list* argument is an array of uninitialized I/O completion block pointers to be filled in by the system before `aio_waitn()` returns. The *nent* argument indicates the maximum number of elements that can be placed in *list*[] and is limited to `_AIO_LISTIO_MAX` = 4096.

The *nwait* argument points to the minimum number of requests `aio_waitn()` should wait for. Upon returning, the content of *nwait* is set to the actual number of requests in the `aiocb` list, which can be greater than the initial value specified in *nwait*. The `aio_waitn()` function attempts to return as many requests as possible, up to the number of outstanding asynchronous I/Os but less than or equal to the maximum specified by the *nent* argument. As soon as the number of outstanding asynchronous I/O requests becomes 0, `aio_waitn()` returns with the current list of completed requests.

The `aiocb` structures returned will have been used in initiating an asynchronous I/O request from any thread in the process with `aio_read(3C)`, `aio_write(3C)`, or `lio_listio(3C)`.

If the time interval expires before the expected number of I/O operations specified by *nwait* are completed, `aio_waitn()` returns the number of completed requests and the content of the *nwait* pointer is updated with that number.

If `aio_waitn()` is interrupted by a signal, *nwait* is set to the number of completed requests.

The application can determine the status of the completed asynchronous I/O by checking the associated error and return status using `aio_error(3C)` and `aio_return(3C)`, respectively.

Return Values Upon successful completion, `aio_waitn()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error.

Errors The `aio_waitn()` function will fail if:

`EAGAIN` There are no outstanding asynchronous I/O requests.

`EFAULT` The *list*[], *nwait*, or *timeout* argument points to an address outside the address space of the process. The `errno` variable is set to `EFAULT` only if this condition is detected by the application process.

`EINTR` The execution of `aio_waitn()` was interrupted by a signal.

`EINVAL` The *timeout* element *tv_sec* or *tv_nsec* is < 0, *nent* is set to 0 or > `_AIO_LISTIO_MAX`, or *nwait* is either set to 0 or is > *nent*.

`ENOMEM` There is currently not enough available memory. The application can try again later.

`ETIME` The time interval expired before *nwait* outstanding requests have completed.

Usage The `aio_waitn()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `aio.h(3HEAD)`, `aio_error(3C)`, `aio_read(3C)`, `aio_write(3C)`, `lio_listio(3C)`, `aio_return(3C)`, `attributes(5)`, `lf64(5)`

Name aio_write – asynchronous write to a file

Synopsis

```
#include <aio.h>

int aio_write(struct aiocb *aiocbp);
```

Description The `aio_write()` function allows the calling process to write *aiocbp*→`aio_nbytes` to the file associated with *aiocbp*→`aio_fildes` from the buffer pointed to by *aiocbp*→`aio_buf`. The function call returns when the write request has been initiated or, at a minimum, queued to the file or device. If `_POSIX_PRIORITIZED_IO` is defined and prioritized I/O is supported for this file, then the asynchronous operation is submitted at a priority equal to the scheduling priority of the process minus *aiocbp*→`aio_reqprio`. The *aiocbp* may be used as an argument to `aio_error(3C)` and `aio_return(3C)` in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding.

The *aiocbp* argument points to an `aiocb` structure. If the buffer pointed to by *aiocbp*→`aio_buf` or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

If `O_APPEND` is not set for the file descriptor *aio_fildes*, then the requested operation takes place at the absolute position in the file as given by *aio_offset*, as if `lseek(2)` were called immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to `SEEK_SET`. If `O_APPEND` is set for the file descriptor, write operations append to the file in the same order as the calls were made. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aiocbp*→`aio_sigevent` structure defines what asynchronous notification is to occur when the asynchronous operation completes, as specified in `signal.h(3HEAD)`.

The *aiocbp*→`aio_lio_opcode` field is ignored by `aio_write()`.

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

If `_POSIX_SYNCHRONIZED_IO` is defined and synchronized I/O is enabled on the file associated with *aiocbp*→`aio_fildes`, the behavior of this function shall be according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding to the address range being changed, the result of that action is undefined.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *aiocbp*→`aio_fildes`.

Return Values The `aio_write()` function returns `0` to the calling process if the I/O operation is successfully queued; otherwise, the function returns `−1` and sets `errno` to indicate the error.

Errors The `aio_write()` function will fail if:

`EAGAIN` The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to `aio_write()`, or asynchronously. If any of the conditions below are detected synchronously, the `aio_write()` function returns −1 and sets `errno` to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to −1, and the error status of the asynchronous operation will be set to the corresponding value.

`EBADF` The *aiocbp*→`aio_fildes` argument is not a valid file descriptor open for writing.

`EINVAL` The file offset value implied by *aiocbp*->`aio_offset` would be invalid, *aiocbp*->`aio_reqprio` is not a valid value, or *aiocbp*→`aio_nbytes` is an invalid value.

In the case that the `aio_write()` successfully queues the I/O operation, the return status of the asynchronous operation will be one of the values normally returned by the `write(2)` function call. If the operation is successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation contains one of the values normally set by the `write()` function call, or one of the following:

`EBADF` The *aiocbp*→`aio_fildes` argument is not a valid file descriptor open for writing.

`EINVAL` The file offset value implied by *aiocbp*->`aio_offset` would be invalid.

`ECANCELED` The requested I/O was canceled before the I/O completed due to an explicit `aio_cancel(3C)` request.

The following condition may be detected synchronously or asynchronously:

`EFBIG` The file is a regular file, *aiobcp*→`aio_nbytes` is greater than 0 and the starting offset in *aiobcp*→`aio_offset` is at or beyond the offset maximum in the open file description associated with *aiocbp*→`aio_fildes`.

Usage The `aio_write()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also aio_cancel(3C), aio_error(3C), aio_read(3C), aio_return(3C), lio_listio(3C), close(2), _Exit(2), fork(2), lseek(2), write(2), aio.h(3HEAD), signal.h(3HEAD), attributes(5), lf64(5), standards(5)

Name assert – verify program assertion

Synopsis

```
#include <assert.h>

void assert(int expression);
```

Description The `assert()` macro inserts diagnostics into applications. When executed, if *expression* is `FALSE` (zero), `assert()` prints the error message

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

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the `assert()` statement. These are respectively the values of the preprocessor macros `__FILE__` and `__LINE__`.

Since `assert()` is implemented as a macro, the *expression* may not contain any string literals.

Compiling with the preprocessor option `-DNDEBUG` or with the preprocessor control statement `#define NDEBUG` ahead of the `#include <assert.h>` statement, will stop assertions from being compiled into the program.

Messages printed from this function are in the native language specified by the `LC_MESSAGES` locale category. See `setlocale(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `abort(3C)`, `gettext(3C)`, `setlocale(3C)`, `attributes(5)`, `standards(5)`

Name atexit – register a function to run at process termination or object unloading

Synopsis

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Description The `atexit()` function registers the function pointed to by *func* to be called without arguments on normal termination of the program or when the object defining the function is unloaded.

Normal termination occurs by either a call to the `exit(3C)` function or a return from `main()`. Object unloading occurs when a call to `dlclose(3C)` results in the object becoming unreferenced.

The number of functions that may be registered with `atexit()` is limited only by available memory (refer to the `_SC_ATEXIT_MAX` argument of `sysconf(3C)`).

After a successful call to any of the `exec(2)` functions, any functions previously registered by `atexit()` are no longer registered.

On process exit, functions are called in the reverse order of their registration. On object unloading, any functions belonging to an unloadable object are called in the reverse order of their registration.

Return Values Upon successful completion, the `atexit()` function returns 0. Otherwise, it returns a non-zero value.

Errors The `atexit()` function may fail if:

`ENOMEM` Insufficient storage space is available.

Usage The functions registered by a call to `atexit()` must return to ensure that all registered functions are called.

There is no way for an application to tell how many functions have already been registered with `atexit()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also exec(2), dlclose(3C), exit(3C), sysconf(3C), attributes(5)

Name atomic_add, atomic_add_8, atomic_add_char, atomic_add_16, atomic_add_short, atomic_add_32, atomic_add_int, atomic_add_long, atomic_add_64, atomic_add_ptr, atomic_add_8_nv, atomic_add_char_nv, atomic_add_16_nv, atomic_add_short_nv, atomic_add_32_nv, atomic_add_int_nv, atomic_add_long_nv, atomic_add_64_nv, atomic_add_ptr_nv – atomic add operations

Synopsis

```
#include <atomic.h>

void atomic_add_8(volatile uint8_t *target, int8_t delta);

void atomic_add_char(volatile uchar_t *target, signed char delta);

void atomic_add_16(volatile uint16_t *target, int16_t delta);

void atomic_add_short(volatile ushort_t *target, short delta);

void atomic_add_32(volatile uint32_t *target, int32_t delta);

void atomic_add_int(volatile uint_t *target, int delta);

void atomic_add_long(volatile ulong_t *target, long delta);

void atomic_add_64(volatile uint64_t *target, int64_t delta);

void atomic_add_ptr(volatile void *target, ssize_t delta);

uint8_t atomic_add_8_nv(volatile uint8_t *target, int8_t delta);

uchar_t atomic_add_char_nv(volatile uchar_t *target, signed char delta);

uint16_t atomic_add_16_nv(volatile uint16_t *target, int16_t delta);

ushort_t atomic_add_short_nv(volatile ushort_t *target, shortdelta);

uint32_t atomic_add_32_nv(volatile uint32_t *target, int32_t delta);

uint_t atomic_add_int_nv(volatile uint_t *target, int delta);

ulong_t atomic_add_long_nv(volatile ulong_t *target, long delta);

uint64_t atomic_add_64_nv(volatile uint64_t *target, int64_t delta);

void *atomic_add_ptr_nv(volatile void *target, ssize_t delta);
```

Description These functions enable the addition of *delta* to the value stored in *target* to occur in an atomic manner.

Return Values The *_nv() variants of these functions return the new value of *target*.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also atomic_and(3C), atomic_bits(3C), atomic_cas(3C), atomic_dec(3C), atomic_inc(3C), atomic_or(3C), atomic_swap(3C), membar_ops(3C), attributes(5), atomic_ops(9F)

Notes The *_nv() variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically* (for example, when decrementing a reference count and checking whether it went to zero).

Name atomic_and, atomic_and_8, atomic_and_uchar, atomic_and_16, atomic_and_ushort, atomic_and_32, atomic_and_uint, atomic_and_ulong, atomic_and_64, atomic_and_8_nv, atomic_and_uchar_nv, atomic_and_16_nv, atomic_and_ushort_nv, atomic_and_32_nv, atomic_and_uint_nv, atomic_and_ulong_nv, atomic_and_64_nv – atomic AND operations

Synopsis

```
#include <atomic.h>

void atomic_and_8(volatile uint8_t *target, uint8_t bits);

void atomic_and_uchar(volatile uchar_t *target, uchar_t bits);

void atomic_and_16(volatile uint16_t *target, uint16_t bits);

void atomic_and_ushort(volatile ushort_t *target, ushort_t bits);

void atomic_and_32(volatile uint32_t *target, uint32_t bits);

void atomic_and_uint(volatile uint_t *target, uint_t bits);

void atomic_and_ulong(volatile ulong_t *target, ulong_t bits);

void atomic_and_64(volatile uint64_t *target, uint64_t bits);

uint8_t atomic_and_8_nv(volatile uint8_t *target, uint8_t bits);

uchar_t atomic_and_uchar_nv(volatile uchar_t *target, uchar_t bits);

uint16_t atomic_and_16_nv(volatile uint16_t *target, uint16_t bits);

ushort_t atomic_and_ushort_nv(volatile ushort_t *target, ushort_t bits);

uint32_t atomic_and_32_nv(volatile uint32_t *target, uint32_t bits);

uint_t atomic_and_uint_nv(volatile uint_t *target, uint_t bits);

ulong_t atomic_and_ulong_nv(volatile ulong_t *target, ulong_t bits);

uint64_t atomic_and_64_nv(volatile uint64_t *target, uint64_t bits);
```

Description These functions enable the bitwise AND of *bits* to the value stored in *target* to occur in an atomic manner.

Return Values The *_nv() variants of these functions return the new value of *target*.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also atomic_add(3C), atomic_bits(3C), atomic_cas(3C), atomic_dec(3C), atomic_inc(3C), atomic_or(3C), atomic_swap(3C), membar_ops(3C), attributes(5), atomic_ops(9F)

Notes The *_nv() variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically*.

Name atomic_bits, atomic_set_long_excl, atomic_clear_long_excl – atomic set and clear bit operations

Synopsis

```
#include <atomic.h>

int atomic_set_long_excl(volatile ulong_t *target, uint_t bit);

int atomic_clear_long_excl(volatile ulong_t *target, uint_t bit);
```

Description The `atomic_set_long_excl()` and `atomic_clear_long_excl()` functions perform an exclusive atomic bit set or clear operation on *target*. The value of *bit* specifies the number of the bit to be modified within target. Bits are numbered from zero to one less than the maximum number of bits in a `long`. If the value of bit falls outside of this range, the result of the operation is undefined.

Return Values The `atomic_set_long_excl()` and `atomic_clear_long_excl()` functions return 0 if *bit* was successfully set or cleared. They return -1 if *bit* was already set or cleared.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `atomic_add(3C)`, `atomic_and(3C)`, `atomic_cas(3C)`, `atomic_dec(3C)`, `atomic_inc(3C)`, `atomic_or(3C)`, `atomic_swap(3C)`, `membar_ops(3C)`, `attributes(5)`, `atomic_ops(9F)`

Name atomic_cas, atomic_cas_8, atomic_cas_uchar, atomic_cas_16, atomic_cas_ushort, atomic_cas_32, atomic_cas_uint, atomic_cas_ulong, atomic_cas_64, atomic_cas_ptr – atomic compare and swap operations

Synopsis

```
#include <atomic.h>

uint8_t atomic_cas_8(volatile uint8_t *target, uint8_t cmp,
     uint8_t newval);

uchar_t atomic_cas_uchar(volatile uchar_t *target, uchar_t cmp,
     uchar_t newval);

uint16_t atomic_cas_16(volatile uint16_t *target, uint16_t cmp,
     uint16_t newval);

ushort_t atomic_cas_ushort(volatile ushort_t *target, ushort_t cmp,
     ushort_t newval);

uint32_t atomic_cas_32(volatile uint32_t *target, uint32_t cmp,
     uint32_t newval);

uint_t atomic_cas_uint(volatile uint_t *target, uint_t cmp,
     uint_t newval);

ulong_t atomic_cas_ulong(volatile ulong_t *target, ulong_t cmp,
     ulong_t newval);

uint64_t atomic_cas_64(volatile uint64_t *target, uint64_t cmp,
     uint64_t newval);

void *atomic_cas_ptr(volatile void *target, void *cmp,
     void *newval);
```

Description These functions enable a compare and swap operation to occur atomically. The value stored in *target* is compared with *cmp*. If these values are equal, the value stored in *target* is replaced with *newval*. The old value stored in *target* is returned by the function whether or not the replacement occurred.

Return Values These functions return the old value of **target*.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also atomic_add(3C), atomic_and(3C), atomic_bits(3C), atomic_dec(3C), atomic_inc(3C), atomic_or(3C), atomic_swap(3C), membar_ops(3C), attributes(5), atomic_ops(9F)

Name atomic_dec, atomic_dec_8, atomic_dec_uchar, atomic_dec_16, atomic_dec_ushort, atomic_dec_32, atomic_dec_uint, atomic_dec_ulong, atomic_dec_64, atomic_dec_ptr, atomic_dec_8_nv, atomic_dec_uchar_nv, atomic_dec_16_nv, atomic_dec_ushort_nv, atomic_dec_32_nv, atomic_dec_uint_nv, atomic_dec_ulong_nv, atomic_dec_64_nv, atomic_dec_ptr_nv – atomic decrement operations

Synopsis

```
#include <atomic.h>

void atomic_dec_8(volatile uint8_t *target);

void atomic_dec_uchar(volatile uchar_t *target);

void atomic_dec_16(volatile uint16_t *target);

void atomic_dec_ushort(volatile ushort_t *target);

void atomic_dec_32(volatile uint32_t *target);

void atomic_dec_uint(volatile uint_t *target);

void atomic_dec_ulong(volatile ulong_t *target);

void atomic_dec_64(volatile uint64_t *target);

void atomic_dec_ptr(volatile void *target);

uint8_t atomic_dec_8_nv(volatile uint8_t *target);

uchar_t atomic_dec_uchar_nv(volatile uchar_t *target);

uint16_t atomic_dec_16_nv(volatile uint16_t *target);

ushort_t atomic_dec_ushort_nv(volatile ushort_t *target);

uint32_t atomic_dec_32_nv(volatile uint32_t *target);

uint_t atomic_dec_uint_nv(volatile uint_t *target);

ulong_t atomic_dec_ulong_nv(volatile ulong_t *target);

uint64_t atomic_dec_64_nv(volatile uint64_t *target);

void *atomic_dec_ptr_nv(volatile void *target);
```

Description These functions enable the decrementing (by one) of the value stored in *target* to occur in an atomic manner.

Return Values The `*_nv()` variants of these functions return the new value of *target*.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also atomic_add(3C), atomic_and(3C), atomic_bits(3C), atomic_cas(3C), atomic_inc(3C), atomic_or(3C), atomic_swap(3C), membar_ops(3C), attributes(5), atomic_ops(9F)

Notes The *_nv() variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value atomically (for example, when decrementing a reference count and checking whether it went to zero).

Name atomic_inc, atomic_inc_8, atomic_inc_uchar, atomic_inc_16, atomic_inc_ushort, atomic_inc_32, atomic_inc_uint, atomic_inc_ulong, atomic_inc_64, atomic_inc_ptr, atomic_inc_8_nv, atomic_inc_uchar_nv, atomic_inc_16_nv, atomic_inc_ushort_nv, atomic_inc_32_nv, atomic_inc_uint_nv, atomic_inc_ulong_nv, atomic_inc_64_nv, atomic_inc_ptr_nv – atomic increment operations

Synopsis

```
#include <atomic.h>

void atomic_inc_8(volatile uint8_t *target);

void atomic_inc_uchar(volatile uchar_t *target);

void atomic_inc_16(volatile uint16_t *target);

void atomic_inc_ushort(volatile ushort_t *target);

void atomic_inc_32(volatile uint32_t *target);

void atomic_inc_uint(volatile uint_t *target);

void atomic_inc_ulong(volatile ulong_t *target);

void atomic_inc_64(volatile uint64_t *target);

void atomic_inc_ptr(volatile void *target);

uint8_t atomic_inc_8_nv(volatile uint8_t *target);

uchar_t atomic_inc_uchar_nv(volatile uchar_t *target);

uint16_t atomic_inc_16_nv(volatile uint16_t *target);

ushort_t atomic_inc_ushort_nv(volatile ushort_t *target);

uint32_t atomic_inc_32_nv(volatile uint32_t *target);

uint_t atomic_inc_uint_nv(volatile uint_t *target);

ulong_t atomic_inc_ulong_nv(volatile ulong_t *target);

uint64_t atomic_inc_64_nv(volatile uint64_t *target);

void *atomic_inc_ptr_nv(volatile void *target);
```

Description These functions enable the incrementing (by one) of the value stored in *target* to occur in an atomic manner.

Return Values The `*_nv()` variants of these functions return the new value of *target*.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also atomic_add(3C), atomic_and(3C), atomic_bits(3C), atomic_cas(3C), atomic_dec(3C), atomic_or(3C), atomic_swap(3C), membar_ops(3C), attributes(5), atomic_ops(9F)

Notes The *_nv() variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically*.

Name atomic_ops – atomic operations

Synopsis

```
#include <atomic.h>
```

Description This collection of functions provides atomic memory operations. There are 8 different classes of atomic operations:

atomic_add(3C)	These functions provide an atomic addition of a signed value to a variable.
atomic_and(3C)	These functions provide an atomic logical 'and' of a value to a variable.
atomic_bits(3C)	These functions provide atomic bit setting and clearing within a variable.
atomic_cas(3C)	These functions provide an atomic comparison of a value with a variable. If the comparison is equal, then swap in a new value for the variable, returning the old value of the variable in either case.
atomic_dec(3C)	These functions provide an atomic decrement on a variable.
atomic_inc(3C)	These functions provide an atomic increment on a variable.
atomic_or(3C)	These functions provide an atomic logical 'or' of a value to a variable.
atomic_swap(3C)	These functions provide an atomic swap of a value with a variable, returning the old value of the variable.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also atomic_add(3C), atomic_and(3C), atomic_bits(3C), atomic_cas(3C), atomic_dec(3C), atomic_inc(3C), atomic_or(3C), atomic_swap(3C), membar_ops(3C), attributes(5)

Notes Atomic instructions ensure global visibility of atomically-modified variables on completion. In a relaxed store order system, this does not guarantee that the visibility of other variables will be synchronized with the completion of the atomic instruction. If such synchronization is required, memory barrier instructions must be used. See membar_ops(3C).

Atomic instructions can be expensive since they require synchronization to occur at a hardware level. This means they should be used with care to ensure that forcing hardware level synchronization occurs a minimum number of times. For example, if you have several variables that need to be incremented as a group, and each needs to be done atomically, then do so with a mutex lock protecting all of them being incremented rather than using the atomic_inc(3C) operation on each of them.

Name atomic_or, atomic_or_8, atomic_or_uchar, atomic_or_16, atomic_or_ushort, atomic_or_32, atomic_or_uint, atomic_or_ulong, atomic_or_64, atomic_or_8_nv, atomic_or_uchar_nv, atomic_or_16_nv, atomic_or_ushort_nv, atomic_or_32_nv, atomic_or_uint_nv, atomic_or_ulong_nv, atomic_or_64_nv – atomic OR operations

Synopsis

```
#include <atomic.h>

void atomic_or_8(volatile uint8_t *target, uint8_t bits);

void atomic_or_uchar(volatile uchar_t *target, uchar_t bits);

void atomic_or_16(volatile uint16_t *target, uint16_t bits);

void atomic_or_ushort(volatile ushort_t *target, ushort_t bits);

void atomic_or_32(volatile uint32_t *target, uint32_t bits);

void atomic_or_uint(volatile uint_t *target, uint_t bits);

void atomic_or_ulong(volatile ulong_t *target, ulong_t bits);

void atomic_or_64(volatile uint64_t *target, uint64_t bits);

uint8_t atomic_or_8_nv(volatile uint8_t *target, uint8_t bits);

uchar_t atomic_or_uchar_nv(volatile uchar_t *target, uchar_t bits);

uint16_t atomic_or_16_nv(volatile uint16_t *target, uint16_t bits);

ushort_t atomic_or_ushort_nv(volatile ushort_t *target, ushort_t bits);

uint32_t atomic_or_32_nv(volatile uint32_t *target, uint32_t bits);

uint_t atomic_or_uint_nv(volatile uint_t *target, uint_t bits);

ulong_t atomic_or_ulong_nv(volatile ulong_t *target, ulong_t bits);

uint64_t atomic_or_64_nv(volatile uint64_t *target, uint64_t bits);
```

Description These functions enable the the bitwise OR of *bits* to the value stored in *target* to occur in an atomic manner.

Return Values The *_nv() variants of these functions return the new value of *target*.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also atomic_add(3C), atomic_and(3C), atomic_bits(3C), atomic_cas(3C), atomic_dec(3C), atomic_inc(3C), atomic_swap(3C), membar_ops(3C), attributes(5), atomic_ops(9F)

Notes The *_nv() variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically*.

Name atomic_swap, atomic_swap_8, atomic_swap_uchar, atomic_swap_16, atomic_swap_ushort, atomic_swap_32, atomic_swap_uint, atomic_swap_ulong, atomic_swap_64, atomic_swap_ptr – atomic swap operations

Synopsis

```
#include <atomic.h>

uint8_t atomic_swap_8(volatile uint8_t *target, uint8_t newval);

uchar_t atomic_swap_uchar(volatile uchar_t *target, uchar_t newval);

uint16_t atomic_swap_16(volatile uint16_t *target, uint16_t newval);

ushort_t atomic_swap_ushort(volatile ushort_t *target, ushort_t newval);

uint32_t atomic_swap_32(volatile uint32_t *target, uint32_t newval);

uint_t atomic_swap_uint(volatile uint_t *target, uint_t newval);

ulong_t atomic_swap_ulong(volatile ulong_t *target, ulong_t newval);

uint64_t atomic_swap_64(volatile uint64_t *target, uint64_t newval);

void *atomic_swap_ptr(volatile void *target, void *newval);
```

Description These functions enable a swap operation to occur atomically. The value stored in *target* is replaced with *newval*. The old value is returned by the function.

Return Values These functions return the old of **target*.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also atomic_add(3C), atomic_and(3C), atomic_bits(3C), atomic_dec(3C), atomic_inc(3C), atomic_or(3C), atomic_cas(3C), membar_ops(3C), attributes(5), atomic_ops(9F)

Name attropen – open a file

Synopsis

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

int attropen(const char *path, const char *attrpath, int oflag,
     /* mode_t mode */...);
```

Description The `attropen()` function is similar to the open(2) function except that it takes a second path argument, *attrpath*, that identifies an extended attribute file associated with the first *path* argument. This function returns a file descriptor for the extended attribute rather than the file named by the initial argument.

The `O_XATTR` flag is set by default for `attropen()` and the *attrpath* argument is always interpreted as a reference to an extended attribute. Extended attributes must be referenced with a relative path; providing an absolute path results in a normal file reference.

Return Values Refer to open(2).

Errors Refer to open(2).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also open(2), attributes(5), fsattr(5)

Name basename – return the last element of a path name

Synopsis

```
#include <libgen.h>

char *basename(char *path);
```

Description The `basename()` function takes the pathname pointed to by *path* and returns a pointer to the final component of the pathname, deleting any trailing '/' characters.

If the string consists entirely of the '/' character, `basename()` returns a pointer to the string "/" .

If *path* is a null pointer or points to an empty string, `basename()` returns a pointer to the string "." .

Return Values The `basename()` function returns a pointer to the final component of *path*.

Usage The `basename()` function may modify the string pointed to by *path*, and may return a pointer to static storage that may then be overwritten by a subsequent call to `basename()`.

When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.

Examples **EXAMPLE 1** Examples for `Input String` and `Output String`

Input String	Output String
"/usr/lib"	"lib"
"/usr/"	"usr"
"/"	"/"

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `basename(1)`, `dirname(3C)`, `attributes(5)`, `standards(5)`

Name bsdmalloc – memory allocator

Synopsis

```
cc [ flag ... ] file ... -lbsdmalloc [ library ... ]

char *malloc(sizeunsigned size;

int free( ptrchar *ptr;

char *realloc( ptr, sizechar *ptr;
unsigned size;
```

Description These routines provide a general-purpose memory allocation package. They maintain a table of free blocks for efficient allocation and coalescing of free storage. When there is no suitable space already free, the allocation routines call `sbrk(2)` to get more memory from the system. Each of the allocation routines returns a pointer to space suitably aligned for storage of any type of object. Each returns a null pointer if the request cannot be completed.

The `malloc()` function returns a pointer to a block of at least *size* bytes, which is appropriately aligned.

The `free()` function releases a previously allocated block. Its argument is a pointer to a block previously allocated by `malloc()` or `realloc()`. The `free()` function does not set `errno`.

The `realloc()` function changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If the new size of the block requires movement of the block, the space for the previous instantiation of the block is freed. If the new size is larger, the contents of the newly allocated portion of the block are unspecified. If *ptr* is `NULL`, `realloc()` behaves like `malloc()` for the specified size. If *size* is 0 and *ptr* is not a null pointer, the space pointed to is freed.

Return Values The `malloc()` and `realloc()` functions return a null pointer if there is not enough available memory. They return a non-null pointer if *size* is 0. These pointers should not be dereferenced. When `realloc()` returns `NULL`, the block pointed to by *ptr* is left intact. Always cast the value returned by `malloc()` and `realloc()`.

Errors If `malloc()` or `realloc()` returns unsuccessfully, `errno` will be set to indicate the following:

`ENOMEM` *size* bytes of memory cannot be allocated because it exceeds the physical limits of the system.

`EAGAIN` There is not enough memory available at this point in time to allocate *size* bytes of memory; but the application could try again later.

Usage Using `realloc()` with a block freed before the most recent call to `malloc()` or `realloc()` results in an error.

Comparative features of the various allocation libraries can be found in the `umem_alloc(3MALLOC)` manual page.

See Also brk(2), malloc(3C), malloc(3MALLOC), mapmalloc(3MALLOC), umem_alloc(3MALLOC)

Warnings Use of `libbsdmalloc` renders an application non-SCD compliant.

The `libbsdmalloc` routines are incompatible with the memory allocation routines in the standard C-library (libc): malloc(3C), alloca(3C), calloc(3C), free(3C), memalign(3C), realloc(3C), and valloc(3C).

Name bsd_signal – simplified signal facilities

Synopsis

```
#include <signal.h>

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

Description The `bsd_signal()` function provides a partially compatible interface for programs written to historical system interfaces (see `USAGE` below).

The function call `bsd_signal`(*sig*,*func*) has an effect as if implemented as:

```
void (*bsd_signal(int sig, void (*func) (int))) (int)
{
     struct sigaction act, oact;

     act.sa_handler = func;
     act.sa_flags = SA_RESTART;
     sigemptyset(&act.sa_mask);
     sigaddset(&act.sa_mask, sig);
     if (sigaction(sig, &act, &oact) == -1)
          return(SIG_ERR);
     return(oact.sa_handler);
}
```

The handler function should be declared:

```
void handler(int sig);
```

where *sig* is the signal number. The behavior is undefined if *func* is a function that takes more than one argument, or an argument of a different type.

Return Values Upon successful completion, `bsd_signal()` returns the previous action for *sig*. Otherwise, `SIG_ERR` is returned and `errno` is set to indicate the error.

Errors Refer to `sigaction(2)`.

Usage This function is a direct replacement for the BSD `signal()` function for simple applications that are installing a single-argument signal handler function. If a BSD signal handler function is being installed that expects more than one argument, the application has to be modified to use `sigaction(2)`. The `bsd_signal()` function differs from `signal()` in that the `SA_RESTART` flag is set and the `SA_RESETHAND` will be clear when `bsd_signal()` is used. The state of these flags is not specified for `signal()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

Standard	See `standards(5)`.

See Also `sigaction(2)`, `sigaddset(3C)`, `sigemptyset(3C)`, `attributes(5)`, `standards(5)`

Name bsearch – binary search a sorted table

Synopsis

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base, size_t nel, size_t size,
     int (*compar)(const void *,const void *));
```

Description The `bsearch()` function is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table (an array) indicating where a datum may be found or a null pointer if the datum cannot be found. The table must be previously sorted in increasing order according to a comparison function pointed to by *compar*.

The *key* argument points to a datum instance to be sought in the table. The *base* argument points to the element at the base of the table. The *nel* argument is the number of elements in the table. The `size` argument is the number of bytes in each element.

The comparison function pointed to by *compar* is called with two arguments that point to the *key* object and to an array element, in that order. The function must return an integer less than, equal to, or greater than 0 if the *key* object is considered, respectively, to be less than, equal to, or greater than the array element.

Return Values The `bsearch()` function returns a pointer to a matching member of the array, or a null pointer if no match is found. If two or more members compare equal, which member is returned is unspecified.

Usage The pointers to the key and the element at the base of the table should be of type pointer-to-element.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

If the number of elements in the table is less than the size reserved for the table, *nel* should be the lower number.

The `bsearch()` function safely allows concurrent access by multiple threads to disjoint data, such as overlapping subtrees or tables.

Examples **EXAMPLE 1** Examples for searching a table containing pointers to nodes.

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This program reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct node {    /* these are stored in the table */
```

EXAMPLE 1 Examples for searching a table containing pointers to nodes. *(Continued)*

```
    char *string;
    int length;
};
static struct node table[] = {    /* table to be searched */
    { "asparagus", 10 },
    { "beans", 6 },
    { "tomato", 7 },
    { "watermelon", 11 },
};

main()
{
    struct node *node_ptr, node;
    /* routine to compare 2 nodes */
    static int node_compare(const void *, const void *);
    char str_space[20];   /* space to read string into */

    node.string = str_space;
    while (scanf("%20s", node.string) != EOF) {
        node_ptr = bsearch( &node,
            table, sizeof(table)/sizeof(struct node),
            sizeof(struct node), node_compare);
        if (node_ptr != NULL) {
            (void) printf("string = %20s, length = %d\n",
                node_ptr->string, node_ptr->length);
        } else {
            (void)printf("not found: %20s\n", node.string);
        }
    }
    return(0);
}

/* routine to compare two nodes based on an  */
/* alphabetical ordering of the string field */
static int
node_compare(const void *node1, const void *node2) {
    return (strcmp(
            ((const struct node *)node1)->string,
            ((const struct node *)node2)->string));
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Standard	See standards(5).

See Also hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C), attributes(5), standards(5)

Name bstring, bcopy, bcmp, bzero – memory operations

Synopsis

```
#include <strings.h>

void bcopy(const void *s1, void *s2, size_t n);

int bcmp(const void *s1, const void *s2, size_t n);

void bzero(void *s, size_t n);
```

Description The `bcopy()`, `bcmp()`, and `bzero()` functions operate as efficiently as possible on memory areas (arrays of bytes bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area. These functions are similar to the `memcpy()`, `memcmp()`, and `memset()` functions described on the `memory(3C)` manual page.

The `bcopy()` function copies *n* bytes from memory area *s1* to *s2*. Copying between objects that overlap will take place correctly.

The `bcmp()` function compares the first *n* bytes of its arguments, returning 0 if they are identical and 1 otherwise. The `bcmp()` function always returns 0 when *n* is 0.

The `bzero()` function sets the first *n* bytes in memory area *s* to 0.

Warnings The `bcopy()` function takes parameters backwards from `memcmp()`. See `memory(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `memory(3C)`, `attributes(5)`, `standards(5)`

Name btowc – single-byte to wide-character conversion

Synopsis

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

wint_t btowc(int c);
```

Description The `btowc()` function determines whether *c* constitutes a valid (one-byte) character in the initial shift state.

The behavior of this function is affected by the `LC_CTYPE` category of the current locale. See `environ(5)`.

Return Values The `btowc()` function returns `WEOF` if *c* has the value `EOF` or if `(unsigned char)`*c* does not constitute a valid (one-byte) character in the initial shift state. Otherwise, it returns the wide-character representation of that character.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `setlocale(3C)`, `wctob(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Notes The `btowc()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Name catgets – read a program message

Synopsis

```
#include <nl_types.h>

char *catgets(nl_catd catd, int set_num, int msg_num, const char *s);
```

Description The `catgets()` function attempts to read message *msg_num*, in set *set_num*, from the message catalog identified by *catd*. The *catd* argument is a catalog descriptor returned from an earlier call to `catopen()`. The *s* argument points to a default message string which will be returned by `catgets()` if the identified message catalog is not currently available.

Return Values If the identified message is retrieved successfully, `catgets()` returns a pointer to an internal buffer area containing the null terminated message string. If the call is unsuccessful for any reason, `catgets()` returns a pointer to *s* and `errno` may be set to indicate the error.

Errors The `catgets()` function may fail if:

`EBADF` The *catd* argument is not a valid message catalogue descriptor open for reading.

`EBADMSG` The number of `%n` specifiers that appear in the message string specified by *s* does not match the number of `%n` specifiers that appear in the message identified by *set_id* and *msg_id* in the specified message catalog.

`EINTR` The read operation was terminated due to the receipt of a signal, and no data was transferred.

`EINVAL` The message catalog identified by *catd* is corrupted.

`ENOMSG` The message identified by *set_id* and *msg_id* is not in the message catalog.

Usage The `catgets()` function can be used safely in multithreaded applications as long as `setlocale(3C)` is not being called to change the locale.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `gencat(1)`, `catclose(3C)`, `catopen(3C)`, `gettext(3C)`, `setlocale(3C)`, `attributes(5)`, `standards(5)`

International Language Environments Guide

Name catopen, catclose – open/close a message catalog

Synopsis

```
#include <nl_types.h>

nl_catd catopen(const char *name, int oflag);

int catclose(nl_catd catd);
```

Description The `catopen()` function opens a message catalog and returns a message catalog descriptor. *name* specifies the name of the message catalog to be opened. If *name* contains a "/", then *name* specifies a complete pathname for the message catalog; otherwise, the environment variable `NLSPATH` is used and `/usr/lib/locale/`*locale*`/LC_MESSAGES` must exist. If `NLSPATH` does not exist in the environment, or if a message catalog cannot be opened in any of the paths specified by `NLSPATH`, then the default path `/usr/lib/locale/`*locale*`/LC_MESSAGES` is used. In the "C" locale, `catopen()` will always succeed without checking the default search path.

The names of message catalogs and their location in the filesystem can vary from one system to another. Individual applications can choose to name or locate message catalogs according to their own special needs. A mechanism is therefore required to specify where the catalog resides.

The `NLSPATH` variable provides both the location of message catalogs, in the form of a search path, and the naming conventions associated with message catalog files. For example:

```
NLSPATH=/nlslib/%L/%N.cat:/nlslib/%N/%L
```

The metacharacter `%` introduces a substitution field, where `%L` substitutes the current setting of either the `LANG` environment variable, if the value of *oflag* is `0`, or the `LC_MESSAGES` category, if the value of *oflag* is `NL_CAT_LOCALE`, and `%N` substitutes the value of the *name* parameter passed to `catopen()`. Thus, in the above example, `catopen()` will search in `/nlslib/$LANG/`*name*`.cat`, if *oflag* is `0`, or in `/nlslib/{LC_MESSAGES}/`*name*`.cat`, if *oflag* is `NL_CAT_LOCALE`.

The `NLSPATH` variable will normally be set up on a system wide basis (in `/etc/profile`) and thus makes the location and naming conventions associated with message catalogs transparent to both programs and users.

The full set of metacharacters is:

`%N` The value of the name parameter passed to `catopen()`.

`%L` The value of `LANG` or `LC_MESSAGES`.

`%l` The value of the *language* element of `LANG` or `LC_MESSAGES`.

`%t` The value of the *territory* element of `LANG` or `LC_MESSAGES`.

`%c` The value of the *codeset* element of `LANG` or `LC_MESSAGES`.

`%%` A single %.

The LANG environment variable provides the ability to specify the user's requirements for native languages, local customs and character set, as an ASCII string in the form

```
LANG=language[_territory[.codeset]]
```

A user who speaks German as it is spoken in Austria and has a terminal which operates in ISO 8859/1 codeset, would want the setting of the LANG variable to be

```
LANG=De_A.88591
```

With this setting it should be possible for that user to find any relevant catalogs should they exist.

Should the LANG variable not be set, the value of LC_MESSAGES as returned by setlocale() is used. If this is NULL, the default path as defined in <nl_types.h> is used.

A message catalogue descriptor remains valid in a process until that process closes it, or a successful call to one of the exec functions. A change in the setting of the LC_MESSAGES category may invalidate existing open catalogues.

If a file descriptor is used to implement message catalogue descriptors, the FD_CLOEXEC flag will be set; see <fcntl.h>.

If the value of *oflag* argument is 0, the LANG environment variable is used to locate the catalogue without regard to the LC_MESSAGES category. If the *oflag* argument is NL_CAT_LOCALE, the LC_MESSAGES category is used to locate the message catalogue.

The catclose() function closes the message catalog identified by *catd*. If a file descriptor is used to implement the type nl_catd, that file descriptor will be closed.

Return Values Upon successful completion, catopen() returns a message catalog descriptor for use on subsequent calls to catgets() and catclose(). Otherwise it returns (nl_catd) −1.

Upon successful completion, catclose() returns 0. Otherwise it returns −1 and sets errno to indicate the error.

Errors The catopen() function may fail if:

EACCES	Search permission is denied for the component of the path prefix of the message catalogue or read permission is denied for the message catalogue.
EMFILE	There are OPEN_MAX file descriptors currently open in the calling process.
ENAMETOOLONG	The length of the pathname of the message catalogue exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
ENFILE	Too many files are currently open in the system.

ENOENT	The message catalogue does not exist or the *name* argument points to an empty string.
ENOMEM	Insufficient storage space is available.
ENOTDIR	A component of the path prefix of the message catalogue is not a directory.

The `catclose()` function may fail if:

EBADF	The catalogue descriptor is not valid.
EINTR	The `catclose()` function was interrupted by a signal.

Usage The `catopen()` and `catclose()` functions can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `gencat(1)`, `catgets(3C)`, `gettext(3C)`, `nl_types.h(3HEAD)`, `setlocale(3C)`, `attributes(5)`, `environ(5)`

Name cfgetispeed, cfgetospeed – get input and output baud rate

Synopsis

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termios_p);

speed_t cfgetospeed(const struct termios *termios_p);
```

Description The `cfgetispeed()` function extracts the input baud rate from the `termios` structure to which the *termios_p* argument points.

The `cfgetospeed()` function extracts the output baud rate from the `termios` structure to which the *termios_p* argument points.

These functions returns exactly the value in the `termios` data structure, without interpretation.

Return Values Upon successful completion, `cfgetispeed()` returns a value of type `speed_t` representing the input baud rate.

Upon successful completion, `cfgetospeed()` returns a value of type `speed_t` representing the output baud rate.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe, and Async-Signal-Safe
Standard	See `standards(5)`.

See Also `cfgetospeed(3C)`, `tcgetattr(3C)`, `attributes(5)`, `standards(5)`, `termio(7I)`

Name cfsetispeed, cfsetospeed – set input and output baud rate

Synopsis

```
#include <termios.h>

int cfsetispeed(struct termios *termios_p, speed_t speed);

int cfsetospeed(struct termios *termios_p, speed_t speed);
```

Description The `cfsetispeed()` function sets the input baud rate stored in the structure pointed to by *termios_p* to *speed.*

The `cfsetospeed()` function sets the output baud rate stored in the structure pointed to by *termios_p* to *speed.*

There is no effect on the baud rates set in the hardware until a subsequent successful call to `tcsetattr(3C)` on the same `termios` structure.

Return Values Upon successful completion, `cfsetispeed()` and `cfsetospeed()` return `0`. Otherwise `−1` is returned, and `errno` may be set to indicate the error.

Errors The `cfsetispeed()` and `cfsetospeed()` functions may fail if:

`EINVAL` The *speed* value is not a valid baud rate.

`EINVAL` The value of *speed* is outside the range of possible speed values as specified in `<termios.h>`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe, and Async-Signal-Safe
Standard	See `standards(5)`.

See Also `cfgetispeed(3C)`, `tcsetattr(3C)`, `attributes(5)`, `standards(5)`, `termio(7I)`

Name clearenv – clear the environment

Synopsis

```
#include <stdlib.h>

int clearenv(void);
```

Description The `clearenv()` function clears the environment of all name-value pairs and sets the value of the external variable `environ(5)` to `NULL`.

Return Values Upon successful completion, the `clearenv()` function returns 0. Otherwise, it returns a non-zero value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `getenv(3C)`, `setenv(3C)`, `attributes(5)`, `environ(5)`

Name clock – report CPU time used

Synopsis

```
#include <time.h>

clock_t clock(void);
```

Description The `clock()` function returns the amount of CPU time (in microseconds) used since the first call to `clock()` in the calling process. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed the `wait`(3C) function, the `pclose`(3C) function, or the `system`(3C) function.

Return Values Dividing the value returned by `clock()` by the constant `CLOCKS_PER_SEC`, defined in the `<time.h>` header, will give the time in seconds. If the process time used is not available or cannot be represented, clock returns the value `(clock_t) −1`.

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

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards`(5).

See Also `times`(2), `popen`(3C), `system`(3C), `wait`(3C), `attributes`(5), `standards`(5)

Name clock_nanosleep – high resolution sleep with specifiable clock

Synopsis

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
     const struct timespec *rqtp, struct timespec *rmtp);
```

Description If the flag `TIMER_ABSTIME` is not set in the *flags* argument, the `clock_nanosleep()` function causes the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. The clock used to measure the time is the clock specified by *clock_id*.

If the flag `TIMER_ABSTIME` is set in the *flags* argument, the `clock_nanosleep()` function causes the current thread to be suspended from execution until either the time value of the clock specified by *clock_id* reaches the absolute time specified by the *rqtp* argument, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. If, at the time of the call, the time value specified by *rqtp* is less than or equal to the time value of the specified clock, then `clock_nanosleep()` returns immediately and the calling process is not suspended.

The suspension time caused by this function can be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution, or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time for the relative `clock_nanosleep()` function (that is, with the `TIMER_ABSTIME` flag not set) will not be less than the time interval specified by rqtp, as measured by the corresponding clock. The suspension for the absolute `clock_nanosleep()` function (that is, with the `TIMER_ABSTIME` flag set) will be in effect at least until the value of the corresponding clock reaches the absolute time specified by *rqtp*, except for the case of being interrupted by a signal.

The use of the `clock_nanosleep()` function has no effect on the action or blockage of any signal.

The `clock_nanosleep()` function fails if the *clock_id* argument refers to the CPU-time clock of the calling thread. It is unspecified if *clock_id* values of other CPU-time clocks are allowed.

Return Values If the `clock_nanosleep()` function returns because the requested time has elapsed, its return value is 0.

If the `clock_nanosleep()` function returns because it has been interrupted by a signal, it returns the corresponding error value. For the relative `clock_nanosleep()` function, if the *rmtp* argument is non-null, the timespec structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* argument is NULL, the remaining time is not returned. The absolute `clock_nanosleep()` function has no effect on the structure referenced by *rmtp*.

If `clock_nanosleep()` fails, it shall return the corresponding error value.

Errors The `clock_nanosleep()` function will fail if:

`EINTR` The `clock_nanosleep()` function was interrupted by a signal.

`EINVAL` The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1,000 million; or the TIMER_ABSTIME flag was specified in *flags* and the *rqtp* argument is outside the range for the clock specified by *clock_id*; or the *clock_id* argument does not specify a known clock, or specifies the CPU-time clock of the calling thread.

`ENOTSUP` The *clock_id* argument specifies a clock for which `clock_nanosleep()` is not supported, such as a CPU-time clock.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `clock_getres(3C)`, `nanosleep(3C)`, `pthread_cond_timedwait(3C)`, `sleep(3C)`, `attributes(5)`, `standards(5)`

Name clock_settime, clock_gettime, clock_getres – high-resolution clock operations

Synopsis

```
#include <time.h>

int clock_settime(clockid_t clock_id, const struct timespec *tp);

int clock_gettime(clockid_t clock_id, struct timespec *tp);

int clock_getres(clockid_t clock_id, struct timespec *res);
```

Description The `clock_settime()` function sets the specified clock, *clock_id*, to the value specified by *tp*. Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock are truncated down to the smaller multiple of the resolution.

The `clock_gettime()` function returns the current value *tp* for the specified clock, *clock_id*.

The resolution of any clock can be obtained by calling `clock_getres()`. Clock resolutions are system-dependent and cannot be set by a process. If the argument *res* is not `NULL`, the resolution of the specified clock is stored in the location pointed to by *res*. If *res* is `NULL`, the clock resolution is not returned. If the time argument of `clock_settime()` is not a multiple of *res*, then the value is truncated to a multiple of *res*.

A clock may be systemwide (that is, visible to all processes) or per-process (measuring time that is meaningful only within a process).

A *clock_id* of `CLOCK_REALTIME` is defined in <`time.h`>. This clock represents the realtime clock for the system. For this clock, the values returned by `clock_gettime()` and specified by `clock_settime()` represent the amount of time (in seconds and nanoseconds) since the Epoch. Additional clocks may also be supported. The interpretation of time values for these clocks is unspecified.

A *clock_id* of `CLOCK_HIGHRES` represents the non-adjustable, high-resolution clock for the system. For this clock, the value returned by `clock_gettime(3C)` represents the amount of time (in seconds and nanoseconds) since some arbitrary time in the past; it is not correlated in any way to the time of day, and thus is not subject to resetting or drifting by way of `adjtime(2)`, `ntp_adjtime(2)`, `settimeofday(3C)`, or `clock_settime()`. The time source for this clock is the same as that for `gethrtime(3C)`.

Additional clocks may also be supported. The interpretation of time values for these clocks is unspecified.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `clock_settime()`, `clock_gettime()` and `clock_getres()` functions will fail if:

`EINVAL` The *clock_id* argument does not specify a known clock.

`ENOSYS` The functions `clock_settime()`, `clock_gettime()`, and `clock_getres()` are not supported by this implementation.

The `clock_settime()` function will fail if:

`EINVAL` The *tp* argument to `clock_settime()` is outside the range for the given clock ID; or the *tp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

The `clock_settime()` function may fail if:

`EPERM` The requesting process does not have the appropriate privilege to set the specified clock.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	`clock_gettime()` is Async-Signal-Safe
Standard	See `standards(5)`.

See Also `time(2)`, `ctime(3C)`, `gethrtime(3C)`, `time.h(3HEAD)`, `timer_gettime(3C)`, `attributes(5)`, `standards(5)`

Name closedir – close a directory stream

Synopsis

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

int closedir(DIR *dirp);
```

Description The `closedir()` function closes the directory stream referred to by the argument *dirp*. Upon return, the value of *dirp* may no longer point to an accessible object of the type `DIR`. If a file descriptor is used to implement type `DIR`, that file descriptor will be closed.

Return Values Upon successful completion, `closedir()` returns `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `closedir()` function may fail if:

`EBADF` The *dirp* argument does not refer to an open directory stream.

`EINTR` The `closedir()` function was interrupted by a signal.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `opendir(3C)`, `attributes(5)`, `standards(5)`

Name closefrom, fdwalk – close or iterate over open file descriptors

Synopsis

```
#include <stdlib.h>

void closefrom(int lowfd);

int fdwalk(int (*func)(void *, int), void *cd);
```

Description The `closefrom()` function calls `close(2)` on all open file descriptors greater than or equal to *lowfd*.

The effect of `closefrom`(*lowfd*) is the same as the code

```
#include <sys/resource.h>
struct rlimit rl;
int i;

getrlimit(RLIMIT_NOFILE, &rl);
for (i = lowfd; i < rl.rlim_max; i++)
     (void) close(i);
```

except that `close()` is called only on file descriptors that are actually open, not on every possible file descriptor greater than or equal to *lowfd*, and `close()` is also called on any open file descriptors greater than or equal to `rl.rlim_max` (and *lowfd*), should any exist.

The `fdwalk()` function first makes a list of all currently open file descriptors. Then for each file descriptor in the list, it calls the user-defined function, *func*(*cd*, *fd*), passing it the pointer to the callback data, *cd*, and the value of the file descriptor from the list, *fd*. The list is processed in file descriptor value order, lowest numeric value first.

If *func*() returns a non-zero value, the iteration over the list is terminated and `fdwalk()` returns the non-zero value returned by *func*(). Otherwise, `fdwalk()` returns 0 after having called *func*() for every file descriptor in the list.

The `fdwalk()` function can be used for fine-grained control over the closing of file descriptors. For example, the `closefrom()` function can be implemented as:

```
static int
close_func(void *lowfdp, int fd)
{
     if (fd >= *(int *)lowfdp)
          (void) close(fd);
     return (0);
}

void
closefrom(int lowfd)
{
     (void) fdwalk(close_func, &lowfd);
}
```

The `fdwalk()` function can then be used to count the number of open files in the process.

Return Values No return value is defined for `closefrom()`. If `close()` fails for any of the open file descriptors, the error is ignored and the file descriptors whose `close()` operation failed might remain open on return from `closefrom()`.

The `fdwalk()` function returns the return value of the last call to the callback function *func*(), or 0 if *func*() is never called (no open files).

Errors No errors are defined. The `closefrom()` and `fdwalk()` functions do not set `errno` but `errno` can be set by `close()` or by another function called by the callback function, *func*().

Files `/proc/self/fd` directory (list of open files)

Usage The act of closing all open file descriptors should be performed only as the first action of a daemon process. Closing file descriptors that are in use elsewhere in the current process normally leads to disastrous results.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `close(2)`, `getrlimit(2)`, `proc(4)`, `attributes(5)`

Name cond_init, cond_wait, cond_timedwait, cond_reltimedwait, cond_signal, cond_broadcast, cond_destroy – condition variables

Synopsis

```
cc –mt [ flag... ] file... [ library... ]
#include <thread.h>
#include <synch.h>

int cond_init(cond_t *cvp, int type, void *arg);

int cond_wait(cond_t *cvp, mutex_t *mp);

int cond_timedwait(cond_t *cvp, mutex_t *mp,
     timestruc_t *abstime);

int cond_reltimedwait(cond_t *cvp, mutex_t *mp,
     timestruc_t *reltime);

int cond_signal(cond_t *cvp);

int cond_broadcast(cond_t *cvp);

int cond_destroy(cond_t *cvp);
```

Description

Initialize

Condition variables and mutexes should be global. Condition variables that are allocated in writable memory can synchronize threads among processes if they are shared by the cooperating processes (see mmap(2)) and are initialized for this purpose.

The scope of a condition variable is either intra-process or inter-process. This is dependent upon whether the argument is passed implicitly or explicitly to the initialization of that condition variable. A condition variable does not need to be explicitly initialized. A condition variable is initialized with all zeros, by default, and its scope is set to within the calling process. For inter-process synchronization, a condition variable must be initialized once, and only once, before use.

A condition variable must not be simultaneously initialized by multiple threads or re-initialized while in use by other threads.

Attributes of condition variables can be set to the default or customized at initialization.

The `cond_init()` function initializes the condition variable pointed to by *cvp*. A condition variable can have several different types of behavior, specified by *type*. No current type uses *arg* although a future type may specify additional behavior parameters with *arg*. The *type* argument c take one of the following values:

`USYNC_THREAD` The condition variable can synchronize threads only in this process. This is the default.

`USYNC_PROCESS` The condition variable can synchronize threads in this process and other processes. Only one process should initialize the condition variable. The object initialized with this attribute must be allocated in memory shared

between processes, either in System V shared memory (see shmop(2)) or in memory mapped to a file (see mmap(2)). It is illegal to initialize the object this way and to not allocate it in such shared memory.

Initializing condition variables can also be accomplished by allocating in zeroed memory, in which case, a *type* of USYNC_THREAD is assumed.

If default condition variable attributes are used, statically allocated condition variables can be initialized by the macro DEFAULTCV.

Default condition variable initialization (intra-process):

```
cond_t cvp;

cond_init(&cvp, NULL, NULL); /*initialize condition variable
                                with default*/
```

or

```
cond_init(&cvp, USYNC_THREAD, NULL);
```

or

```
cond_t  cond  = DEFAULTCV;
```

Customized condition variable initialization (inter-process):

```
cond_init(&cvp, USYNC_PROCESS, NULL); /* initialize cv with
                                         inter-process scope */
```

Condition Wait

The condition wait interface allows a thread to wait for a condition and atomically release the associated mutex that it needs to hold to check the condition. The thread waits for another thread to make the condition true and that thread's resulting call to signal and wakeup the waiting thread.

The cond_wait() function atomically releases the mutex pointed to by *mp* and causes the calling thread to block on the condition variable pointed to by *cvp*. The blocked thread may be awakened by cond_signal(), cond_broadcast(), or when interrupted by delivery of a UNIX signal or a fork().

The cond_wait(), cond_timedwait(), and cond_reltimedwait() functions always return with the mutex locked and owned by the calling thread even when returning an error, except when the mutex has the LOCK_ROBUST attribute and has been left irrecoverable by the mutex's last owner. The cond_wait(), cond_timedwait(), and cond_reltimedwait() functions return the appropriate error value if they fail to internally reacquire the mutex.

Condition Signaling

A condition signal allows a thread to unblock a single thread waiting on the condition variable, whereas a condition broadcast allows a thread to unblock all threads waiting on the condition variable.

The `cond_signal()` function unblocks one thread that is blocked on the condition variable pointed to by *cvp*.

The `cond_broadcast()` function unblocks all threads that are blocked on the condition variable pointed to by *cvp*.

If no threads are blocked on the condition variable, then `cond_signal()` and `cond_broadcast()` have no effect.

The `cond_signal()` or `cond_broadcast()` functions can be called by a thread whether or not it currently owns the mutex that threads calling `cond_wait()`, `cond_timedwait()`, or `cond_reltimedwait()` have associated with the condition variable during their waits. If, however, predictable scheduling behavior is required, then that mutex should be locked by the thread prior to calling `cond_signal()` or `cond_broadcast()`.

Destroy The condition destroy functions destroy any state, but not the space, associated with the condition variable.

The `cond_destroy()` function destroys any state associated with the condition variable pointed to by *cvp*. The space for storing the condition variable is not freed.

Return Values Upon successful completion, these functions return `0`. Otherwise, a non-zero value is returned to indicate the error.

Errors The `cond_timedwait()` and `cond_reltimedwait()` functions will fail if:

`ETIME` The time specified by *abstime* or *reltime* has passed.

The `cond_wait()`, `cond_timedwait()`, and `cond_reltimedwait()` functions will fail if:

`EINTR` Interrupted. The calling thread was awakened by the delivery of a UNIX signal.

If the mutex pointed to by *mp* is a robust mutex (initialized with the `LOCK_ROBUST` attribute), the `cond_wait()`, `cond_timedwait()` and `cond_reltimedwait()` functions will, under the specified conditions, return the following error values. For complete information, see the description of the `mutex_lock()` function on the `mutex_init(3C)` manual page.

`ENOTRECOVERABLE` The mutex was protecting the state that has now been left irrecoverable. The mutex has not been acquired.

`EOWNERDEAD` The last owner of the mutex died while holding the mutex, possibly leaving the state it was protecting inconsistent. The mutex is now owned by the caller.

These functions may fail if:

`EFAULT` The *cond*, *attr*, *cvp*, *arg*, *abstime*, or *mutex* argument points to an illegal address.

EINVAL Invalid argument. For `cond_init()`, *type* is not a recognized type. For `cond_timedwait()`, the number of nanoseconds is greater than or equal to 1,000,000,000.

Examples

EXAMPLE 1 Use `cond_wait()` in a loop to test some condition.

The `cond_wait()` functin is normally used in a loop testing some condition, as follows:

```
(void) mutex_lock(mp);
while (cond == FALSE) {
     (void) cond_wait(cvp, mp);
}
(void) mutex_unlock(mp);
```

EXAMPLE 2 Use `cond_timedwait()` in a loop to test some condition.

The `cond_timedwait()` function is normally used in a loop testing some condition. It uses an absolute timeout value as follows:

```
timestruc_t to;
...
(void) mutex_lock(mp);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
      err = cond_timedwait(cvp, mp, &to);
      if (err == ETIME) {
            /* timeout, do something */
            break;
      }
}
(void) mutex_unlock(mp);
```

EXAMPLE 3 Use `cond_reltimedwait()` in a loop to test some condition.

The `cond_reltimedwait()` function is normally used in a loop testing in some condition. It uses a relative timeout value as follows:

```
timestruc_t to;
...
(void) mutex_lock(mp);
while (cond == FALSE) {
     to.tv_sec = TIMEOUT;
     to.tv_nsec = 0;
     err = cond_reltimedwait(cvp, mp, &to);
     if (err == ETIME) {
          /* timeout, do something */
          break;
     }
```

EXAMPLE 3 Use `cond_reltimedwait()` in a loop to test some condition. *(Continued)*

```
}
(void) mutex_unlock(mp);
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also

`fork(2)`, `mmap(2)`, `setitimer(2)`, `shmop(2)`, `mutex_init(3C)`, `signal(3C)`, `attributes(5)`, `condition(5)`, `mutex(5)`, `standards(5)`

Notes

If more than one thread is blocked on a condition variable, the order in which threads are unblocked is determined by the scheduling policy. When each thread, unblocked as a result of a `cond_signal()` or `cond_broadcast()`, returns from its call to `cond_wait()` or `cond_timedwait()` , the thread owns the mutex with which it called `cond_wait()`, `cond_timedwait()`, or `cond_reltimedwait()`. The thread(s) that are unblocked compete for the mutex according to the scheduling policy and as if each had called `mutex_lock(3C)`.

When `cond_wait()` returns the value of the condition is indeterminate and must be reevaluated.

The `cond_timedwait()` and `cond_reltimedwait()` functions are similar to `cond_wait()`, except that the calling thread will not wait for the condition to become true past the absolute time specified by *abstime* or the relative time specified by *reltime*. Note that `cond_timedwait()` or `cond_reltimedwait()` might continue to block as it trys to reacquire the mutex pointed to by *mp*, which may be locked by another thread. If either `cond_timedwait()` or `cond_reltimedwait()` returns because of a timeout, it returns the error value `ETIME`.

Name confstr – get configurable variables

Synopsis

```
#include <unistd.h>

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

Description The `confstr()` function provides a method for applications to get configuration-defined string values. Its use and purpose are similar to the `sysconf(3C)` function, but it is used where string values rather than numeric values are returned.

The *name* argument represents the system variable to be queried.

If *len* is not `0`, and if *name* has a configuration-defined value, `confstr()` copies that value into the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes, including the terminating null, then `confstr()` truncates the string to *len*–1 bytes and null-terminates the result. The application can detect that the string was truncated by comparing the value returned by `confstr()` with *len*.

If *len* is `0`, `confstr()` still returns the integer value as defined below, but does not return the string.

The `confstr()` function supports the following values for *name*, defined in `<unistd.h>`, for both SPARC and x86:

`_CS_LFS64_CFLAGS`	If `_LFS64_LARGEFILE` is defined in `<unistd.h>`, this value is the set of initial options to be given to the `cc` and `c89` utilities to build an application using the Large File Summit transitional compilation environment (see `lfcompile64(5)`).
`_CS_LFS64_LDFLAGS`	If `_LFS64_LARGEFILE` is defined in `<unistd.h>`, this value is the set of final options to be given to the `cc` and `c89` utilities to build an application using the Large File Summit transitional compilation environment (see `lfcompile64(5)`).
`_CS_LFS64_LIBS`	If `_LFS64_LARGEFILE` is defined in `<unistd.h>`, this value is the set of libraries to be given to the `cc` and `c89` utilities to build an application using the Large File Summit transitional compilation environment (see `lfcompile64(5)`).
`_CS_LFS64_LINTFLAGS`	If `_LFS64_LARGEFILE` is defined in `<unistd.h>`, this value is the set of options to be given to the `lint` utility to check application source using the Large File Summit transitional compilation environment (see `lfcompile64(5)`).

_CS_LFS_CFLAGS	If _LFS_LARGEFILE is defined in <unistd.h>, this value is the set of initial options to be given to the cc and c89 utilities to build an application using the Large File Summit large file compilation environment for 32-bit applications (see lfcompile(5)).
_CS_LFS_LDFLAGS	If _LFS_LARGEFILE is defined in <unistd.h>, this value is the set of final options to be given to the cc and c89 utilities to build an application using the Large File Summit large file compilation environment for 32-bit applications (see lfcompile(5)).
_CS_LFS_LIBS	If _LFS_LARGEFILE is defined in <unistd.h>, this value is the set of libraries to be given to the cc and c89 utilities to build an application using the Large File Summit large file compilation environment for 32-bit applications (see lfcompile(5)).
_CS_LFS_LINTFLAGS	If _LFS_LARGEFILE is defined in <unistd.h>, this value is the set of options to be given to the lint utility to check application source using the Large File Summit large file compilation environment for 32-bit applications (see lfcompile(5)).
_CS_PATH	If the ISO POSIX.2 standard is supported, this is the value for the PATH environment variable that finds all standard utilities. Otherwise the meaning of this value is unspecified.
_CS_POSIX_V6_ILP32_OFF32_CFLAGS	If sysconf(_SC_V6_ILP32_OFF32) returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the c99 utility to build an application using a programming model with 32-bit int, long, pointer, and off_t types.
_CS_POSIX_V6_ILP32_OFF32_LDFLAGS	If sysconf(_SC_V6_ILP32_OFF32) returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the c99 utility to build an application using a programming model with 32-bit int, long, pointer, and off_t types.

`_CS_POSIX_V6_ILP32_OFF32_LIBS`	If `sysconf(_SC_V6_ILP32_OFF32)` returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the `c99` utility to build an application using a programming model with 32-bit `int`, `long`, `pointer`, and `off_t` types.
`_CS_POSIX_V6_ILP32_OFFBIG_CFLAGS`	If `sysconf(_SC_V6_ILP32_OFFBIG)` returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the `c99` utility to build an application using a programming model with 32-bit `int`, `long`, and `pointer` types, and an `off_t` type using at least 64 bits.
`_CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS`	If `sysconf(_SC_V6_ILP32_OFFBIG)` returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the `c99` utility to build an application using a programming model with 32-bit `int`, `long`, and `pointer` types, and an `off_t` type using at least 64 bits.
`_CS_POSIX_V6_ILP32_OFFBIG_LIBS`	If `sysconf(_SC_V6_ILP32_OFFBIG)` returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the `c99` utility to build an application using a programming model with 32-bit `int`, `long`, and `pointer` types, and an `off_t` type using at least 64 bits.
`_CS_POSIX_V6_LP64_OFF64_CFLAGS`	If `sysconf(_SC_V6_LP64_OFF64)` returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the `c99` utility to build an application using a programming model with 64-bit `int`, `long`, `pointer`, and `off_t` types.
`_CS_POSIX_V6_LP64_OFF64_LDFLAGS`	If `sysconf(_SC_V6_LP64_OFF64)` returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the `c99` utility to build an application using a programming model with 64-bit `int`, `long`, `pointer`, and `off_t` types.

_CS_POSIX_V6_LP64_OFF64_LIBS	If sysconf(_SC_V6_LP64_OFF64) returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the c99 utility to build an application using a programming model with 64-bit int, long, pointer, and off_t types.
_CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS	If sysconf(_SC_V6_LPBIG_OFFBIG) returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the c99 utility to build an application using a programming model with an int type using at least 32 bits and long, pointer, and off_t types using at least 64 bits.
_CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS	If sysconf(_SC_V6_LPBIG_OFFBIG) returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the c99 utility to build an application using a programming model with an int type using at least 32 bits and long, pointer, and off_t types using at least 64 bits.
_CS_POSIX_V6_LPBIG_OFFBIG_LIBS	If sysconf(_SC_V6_LPBIG_OFFBIG) returns -1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the c99 utility to build an application using a programming model with an int type using at least 32 bits and long, pointer, and off_t types using at least 64 bits.
_CS_POSIX_V6_WIDTH_RESTRICTED_ENVS	This value is a <newline>-separated list of names of programming environments supported by the implementation in which the widths of the blksize_t, cc_t, mode_t, nfds_t, pid_t, ptrdiff_t, size_t, speed_t, ssize_t, suseconds_t, tcflag_t, useconds_t, wchar_t, and wint_t types are no greater than the width of type long.
_CS_XBS5_ILP32_OFF32_CFLAGS	If sysconf(_SC_XBS5_ILP32_OFF32) returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the cc and c89 utilities to build an application using a programming model with 32-bit int, long, pointer, and off_t types.

`_CS_XBS5_ILP32_OFF32_LDFLAGS`	If `sysconf(_SC_XBS5_ILP32_OFF32)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the `cc` and `c89` utilities to build an application using a programming model with 32-bit `int`, `long`, `pointer`, and `off_t` types.
`_CS_XBS5_ILP32_OFF32_LIBS`	If `sysconf(_SC_XBS5_ILP32_OFF32)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the `cc` and `c89` utilities to build an application using a programming model with 32-bit `int`, `long`, `pointer`, and `off_t` types.
`_CS_XBS5_ILP32_OFF32_LINTFLAGS`	If `sysconf(_SC_XBS5_ILP32_OFF32)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the `lint` utility to check application source using a programming model with 32-bit `int`, `long`, `pointer`, and `off_t` types.
`_CS_XBS5_ILP32_OFFBIG_CFLAGS`	If `sysconf(_SC_XBS5_ILP32_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the `cc` and `c89` utilities to build an application using a programming model with 32-bit `int`, `long`, and `pointer` types, and an `off_t` type using at least 64 bits.
`_CS_XBS5_ILP32_OFFBIG_LDFLAGS`	If `sysconf(SC_XBS5_ILP32_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the `cc` and `c89` utilities to build an application using a programming model with 32-bit `int`, `long`, and `pointer` types, and an `off_t` type using at least 64 bits.
`_CS_XBS5_ILP32_OFFBIG_LIBS`	If `sysconf(_SC_XBS5_ILP32_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the `cc` and `c89` utilities to build an application using a programming model with 32-bit `int`, `long`, and `pointer` types, and an `off_t` type using at least 64 bits.

`_CS_XBS5_ILP32_OFFBIG_LINTFLAGS`	If `sysconf(_SC_XBS5_ILP32_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the `lint` utility to check an application using a programming model with 32-bit `int`, `long`, and `pointer` types, and an `off_t` type using at least 64 bits.

The `confstr()` function supports the following values for *name*, defined in `<unistd.h>`, for SPARC only:

`_CS_XBS5_LP64_OFF64_CFLAGS`	If `sysconf(_SC_XBS5_LP64_OFF64)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the `cc` and `c89` utilities to build an application using a programming model with 64-bit `int`, `long`, `pointer`, and `off_t` types.
`_CS_XBS5_LP64_OFF64_LDFLAGS`	If `sysconf(_SC_XBS5_LP64_OFF64)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the `cc` and `c89` utilities to build an application using a programming model with 64-bit `int`, `long`, `pointer`, and `off_t` types.
`_CS_XBS5_LP64_OFF64_LIBS`	If `sysconf(_SC_XBS5_LP64_OFF64)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the `cc` and `c89` utilities to build an application using a programming model with 64-bit `int`, `long`, `pointer`, and `off_t` types.
`_CS_XBS5_LP64_OFF64_LINTFLAGS`	If `sysconf(_SC_XBS5_LP64_OFF64)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the `lint` utility to check application source using a programming model with 64-bit `int`, `long`, `pointer`, and `off_t` types.
`_CS_XBS5_LPBIG_OFFBIG_CFLAGS`	If `sysconf(_SC_XBS5_LPBIG_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the `cc` and `c89` utilities to build an application using a programming model with an `int` type using at least 32 bits and `long`, `pointer`, and `off_t` types using at least 64 bits.

`_CS_XBS5_LPBIG_OFFBIG_LDFLAGS`	If `sysconf(_SC_XBS5_LPBIG_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the `cc` and `c89` utilities to build an application using a programming model with an `int` type using at least 32 bits and `long`, `pointer`, and `off_t` types using at least 64 bits.
`_CS_XBS5_LPBIG_OFFBIG_LIBS`	If `sysconf(_SC_XBS5_LPBIG_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the `cc` and `c89` utilities to build an application using a programming model with an `int` type using at least 32 bits and `long`, `pointer`, and `off_t` types using at least 64 bits.
`_CS_XBS5_LPBIG_OFFBIG_LINTFLAGS`	If `sysconf(_SC_XBS5_LPBIG_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the `lint` utility to check application source using a programming model with an `int` type using at least 32 bits and `long`, `pointer`, and `off_t` types using at least 64 bits.

Return Values If *name* has a configuration-defined value, the `confstr()` function returns the size of buffer that would be needed to hold the entire configuration-defined value. If this return value is greater than *len*, the string returned in *buf* is truncated.

If *name* is invalid, `confstr()` returns `0` and sets `errno` to indicate the error.

If *name* does not have a configuration-defined value, `confstr()` returns `0` and leaves `errno` unchanged.

Errors The `confstr()` function will fail if:

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Mt-Safe
Standard	See `standards(5)`.

See Also pathconf(2), sysconf(3C), attributes(5), lfcompile(5), lfcompile64(5), standards(5)

Name crypt – string encoding function

Synopsis

```
#include <crypt.h>

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

Standard conforming

```
#include <unistd.h>

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

Description The `crypt()` function encodes strings suitable for secure storage as passwords. It generates the password hash given the *key* and *salt*.

The *key* argument is the plain text password to be encrypted.

If the first character of *salt* is "$", `crypt()` uses `crypt.conf(4)` to determine which shared module to load for the encryption algorithm. The algorithm name `crypt()` uses to search in `crypt.conf` is the string between the first and second "$", or between the first "$" and first "," if a "," comes before the second "$".

If the first character of *salt* is not "$", the algorithm described on `crypt_unix(5)` is used.

Return Values Upon successful completion, `crypt()` returns a pointer to the encoded string. Otherwise it returns a null pointer and sets `errno` to indicate the error.

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

Errors The `crypt()` function will fail if:

`EINVAL` An entry in `crypt.conf` is invalid.

`ELIBACC` The required shared library was not found.

`ENOMEM` There is insufficient memory to generate the hash.

`ENOSYS` The functionality is not supported on this system.

Usage The values returned by this function might not be portable among standard-conforming systems. See `standards(5)`.

Applications should not use `crypt()` to store or verify user passwords but should use the functions described on `pam(3PAM)` instead.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also passwd(1), crypt_genhash_impl(3C), crypt_gensalt(3C), crypt_gensalt_impl(3C), getpassphrase(3C), pam(3PAM), passwd(4), policy.conf(4), attributes(5), crypt_unix(5), standards(5)

Name crypt_genhash_impl – generate encrypted password

Synopsis

```
#include <crypt.h>

char *crypt_genhash_impl(char *ctbuffer, size_t ctbufflen,
     const char *plaintext, const char *salt, const char **params);
```

Description The `crypt_genhash_impl()` function is called by `crypt(3C)` to generate the encrypted password *plaintext*.

The *ctbuffer* argument is a pointer to an MT-safe buffer of *ctbufflen* size that is used to return the result.

The *salt* argument is the salt used in encoding.

The *params* argument is an *argv*-like null-terminated vector of type `char *`. The first element of *params* represents the mechanism token name from `crypt.conf(4)`. The remaining elements of *params* represent strings of the form `<parameter>[=<value>]` to allow passing in additional information from the `crypt.conf` entry, such as specifing rounds information "`rounds=4096`".

The `crypt_genhash_impl()` function must not `free(3C)` *ctbufflen* on error.

Return Values Upon successful completion, `crypt_genhash_impl()` returns a pointer to the encoded version of *plaintext*. Otherwise a null pointer is returned and `errno` is set to indicate the error.

Errors The `crypt_genhash_impl()` function will fail if:

`EINVAL` The configuration file `crypt.conf` contains an invalid entry.

`ELIBACC` The required shared library was not found.

`ENOMEM` There is insufficient memory to perform hashing.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `passwd(1)`, `crypt(3C)`, `crypt_gensalt_impl(3C)`, `free(3C)`, `getpassphrase(3C)`, `crypt.conf(4)`, `passwd(4)`, `attributes(5)`

Name crypt_gensalt – generate salt string for string encoding

Synopsis

```
#include <crypt.h>

char *crypt_gensalt(const char *oldsalt, const struct passwd *userinfo);
```

Description The `crypt_gensalt()` function generates the salt string required by `crypt(3C)`.

If *oldsalt* is `NULL`, `crypt_gensalt()` uses the algorithm defined by `CRYPT_DEFAULT` in `/etc/security/policy.conf`. See `policy.conf(4)`.

If *oldsalt* is non-null, `crypt_gensalt()` determines if the algorithm specified by *oldsalt* is allowable by checking the `CRYPT_ALGORITHMS_ALLOW` and `CRYPT_ALGORITHMS_DEPRECATE` variables in `/etc/security/policy.conf`. If the algorithm is allowed, `crypt_gensalt()` loads the appropriate shared library and calls `crypt_gensalt_impl(3C)`. If the algorithm is not allowed or there is no entry for it in `crypt.conf`, `crypt_gensalt()` uses the default algorithm.

The mechanism just described provides a means to migrate users to new password hashing algorithms when the password is changed.

Return Values Upon successful completion, `crypt_gensalt()` returns a pointer to the new salt. Otherwise a null pointer is returned and `errno` is set to indicate the error.

Errors The `crypt_gensalt()` function will fail if:

`EINVAL` The configuration file `crypt.conf` contains an invalid entry.

`ELIBACC` The required shared library was not found.

`ENOMEM` There is insufficient memory to perform hashing.

Usage The value returned by `crypt_gensalt()` points to a null-terminated string. The caller of `crypt_gensalt()` is responsible for calling `free(3C)`.

Applications dealing with user authentication and password changing should not call `crypt_gensalt()` directly but should instead call the appropriate `pam(3PAM)` functions.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `passwd(1)`, `crypt(3C)`, `crypt_genhash_impl(3C)`, `crypt_gensalt_impl(3C)`, `getpassphrase(3C)`, `malloc(3C)`, `pam(3PAM)`, `crypt.conf(4)`, `passwd(4)`, `policy.conf(4)`, `attributes(5)`

Name crypt_gensalt_impl – generate salt for password encryption

Synopsis

```
#include <crypt.h>

char *crypt_gensalt_impl(char *gsbuffer, size_t gsbufflen,
     const char *oldsalt, const struct passwd *userinfo,
     const char **params);
```

Description The `crypt_gensalt_impl()` function is called by `crypt_gensalt(3C)` to generate the salt for password encryption.

The *gsbuffer* argument is a pointer to an MT-safe buffer of size *gsbufflen*.

The *oldsalt* and *userinfo* arguments are passed unchanged from `crypt_gensalt(3C)`.

The *params* argument is an *argv*-like null terminated vector of type `char *`. The first element of *params* represents the mechanism token name from `crypt.conf(4)`. The remaining elements of *params* represent strings of the form `<parameter>[=<value>]` to allow passing in additional information from the `crypt.conf` entry, such as specifying rounds information "`rounds=4096`".

The value returned by `crypt_gensalt_impl()` points to a thread-specific buffer to be freed by the caller of `crypt_gensalt(3C)` after calling `crypt(3C)`.

Return Values Upon successful completion, `crypt_gensalt_impl()` returns a pointer to the new salt. Otherwise a null pointer is returned and `errno` is set to indicate the error.

Errors The `crypt_gensalt_impl()` function will fail if:

`EINVAL` The configuration file `crypt.conf` contains an invalid entry.

`ELIBACC` The required crypt shared library was not found.

`ENOMEM` There is insufficient memory to perform hashing.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `passwd(1)`, `crypt(3C)`, `crypt_genhash_impl(3C)`, `crypt_gensalt(3C)`, `getpassphrase(3C)`, `crypt.conf(4)`, `passwd(4)`, `attributes(5)`

Name cset, csetlen, csetcol, csetno, wcsetno – get information on EUC codesets

Synopsis

```
#include <euc.h>

int csetlen(int codeset);

int csetcol(int codeset);

int csetno(unsigned char c);

#include <widec.h>

int wcsetno(wchar_t pc);
```

Description Both `csetlen()` and `csetcol()` take a code set number *codeset*, which must be 0, 1, 2, or 3. The `csetlen()` function returns the number of bytes needed to represent a character of the given Extended Unix Code (EUC) code set, excluding the single-shift characters SS2 and SS3 for codesets 2 and 3. The `csetcol()` function returns the number of columns a character in the given EUC code set would take on the display.

The `csetno()` function is implemented as a macro that returns a codeset number (0, 1, 2, or 3) for the EUC character whose first byte is *c*. For example,

```
#include<euc.h>
 . . .
x+=csetcol(csetno(c));
```

increments a counter "x" (such as the cursor position) by the width of the character whose first byte is *c*.

The `wcsetno()` function is implemented as a macro that returns a codeset number (0, 1, 2, or 3) for the given process code character *pc*. For example,

```
#include<euc.h>
#include<widec.h>
 . . .
x+=csetcol(wcsetno(pc));
```

increments a counter "x" (such as the cursor position) by the width of the Process Code character *pc*.

Usage These functions work only for the EUC locales.

The `cset()`, `csetlen()`, `csetcol()`, `csetno()`, and `wcsetno()` functions can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe with exceptions

See Also setlocale(3C) euclen(3C), attributes(5)

Name ctermid, ctermid_r – generate path name for controlling terminal

Synopsis

```
#include <stdio.h>

char *ctermid(char *s);

char *ctermid_r(char *s);
```

Description

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

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

`ctermid_r()` The `ctermid_r()` function behaves as `ctermid()` except that if *s* is a null pointer, the function returns `NULL`.

Usage The difference between `ctermid()` and `ttyname(3C)` is that `ttyname()` must be passed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while `ctermid()` returns a string (`/dev/tty`) that will refer to the terminal if used as a file name. The `ttyname()` function is useful only if the process already has at least one file open to a terminal.

The `ctermid()` function is unsafe in multithreaded applications. The `ctermid_r()` function is MT-Safe and should be used instead.

When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should be used only with multithreaded applications.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	`ctermid()` is Standard
MT-Level	`ctermid()` is Unsafe; `ctermid_r()` is MT-Safe

See Also `ttyname(3C)`, `attributes(5)`

Name ctime, ctime_r, localtime, localtime_r, gmtime, gmtime_r, asctime, asctime_r, tzset – convert date and time to string

Synopsis

```
#include <time.h>

char *ctime(const time_t *clock);

struct tm *localtime(const time_t *clock);

struct tm *gmtime(const time_t *clock);

char *asctime(const struct tm *tm);

extern time_t timezone, altzone;
extern int daylight;
extern char *tzname[2];

void tzset(void);

char *ctime_r(const time_t *clock, char *buf, int buflen);

struct tm *localtime_r(const time_t *restrict clock,
     struct tm *restrict res);

struct tm *gmtime_r(const time_t *restrict clock,
     struct tm *restrict res);

char *asctime_r(const struct tm *restrict tm, char *restrict buf,
     int buflen);
```

Standard conforming

```
cc [ flag... ] file... -D_POSIX_PTHREAD_SEMANTICS [ library... ]

char *ctime_r(const time_t *clock, char *buf);

char *asctime_r(const struct tm *tm, char *buf);
```

Description The `ctime()` function converts the time pointed to by *clock*, representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970), to local time in the form of a 26-character string, as shown below. Time zone and daylight savings corrections are made before string generation. The fields are in constant width:

Fri Sep 13 00:00:00 1986\n\0

The `ctime()` function is equivalent to:

```
asctime(localtime(clock))
```

The `ctime()`, `asctime()`, `gmtime()`, and `localtime()` functions return values in one of two thread-specific data objects: a broken-down time structure and an array of `char`. Execution of any of the functions can overwrite the information returned in either of these objects by any of the other functions executed by the same thread.

The `ctime_r()` function has the same functionality as `ctime()` except that the caller must supply a buffer *buf* with length *buflen* to store the result; *buf* must be at least 26 bytes. The standard-conforming `ctime_r()` function does not take a *buflen* parameter.

The `localtime()` and `gmtime()` functions return pointers to `tm` structures (see below). The `localtime()` function corrects for the main time zone and possible alternate ("daylight savings") time zone; the `gmtime()` function converts directly to Coordinated Universal Time (UTC), which is what the UNIX system uses internally.

The `localtime_r()` and `gmtime_r()` functions have the same functionality as `localtime()` and `gmtime()` respectively, except that the caller must supply a buffer *res* to store the result.

The `asctime()` function converts a `tm` structure to a 26-character string, as shown in the previous example, and returns a pointer to the string.

The `asctime_r()` function has the same functionality as `asctime()` except that the caller must supply a buffer *buf* with length *buflen* for the result to be stored. The *buf* argument must be at least 26 bytes. The standard-conforming `asctime_r()` function does not take a *buflen* parameter. The `asctime_r()` function returns a pointer to *buf* upon success. In case of failure, `NULL` is returned and `errno` is set.

Declarations of all the functions and externals, and the `tm` structure, are in the `<time.h>` header. The members of the `tm` structure are:

```
int   tm_sec;    /* seconds after the minute — [0, 60] */
                 /* for leap seconds */
int   tm_min;    /* minutes after the hour — [0, 59] */
int   tm_hour;   /* hour since midnight — [0, 23] */
int   tm_mday;   /* day of the month — [1, 31] */
int   tm_mon;    /* months since January — [0, 11] */
int   tm_year;   /* years since 1900 */
int   tm_wday;   /* days since Sunday — [0, 6] */
int   tm_yday;   /* days since January 1 — [0, 365] */
int   tm_isdst;  /* flag for alternate daylight savings time */
```

The value of `tm_isdst` is positive if daylight savings time is in effect, zero if daylight savings time is not in effect, and negative if the information is not available. Previously, the value of `tm_isdst` was defined as non-zero if daylight savings was in effect.

The external `time_t` variable `altzone` contains the difference, in seconds, between Coordinated Universal Time and the alternate time zone. The external variable `timezone` contains the difference, in seconds, between UTC and local standard time. The external variable `daylight` indicates whether time should reflect daylight savings time. Both `timezone` and `altzone` default to 0 (UTC). The external variable `daylight` is non-zero if an alternate time zone exists. The time zone names are contained in the external variable `tzname`, which by default is set to:

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

These functions know about the peculiarities of this conversion for various time periods for the U.S. (specifically, the years 1974, 1975, and 1987). They start handling the new daylight savings time starting with the first Sunday in April, 1987.

The `tzset()` function uses the contents of the environment variable `TZ` to override the value of the different external variables. It is called by `asctime()` and can also be called by the user. If `TZ` is not specified or has an invalid setting, `tzset()` uses `GMT0`. See `environ(5)` for a description of the `TZ` environment variable.

Starting and ending times are relative to the current local time zone. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be 2 AM. The effects of `tzset()` change the values of the external variables `timezone`, `altzone`, `daylight`, and `tzname`.

Note that in most installations, `TZ` is set to the correct value by default when the user logs on, using the local `/etc/default/init` file (see `TIMEZONE(4)`).

Return Values Upon successful completion, the `gmtime()` and `localtime()` functions return a pointer to a `struct tm`. If an error is detected, `gmtime()` and `localtime()` return a null pointer.

Upon successful completion, the `gmtime_r()` and `localtime_r()` functions return the address of the structure pointed to by the *res* argument. If an error is detected, `gmtime_r()` and `localtime_r()` return a null pointer and set `errno` to indicate the error.

Errors The `ctime_r()` and `asctime_r()` functions will fail if:

`ERANGE` The length of the buffer supplied by the caller is not large enough to store the result.

The `gmtime()`, `gmtime_r()`, `localtime()`, and `localtime_r()` functions will fail if:

`EOVERFLOW` The result cannot be represented.

Usage These functions do not support localized date and time formats. The `strftime(3C)` function can be used when localization is required.

The `localtime()`, `localtime_r()`, `gmtime()`, `gmtime_r()`, `ctime()`, and `ctime_r()` functions assume Gregorian dates. Times before the adoption of the Gregorian calendar will not match historial records.

Examples **EXAMPLE 1** Examples of the `tzset()` function.

The `tzset()` function scans the contents of the environment variable and assigns the different fields to the respective variable. For example, the most complete setting for New Jersey in 1986 could be:

```
EST5EDT4,116/2:00:00,298/2:00:00
```

EXAMPLE 1 Examples of the `tzset()` function. *(Continued)*

or simply

```
EST5EDT
```

An example of a southern hemisphere setting such as the Cook Islands could be

```
KDT9:30KST10:00,63/5:00,302/20:00
```

In the longer version of the New Jersey example of TZ, `tzname[0]` is EST, `timezone` is set to 5*60*60, `tzname[1]` is EDT, `altzone` is set to 4*60*60, the starting date of the alternate time zone is the 117th day at 2 AM, the ending date of the alternate time zone is the 299th day at 2 AM (using zero-based Julian days), and `daylight` is set positive. Starting and ending times are relative to the current local time zone. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be 2 AM. The effects of `tzset()` are thus to change the values of the external variables `timezone`, `altzone`, `daylight`, and `tzname`. The `ctime()`, `localtime()`, `mktime()`, and `strftime()` functions also update these external variables as if they had called `tzset()` at the time specified by the `time_t` or `struct tm` value that they are converting.

Bugs The `zoneinfo` timezone data files do not transition past Tue Jan 19 03:14:07 2038 UTC. Therefore for 64-bit applications using `zoneinfo` timezones, calculations beyond this date might not use the correct offset from standard time, and could return incorrect values. This affects the 64-bit version of `localtime()`, `localtime_r()`, `ctime()`, and `ctime_r()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions are safe to use in multithread applications because they employ thread-specific data. However, their use is discouraged because standards do not require them to be thread-safe. The `asctime_r()` and `gmtime_r()` functions are MT-Safe. The `ctime_r()`, `localtime_r()`, and `tzset()` functions are MT-Safe in multithread applications, as long as no user-defined function directly modifies one of the following variables: `timezone`, `altzone`, `daylight`, and `tzname`. These four variables are not MT-Safe to access. They are modified by the `tzset()` function in an MT-Safe manner. The `mktime()`, `localtime_r()`, and `ctime_r()` functions call `tzset()`.

See Also time(2), Intro(3), getenv(3C), mktime(3C), printf(3C), putenv(3C), setlocale(3C), strftime(3C), TIMEZONE(4), attributes(5), environ(5), standards(5)

Notes When compiling multithreaded programs, see Intro(3).

The return values for `asctime()`, `ctime()`, `gmtime()`, and `localtime()` point to thread-specific data whose content is overwritten by each call by the same thread.

Setting the time during the interval of change from `timezone` to `altzone` or vice versa can produce unpredictable results. The system administrator must change the Julian start and end days annually.

If `tzset()` has previously evaluated the timezone identified by the value of the `TZ` environment variable, `tzset()` can reuse the previous settings of the external variables `altzone`, `daylight`, `timezone`, and `tzname[]` associated with that timezone.

Solaris 2.4 and earlier releases provided definitions of the `ctime_r()`, `localtime_r()`, `gmtime_r()`, and `asctime_r()` functions as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface for `ctime_r()` and `asctime_r()`. Support for the Draft 6 interface is provided for compatibility only and might not be supported in future releases. New applications and libraries should use the standard-conforming interface.

For POSIX.1c-conforming applications, the `_POSIX_PTHREAD_SEMANTICS` and `_REENTRANT` flags are automatically turned on by defining the `_POSIX_C_SOURCE` flag with a value >= 199506L.

In Solaris 10, `gmtime()`, `gmtime_r()`, `localtime()`, and `localtime_r()` were updated to return a null pointer if an error is detected. This change was based on the SUSv3 specification. See standards(5).

Name ctype, isalpha, isalnum, isascii, isblank, iscntrl, isdigit, islower, isprint, isspace, isupper, ispunct, isgraph, isxdigit – character handling

Synopsis

```
#include <ctype.h>

int isalpha(int c);

int isalnum(int c);

int isascii(int c);

int isblank(int c);

int iscntrl(int c);

int isdigit(int c);

int isgraph(int c);

int islower(int c);

int isprint(int c);

int ispunct(int c);

int isspace(int c);

int isupper(int c);

int isxdigit(int c);
```

Description These macros classify character-coded integer values. Each is a predicate returning non-zero for true, `0` for false. The behavior of these macros, except `isascii()`, is affected by the current locale (see `setlocale(3C)`). To modify the behavior, change the `LC_TYPE` category in `setlocale()`, that is, `setlocale(LC_CTYPE, `*newlocale*`)`. In the "C" locale, or in a locale where character type information is not defined, characters are classified according to the rules of the US-ASCII 7-bit coded character set.

The `isascii()` macro is defined on all integer values. The rest are defined only where the argument is an `int`, the value of which is representable as an `unsigned char`, or `EOF`, which is defined by the `<stdio.h>` header and represents end-of-file.

Functions exist for all the macros defined below. To get the function form, the macro name must be undefined (for example, `#undef isdigit`).

For macros described with `Default` and `Standard conforming` versions, standard-conforming behavior is provided for standard-conforming applications (see `standards(5)`) and for applications that define `__XPG4_CHAR_CLASS__` before including `<ctype.h>`.

Default `isalpha()` Tests for any character for which `isupper()` or `islower()` is true.

Standard conforming	`isalpha()`	Tests for any character for which `isupper()` or `islower()` is true, or any character that is one of the current locale-defined set of characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, or `isspace()` is true. In "C" locale, `isalpha()` returns true only for the characters for which `isupper()` or `islower()` is true.
	`isalnum()`	Tests for any character for which `isalpha()` or `isdigit()` is true (letter or digit).
	`isascii()`	Tests for any ASCII character, code between `0` and `0177` inclusive.
	`isblank()`	Tests whether *c* is a character of class blank in the current locale. This macro/function is not available to applications conforming to standards prior to SUSv3. See `standards(5)`
	`iscntrl()`	Tests for any "control character" as defined by the character set.
	`isdigit()`	Tests for any decimal-digit character.
Default	`isgraph()`	Tests for any character for which `ispunct()`, `isupper()`, `islower()`, and `isdigit()` is true.
Standard conforming	`isgraph()`	Tests for any character for which `isalnum()` and `ispunct()` are true, or any character in the current locale-defined "graph" class which is neither a space (" ") nor a character for which `iscntrl()` is true.
	`islower()`	Tests for any character that is a lower-case letter or is one of the current locale-defined set of characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, `isspace()`, or `isupper()` is true. In the "C" locale, `islower()` returns true only for the characters defined as lower-case ASCII characters.
Default	`isprint()`	Tests for any character for which `ispunct()`, `isupper()`, `islower()`, `isdigit()`, and the space character (" ") is true.
Standard conforming	`isprint()`	Tests for any character for which `iscntrl()` is false, and `isalnum()`, `isgraph()`, `ispunct()`, the space character (" "), and the characters in the current locale-defined "print" class are true.
	`ispunct()`	Tests for any printing character which is neither a space (" ") nor a character for which `isalnum()` or `iscntrl()` is true.
	`isspace()`	Tests for any space, tab, carriage-return, newline, vertical-tab or form-feed (standard white-space characters) or for one of the current locale-defined set of characters for which `isalnum()` is false. In the "C" locale, `isspace()` returns true only for the standard white-space characters.
	`isupper()`	Tests for any character that is an upper-case letter or is one of the current locale-defined set of characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, `isspace()`, or `islower()` is true. In the "C" locale, `isupper()` returns true only for the characters defined as upper-case ASCII characters.

Default	`isxdigit()`	Tests for any hexadecimal-digit character (`[0–9]`, `[A–F]`, or `[a–f]`).
Standard conforming	`isxdigit()`	Tests for any hexadecimal-digit character (`[0–9]`, `[A–F]`, or `[a–f]` or the current locale-defined sets of characters representing the hexadecimal digits `10` to `15` inclusive). In the "C" locale, only `0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f` are included.

Return Values If the argument to any of the character handling macros is not in the domain of the function, the result is undefined. Otherwise, the macro or function returns non-zero if the classification is `TRUE` and `0` if the classification is `FALSE`.

Usage These macros or functions can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `setlocale(3C)`, `stdio(3C)`, `ascii(5)`, `environ(5)`, `standards(5)`

Name cuserid – get character login name of the user

Synopsis

```
#include <stdio.h>

char *cuserid(char *s);
```

Description The `cuserid()` function generates a character-string representation of the login name under which the owner of the current process is logged in. If *s* is a null pointer, this representation is generated in an internal static area whose address is returned. Otherwise, *s* is assumed to point to an array of at least `L_cuserid` characters; the representation is left in this array. The constant `L_cuserid` is defined in the <`stdio.h`> header.

In multithreaded applications, the caller must always supply an array *s* for the return value.

Return Values If the login name cannot be found, `cuserid()` returns a null pointer. If *s* is not a null pointer, the null character '`\0`' will be placed at *s*[`0`].

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `getlogin(3C)`, `getpwnam(3C)`, `attributes(5)`

Name daemon – basic daemonization function

Synopsis

```
#include <stdlib.h>

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

Description The `daemon()` function provides a means for applications to run in the background.

This function ensures that the process calling this function:

- runs in the background
- detaches from the controlling terminal
- forms a new process group
- is not a session group leader.

The arguments to this function are treated as boolean variables and are evaluated using negative logic.

If the *nochdir* argument is zero the working directory will be changed to the root directory (/); otherwise it will not be.

If the *noclose* argument is zero the descriptors 0, 1, and 2 (normally corresponding to standard input, output and error output, depending on the application) will be redirected to `/dev/null`; otherwise they will not be.

Return Values Upon successful completion, `daemon()` returns 0. Otherwise it returns -1 and sets `errno` to the values specified for `fork(2)`, `setsid(2)`, `open(2)`, and `dup(2)`.

If `daemon()` is called with *noclose* set to 0 and fails to redirect descriptors 0, 1, and 2 to `/dev/null`, those descriptors are not guaranteed to be the same as before the call.

Examples **EXAMPLE 1** Using daemon to run a process in the background.

The `main()` function of a network server could look like this:

```
int background;    /* background flag */

/* Load and verify the configuration. */

/* Go into background. */
if (background && daemon(0, 0) < 0)
        err(1, "daemon");

/* Process requests here. */
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also Intro(2), dup(2), fork(2), open(2), setsid(2), attributes(5)

Name decimal_to_floating, decimal_to_single, decimal_to_double, decimal_to_extended, decimal_to_quadruple – convert decimal record to floating-point value

Synopsis

```
#include <floatingpoint.h>

void decimal_to_single(single *px, decimal_mode *pm,
     decimal_record *pd, fp_exception_field_type *ps);

void decimal_to_double(double *px, decimal_mode *pm,
     decimal_record *pd, fp_exception_field_type *ps);

void decimal_to_extended(extended *px, decimal_mode *pm,
     decimal_record *pd, fp_exception_field_type *ps);

void decimal_to_quadruple(quadruple *px, decimal_mode *pm,
     decimal_record *pd, fp_exception_field_type *ps);
```

Description These functions convert the decimal record **pd* to a floating-point value **px* observing the rounding direction specified in **pm* and setting **ps* to reflect any floating-point exceptions that occur.

When *pd*->`fpclass` is `fp_zero`, `fp_infinity`, `fp_quiet`, or `fp_signaling`, **px* is set to zero, infinity, a quiet NaN, or a signaling NaN, respectively, with the sign indicated by *pd*->`sign`. All other fields in **pd* are ignored.

When *pd*->`fpclass` is `fp_normal` or `fp_subnormal`, *pd*->`ds` must contain a null-terminated string of one or more ASCII digits representing a non-zero integer *m*, and *pd*->`ndigits` must be equal to the length of this string. Then **px* is set to a correctly rounded approximation to

```
–1**(pd->sign) * m * 10**(pd->exponent)
```

pd->`more` can be set to a non-zero value to indicate that insignificant trailing digits were omitted from *pd*->`ds`. In this case, *m* is replaced by *m* + *delta* in the expression above, where *delta* is some tiny positive fraction.

The converted value is rounded according to the rounding direction specified in *pm*->`rd`. *pm*->`df` and *pm*->`ndigits` are not used.

On exit, **ps* contains a bitwise OR of flags corresponding to any floating-point exceptions that occurred. The only possible exceptions are underflow, overflow, and inexact. If no floating-point exceptions occurred, **ps* is set to zero.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also scanf(3C), string_to_decimal(3C), strtod(3C), attributes(5)

Name difftime – computes the difference between two calendar times

Synopsis

```
#include <time.h>

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

Description The `difftime()` function computes the difference between two calendar times.

Return Values The `difftime()` functions returns the difference *(time1-time0)* expressed in seconds as a `double`.

Usage The `difftime()` function is provided because there are no general arithmetic properties defined for type `time_t`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `ctime(3C)`, `attributes(5)`, `standards(5)`

Name directio – provide advice to file system

Synopsis

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

int directio(int fildes, int advice);
```

Description The `directio()` function provides advice to the system about the expected behavior of the application when accessing the data in the file associated with the open file descriptor *fildes*. The system uses this information to help optimize accesses to the file's data. The `directio()` function has no effect on the semantics of the other operations on the data, though it may affect the performance of other operations.

The *advice* argument is kept per file; the last caller of `directio()` sets the *advice* for all applications using the file associated with *fildes*.

Values for *advice* are defined in `<sys/fcntl.h>`.

`DIRECTIO_OFF` Applications get the default system behavior when accessing file data.

When an application reads data from a file, the data is first cached in system memory and then copied into the application's buffer (see `read(2)`). If the system detects that the application is reading sequentially from a file, the system will asynchronously "read ahead" from the file into system memory so the data is immediately available for the next `read(2)` operation.

When an application writes data into a file, the data is first cached in system memory and is written to the device at a later time (see `write(2)`). When possible, the system increases the performance of `write(2)` operations by cacheing the data in memory pages. The data is copied into system memory and the `write(2)` operation returns immediately to the application. The data is later written asynchronously to the device. When possible, the cached data is "clustered" into large chunks and written to the device in a single write operation.

The system behavior for `DIRECTIO_OFF` can change without notice.

`DIRECTIO_ON` The system behaves as though the application is not going to reuse the file data in the near future. In other words, the file data is not cached in the system's memory pages.

When possible, data is read or written directly between the application's memory and the device when the data is accessed with `read(2)` and `write(2)` operations. When such transfers are not possible, the system switches back to the default behavior, but just for that operation. In general, the transfer is possible when the application's buffer is aligned on a

two-byte (short) boundary, the offset into the file is on a device sector boundary, and the size of the operation is a multiple of device sectors.

This advisory is ignored while the file associated with *fildes* is mapped (see `mmap(2)`).

The system behavior for `DIRECTIO_ON` can change without notice.

Return Values Upon successful completion, `directio()` returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `directio()` function will fail if:

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

`ENOTTY` The *fildes* argument is not associated with a file system that accepts advisory functions.

`EINVAL` The value in *advice* is invalid.

Usage Small sequential I/O generally performs best with `DIRECTIO_OFF`.

Large sequential I/O generally performs best with `DIRECTIO_ON`, except when a file is sparse or is being extended and is opened with `O_SYNC` or `O_DSYNC` (see `open(2)`).

The `directio()` function is supported for the NFS and UFS file system types (see `fstyp(1M)`).

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `fstyp(1M)`, `mmap(2)`, `open(2)`, `read(2)`, `write(2)`, `fcntl.h(3HEAD)`, `attributes(5)`

Warnings Switching between `DIRECTIO_OFF` and `DIRECTIO_ON` can slow the system because each switch to `DIRECTIO_ON` might entail flushing the file's data from the system's memory.

Name dirfd – get directory stream file descriptor

Synopsis

```
#include <dirent.h>

int dirfd(DIR *dir);
```

Description The `dirfd()` function returns the file descriptor associated with the directory stream *dir*.

This file descriptor is the one used internally by the directory stream operations. See `opendir(3C)`, `closedir(3C)`, `readdir(3C)`, `rewinddir(3C)`, `seekdir(3C)`, `telldir(3C)`. The file descriptor is automatically closed when `closedir()` is called for the directory stream *dir* or when one of the `exec` functions is called. See `exec(2)`.

The file descriptor can safely be used only by functions that do not depend on or alter the file position, such as `fstat(2)` and `fchdir(2)`. Closing the file descriptor with `close(2)` or modifying the file position by means other than the directory stream operations listed above causes undefined behavior to occur when one of the directory stream operations is subsequently called with the directory stream *dir*.

Return Values Upon successful completion, the `dirfd()` function returns an open file descriptor for the directory associated with the directory stream *dir*.

Errors There are no defined error returns. Passing an invalid directory stream as an argument to the `dirfd()` function results in undefined behavior.

Usage The `dirfd()` function is intended to be used to obtain a file descriptor for use with the `fchdir()` function.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `close(2)`, `exec(2)`, `fchdir(2)`, `fstat(2)`, `closedir(3C)`, `opendir(3C)`, `readdir(3C)`, `rewinddir(3C)`, `seekdir(3C)`, `telldir(3C)`, `attributes(5)`

Name dirname – report the parent directory name of a file path name

Synopsis

```
#include <libgen.h>

char *dirname(char *path);
```

Description The `dirname()` function takes a pointer to a character string that contains a pathname, and returns a pointer to a string that is a pathname of the parent directory of that file. Trailing '/' characters in the path are not counted as part of the path.

If *path* does not contain a '/', then `dirname()` returns a pointer to the string "." . If *path* is a null pointer or points to an empty string, `dirname()` returns a pointer to the string "." .

Return Values The `dirname()` function returns a pointer to a string that is the parent directory of *path*. If *path* is a null pointer or points to an empty string, a pointer to a string "." is returned.

Errors No errors are defined.

Examples **EXAMPLE 1** Changing the Current Directory to the Parent Directory.

The following code fragment reads a pathname, changes the current working directory to the parent directory of the named file (see `chdir(2)`), and opens the file.

```
char path[[MAXPATHLEN], *pathcopy;
int fd;
fgets(path, MAXPATHLEN, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

EXAMPLE 2 Sample Input and Output Strings for `dirname()`.

In the following table, the input string is the value pointed to by *path*, and the output string is the return value of the `dirname()` function.

Input String	Output String
“/usr/lib"”	“/usr”
“/usr/”	“/”
“usr”	“/”
“/”	“/”
“.”	“.”
“..”	“.”

Usage The `dirname()` function modifies the string pointed to by *path*.

The `dirname()` and `basename(3C)` functions together yield a complete pathname. The expression `dirname(`*path*`)` obtains the pathname of the directory where `basename(`*path*`)` is found.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `basename(1)`, `chdir(2)`, `basename(3C)`, `attributes(5)`, `standards(5)`

Name div, ldiv, lldiv – compute the quotient and remainder

Synopsis

```
#include <stdlib.h>

div_t div(int numer, int denom);

ldiv_t ldiv(long int numer, long int denom);

lldiv_t lldiv(long long numer, long long denom);
```

Description The `div()` function computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. It provides a well-defined semantics for the signed integral division and remainder operations, unlike the implementation-defined semantics of the built-in operations. The sign of the resulting quotient is that of the algebraic quotient, and if the division is inexact, the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, *quotient* * *denom* + *remainder* will equal *numer*.

The `ldiv()` and `lldiv()` functions are similar to `div()`, except that the arguments and the members of the returned structure are different. The `ldiv()` function returns a structure of type `ldiv_t` and has type `long int`. The `lldiv()` function returns a structure of type `lldiv_t` and has type `long long`.

Return Values The `div()` function returns a structure of type `div_t`, comprising both the quotient and remainder:

```
int   quot;   /*quotient*/
int   rem;    /*remainder*/
```

The `ldiv()` function returns a structure of type `ldiv_t` and `lldiv()` returns a structure of type `lldiv_t`, comprising both the quotient and remainder:

```
long int   quot;   /*quotient*/
long int   rem;    /*remainder*/
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `attributes(5)`, `standards(5)`

Name dladdr, dladdr1 – translate address to symbolic information

Synopsis

```
#include <dlfcn.h>

int dladdr(void *address, Dl_info_t *dlip);

int dladdr1(void *address, Dl_info_t *dlip, void **info, int flags);
```

Description The `dladdr()` and `dladdr1()` functions determine if the specified *address* is located within one of the mapped objects that make up the current applications address space. An address is deemed to fall within a mapped object when it is between the base address, and the *_end* address of that object. See NOTES. If a mapped object fits this criteria, the symbol table made available to the runtime linker is searched to locate the nearest symbol to the specified address. The nearest symbol is one that has a value less than or equal to the required address.

The `Dl_info_t` structure must be preallocated by the user. The structure members are filled in by `dladdr()`, based on the specified *address*. The `Dl_info_t` structure includes the following members:

```
const char    *dli_fname;
void          *dli_fbase;
const char    *dli_sname;
void          *dli_saddr;
```

The `Dl_info_t` members provide the following information.

`dli_fname` Contains a pointer to the filename of the containing object.

`dli_fbase` Contains the base address of the containing object.

`dli_sname` Contains a pointer to the symbol name that is nearest to the specified address. This symbol either represents the exact address that was specified, or is the nearest symbol with a lower address.

`dli_saddr` Contains the actual address of the symbol pointed to by `dli_sname`.

The `dladdr1()` function provides for addition information to be returned as specified by the *flags* argument:

`RTLD_DL_SYMENT` Obtain the ELF symbol table entry for the matched symbol. The *info* argument points to a symbol pointer as defined in `<sys/elf.h>` (`Elf32_Sym` ***info* or `Elf64_Sym` ***info*). Most of the information found in an ELF symbol can only be properly interpreted by the runtime linker. However, there are two fields that contain information useful to the caller of `dladdr1()`: The `st_size` field contains the size of the referenced item. The `st_info` field provides symbol type and binding attributes. See the *Linker and Libraries Guild* for more information.

`RTLD_DL_LINKMAP` Obtain the `Link_map` for the matched file. The *info* argument points to a `Link_map` pointer as defined in <`sys/link.h`> (`Link_map` ***info*).

Return Values If the specified *address* cannot be matched to a mapped object, a `0` is returned. Otherwise, a non-zero return is made and the associated `Dl_info_t` elements are filled.

Usage The `dladdr()` and `dladdr1()` functions are one of a family of functions that give the user direct access to the dynamic linking facilities. These facilities are available to dynamically-linked processes only. See *Linker and Libraries Guide*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `ld(1)`, `dlclose(3C)`, `dldump(3C)`, `dlerror(3C)`, `dlopen(3C)`, `dlsym(3C)`, `attributes(5)`

Linker and Libraries Guide

Notes The `Dl_info_t` pointer elements point to addresses within the mapped objects. These pointers can become invalid if objects are removed prior to these elements use. See `dlclose(3C)`.

If no symbol is found to describe the specified address, both the `dli_sname` and `dli_saddr` members are set to `0`.

If the address specified exists within a mapped object in the range between the base address and the address of the first global symbol in the object, the reserved local symbol `_START_` is returned. This symbol acts as a label representing the start of the mapped object. As a label, this symbol has no size. The `dli_saddr` member is set to the base address of the associated object. The `dli_sname` member is set to the symbol name `_START_`. If the flag argument is set to `RTLD_DL_SYMENT`, symbol information for `_START_` is returned.

If an object is acting as a filter, care should be taken when interpreting the address of any symbol obtained using a handle to this object. For example, using `dlsym(3C)` to obtain the symbol *_end* for this object, results in returning the address of the symbol *_end* within the filtee, not the filter. For more information on filters see the *Linker and Libraries Guide*.

Name dlclose – close a shared object

Synopsis

```
#include <dlfcn.h>

int dlclose(void *handle);
```

Description The `dlclose()` function decrements the reference count of the supplied *handle*. This *handle* represents an executable object file and its dependencies, acquired from a previous call to `dlopen()`. A *handle* that is no longer referenced is processed in an attempt to unload any objects that are associated with the *handle* from the current process. An unreferenced *handle* is no longer available to `dlsym()`.

Any finalization code within an object is executed prior to that object being unloaded. Any routines registered by an object using `atexit(3C)` are called prior to that object being unloaded. See NOTES.

Return Values If the *handle* was successfully unreferenced, `dlclose()` returns `0`. If the *handle* is invalid, or an error occurred as a result of unloading an object, `dlclose()` returns a non-zero value. Additional diagnostic information is available through `dlerror()`.

Usage The `dlclose()` function is one of a family of functions that give the user direct access to the dynamic linking facilities. These facilities are available to dynamically-linked processes only. See the *Linker and Libraries Guide*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `ld(1)`, `ld.so.1(1)`, `atexit(3C)`, `dladdr(3C)`, `dldump(3C)`, `dlerror(3C)`, `dlopen(3C)`, `dlsym(3C)`, `attributes(5)`, `standards(5)`

Linker and Libraries Guide

Notes A successful invocation of `dlclose()` does not guarantee that the objects associated with the *handle* are removed from the address space of the current process. Objects can be referenced by multiple *handles*, or by other objects. An object is not removed from the address space of the current process until all references to that object are removed.

Once an object has been closed by `dlclose()`, referencing symbols contained in that object can cause undefined behavior.

As part of unloading an object, finalization code within the object is called *before* the `dlclose()` returns. This finalization is user code, and as such, can produce errors that can not be caught by `dlclose()`. For example, an object loaded using `RTLD_LAZY` that attempts to call a

function that can not be located, results in process termination. Erroneous programming practices within the finalization code can also result in process termination. The runtime linkers debugging facility can offer help identifying these types of error. See the `LD_DEBUG` environment variable of `ld.so.1`(1).

Name dldump – create a new file from a dynamic object component of the calling process

Synopsis

```
#include <dlfcn.h>

int dldump(const char * ipath, const char * opath, int flags);
```

Description The `dldump()` function creates a new dynamic object *opath* from an existing dynamic object *ipath* that is bound to the current process. An *ipath* value of `0` is interpreted as the dynamic object that started the process. The new object is constructed from the existing objects' disc file. Relocations can be applied to the new object to pre-bind it to other dynamic objects, or fix the object to a specific memory location. In addition, data elements within the new object can be obtained from the objects' memory image as this data exists in the calling process.

These techniques allow the new object to be executed with a lower startup cost. This reduction can be because of less relocations being required to load the object, or because of a reduction in the data processing requirements of the object. However, limitations can exist in using these techniques. The application of relocations to the new dynamic object *opath* can restrict its flexibility within a dynamically changing environment. In addition, limitations in regards to data usage can make dumping a memory image impractical. See `EXAMPLES`.

The runtime linker verifies that the dynamic object *ipath* is mapped as part of the current process. Thus, the object must either be the dynamic object that started the process, one of the process's dependencies, or an object that has been preloaded. See `exec(2)`, and `ld.so.1(1)`.

As part of the runtime processing of a dynamic object, *relocation* records within the object are interpreted and applied to offsets within the object. These offsets are said to be *relocated*. Relocations can be categorized into two basic types: *non-symbolic* and *symbolic*.

The *non-symbolic* relocation is a simple *relative* relocation that requires the base address at which the object is mapped to perform the relocation. The *symbolic* relocation requires the address of an associated symbol, and results in a *binding* to the dynamic object that defines this symbol. The symbol definition can originate from any of the dynamic objects that make up the process, that is, the object that started the process, one of the process's dependencies, an object that has been preloaded, or the dynamic object being relocated.

The *flags* parameter controls the relocation processing and other attributes of producing the new dynamic object *opath*. Without any *flags*, the new object is constructed solely from the contents of the *ipath* disc file without any relocations applied.

Various relocation flags can be `or`'ed into the *flags* parameter to affect the relocations that are applied to the new object. *Non-symbolic* relocations can be applied using the following:

`RTLD_REL_RELATIVE` Relocation records from the object *ipath*, that define *relative* relocations, are applied to the object *opath*.

A variety of *symbolic* relocations can be applied using the following flags (each of these flags also implies `RTLD_REL_RELATIVE` is in effect):

RTLD_REL_EXEC	Symbolic relocations that result in binding *ipath* to the dynamic object that started the process, commonly a dynamic executable, are applied to the object *opath*.
RTLD_REL_DEPENDS	Symbolic relocations that result in binding *ipath* to any of the dynamic dependencies of the process are applied to the object *opath*.
RTLD_REL_PRELOAD	Symbolic relocations that result in binding *ipath* to any objects preloaded with the process are applied to the object *opath*. See LD_PRELOAD in ld.so.1(1).
RTLD_REL_SELF	Symbolic relocations that result in binding *ipath* to itself, are applied to the object *opath*.
RTLD_REL_WEAK	Weak relocations that remain unresolved are applied to the object *opath* as 0.
RTLD_REL_ALL	*All* relocation records defined in the object *ipath* are applied to the new object *opath*. This is basically a concatenation of all the above relocation flags.

Note that for dynamic executables, RTLD_REL_RELATIVE, RTLD_REL_EXEC, and RTLD_REL_SELF have no effect. See EXAMPLES.

If relocations, knowledgeable of the base address of the mapped object, are applied to the new object *opath*, then the new object becomes fixed to the location that the *ipath* image is mapped within the current process.

Any relocations applied to the new object *opath* will have the original relocation record removed so that the relocation will not be applied more than once. Otherwise, the new object *opath* will retain the relocation records as they exist in the *ipath* disc file.

The following additional attributes for creating the new dynamic object *opath* can be specified using the *flags* parameter:

RTLD_MEMORY	The new object *opath* is constructed from the current memory contents of the *ipath* image as it exists in the calling process. This option allows data modified by the calling process to be captured in the new object. Note that not all data modifications may be applicable for capture; significant restrictions exist in using this technique. See EXAMPLES. By default, when processing a dynamic executable, any allocated memory that follows the end of the data segment is captured in the new object (see malloc(3C) and brk(2)). This data, which represents the process heap, is saved as a new *.SUNW_heap* section in the object *opath*. The objects' program headers and symbol entries, such as _end, are adjusted accordingly. See also RTLD_NOHEAP. When using this attribute, any relocations that have been applied to the *ipath* memory image that do not fall into one of the requested

relocation categories are undone, that is, the relocated element is returned to the value as it existed in the *ipath* disc file.

`RTLD_STRIP` Only collect allocatable sections within the object *opath*. Sections that are not part of the dynamic objects' memory image are removed. `RTLD_STRIP` reduces the size of the *opath* disc file and is comparable to having run the new object through `strip(1)`.

`RTLD_NOHEAP` Do not save any heap to the new object. This option is only meaningful when processing a dynamic executable with the `RTLD_MEMORY` attribute and allows for reducing the size of the *opath* disc file. The executable must confine its data initialization to data elements within its data segment, and must not use any allocated data elements that comprise the heap.

It should be emphasized, that an object created by `dldump()` is simply an updated ELF object file. No additional state regarding the process at the time `dldump()` is called is maintained in the new object. `dldump()` does not provide a panacea for checkpoint and resume. A new dynamic executable, for example, will not start where the original executable called `dldump()`. It will gain control at the executable's normal entry point. See `EXAMPLES`.

Return Values

On successful creation of the new object, `dldump()` returns `0`. Otherwise, a non-zero value is returned and more detailed diagnostic information is available through `dlerror()`.

Examples

EXAMPLE 1 Sample code using `dldump()`.

The following code fragment, which can be part of a dynamic executable `a.out`, can be used to create a new shared object from one of the dynamic executables' dependencies `libfoo.so.1`:

```
const char *    ipath = "libfoo.so.1";
const char *    opath = "./tmp/libfoo.so.1";
...
if (dldump(ipath, opath, RTLD_REL_RELATIVE) != 0)
        (void) printf("dldump failed: %s\n", dlerror( ));
```

The new shared object *opath* is fixed to the address of the mapped *ipath* bound to the dynamic executable `a.out`. All relative relocations are applied to this new shared object, which will reduce its relocation overhead when it is used as part of another process.

By performing only relative relocations, any symbolic relocation records remain defined within the new object, and thus the dynamic binding to external symbols will be preserved when the new object is used.

Use of the other relocation flags can fix specific relocations in the new object and thus can reduce even more the runtime relocation startup cost of the new object. However, this will also restrict the flexibility of using the new object within a dynamically changing environment, as it will bind the new object to some or all of the dynamic objects presently mapped as part of the process.

EXAMPLE 1 Sample code using `dldump()`. *(Continued)*

For example, the use of `RTLD_REL_SELF` will cause any references to symbols from *ipath* to be bound to definitions within itself if no other preceding object defined the same symbol. In other words, a call to *foo( )* within *ipath* will bind to the definition *foo* within the same object. Therefore, *opath* will have one less binding that must be computed at runtime. This reduces the startup cost of using *opath* by other applications; however, interposition of the symbol *foo* will no longer be possible.

Using a dumped shared object with applied relocations as an applications dependency normally requires that the application have the same dependencies as the application that produced the dumped image. Dumping shared objects, and the various flags associated with relocation processing, have some specialized uses. However, the technique is intended as a building block for future technology.

The following code fragment, which is part of the dynamic executable `a.out`, can be used to create a new version of the dynamic executable:

```
static char *       dumped = 0;
const char *        opath = "./a.out.new";
...
if (dumped == 0) {
        char        buffer[100];
        int         size;
        time_t      seconds;
        ...
        /* Perform data initialization */
        seconds = time((time_t *)0);
        size = cftime(buffer, (char *)0, &seconds);
        if ((dumped = (char *)malloc(size + 1)) == 0) {
                (void) printf("malloc failed: %s\n", strerror(errno));
                return (1);
        }
        (void) strcpy(dumped, buffer);
        ...
        /*
         * Tear down any undesirable data initializations and
         * dump the dynamic executables memory image.
         */
        _exithandle( );
        _exit(dldump(0, opath, RTLD_MEMORY));
}
(void) printf("Dumped: %s\n", dumped);
```

Any modifications made to the dynamic executable, up to the point the `dldump()` call is made, are saved in the new object `a.out.new`. This mechanism allows the executable to update parts of its data segment and heap prior to creating the new object. In this case, the date the

EXAMPLE 1 Sample code using `dldump()`. *(Continued)*

executable is dumped is saved in the new object. The new object can then be executed without having to carry out the same (presumably expensive) initialization.

For greatest flexibility, this example does not save *any* relocated information. The elements of the dynamic executable *ipath* that have been modified by relocations at process startup, that is, references to external functions, are returned to the values of these elements as they existed in the *ipath* disc file. This preservation of relocation records allows the new dynamic executable to be flexible, and correctly bind and initialize to its dependencies when executed on the same or newer upgrades of the OS.

Fixing relocations by applying some of the relocation flags would bind the new object to the dependencies presently mapped as part of the process calling `dldump()`. It may also remove necessary copy relocation processing required for the correct initialization of its shared object dependencies. Therefore, if the new dynamic executables' dependencies have no specialized initialization requirements, the executable may still only interact correctly with the dependencies to which it binds if they were mapped to the same locations as they were when `dldump()` was called.

Note that for dynamic executables, `RTLD_REL_RELATIVE`, `RTLD_REL_EXEC`, and `RTLD_REL_SELF` have no effect, as relocations within the dynamic executable will have been fixed when it was created by `ld(1)`.

When `RTLD_MEMORY` is used, care should be taken to insure that dumped data sections that reference external objects are not reused without appropriate re-initialization. For example, if a data item contains a file descriptor, a variable returned from a shared object, or some other external data, and this data item has been initialized prior to the `dldump()` call, its value will have no meaning in the new dumped image.

When `RTLD_MEMORY` is used, any modification to a data item that is initialized via a relocation whose relocation record will be retained in the new image will effectively be lost or invalidated within the new image. For example, if a pointer to an external object is incremented prior to the `dldump()` call, this data item will be reset to its disc file contents so that it can be relocated when the new image is used; hence, the previous increment is lost.

Non-idempotent data initializations may prevent the use of `RTLD_MEMORY`. For example, the addition of elements to a linked-list via `init` sections can result in the linked-list data being captured in the new image. Running this new image may result in `init` sections continuing to add new elements to the list without the prerequisite initialization of the list head. It is recommended that `_exithandle(3C)` be called before `dldump()` to tear down any data initializations established via initialization code. Note that this may invalidate the calling image; thus, following the call to `dldump()`, only a call to `_Exit(2)` should be made.

Usage The `dldump()` function is one of a family of functions that give the user direct access to the dynamic linking facilities. These facilities are available to dynamically-linked processes only. See *Linker and Libraries Guide*).

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcs
MT-Level	MT-Safe

See Also `ld(1)`, `ld.so.1(1)`, `strip(1)`, `_Exit(2)`, `brk(2)`, `exec(2)`, `_exithandle(3C)`, `dladdr(3C)`, `dlclose(3C)`, `dlerror(3C)`, `dlopen(3C)`, `dlsym(3C)`, `end(3C)`, `malloc(3C)`, `attributes(5)`

Linker and Libraries Guide

Notes These functions are available to dynamically-linked processes only.

Any `NOBITS` sections within the *ipath* are expanded to `PROGBITS` sections within the *opath*. `NOBITS` sections occupy no space within an ELF file image. `NOBITS` sections declare memory that must be created and zero-filled when the object is mapped into the runtime environment. `.bss` is a typical example of this section type. `PROGBITS` sections, on the other hand, hold information defined by the object within the ELF file image. This section conversion reduces the runtime initialization cost of the new dumped object but increases the objects' disc space requirement.

When a shared object is dumped, and relocations are applied which are knowledgeable of the base address of the mapped object, the new object is fixed to this new base address. The dumped object has its ELF type reclassified to be a dynamic executable. The dumped object can be processed by the runtime linker, but is not valid as input to the link-editor.

If relocations are applied to the new object, any remaining relocation records are reorganized for better locality of reference. The relocation sections are renamed to `.SUNW_reloc` and the association with the section to relocate, is lost. Only the offset of the relocation record is meaningful. `.SUNW_reloc` relocations do not make the new object invalid to either the runtime linker or link-editor, but can reduce the objects analysis with some ELF readers.

Name dlerror – get diagnostic information

Synopsis

```
#include <dlfcn.h>

char *dlerror(void);
```

Description The `dlerror()` function returns a null-terminated character string that describes the last error that occurred during dynamic linking processing. The returned string contains no trailing newline. If no dynamic linking errors have occurred since the last invocation of `dlerror()`, `dlerror()` returns `NULL`. Thus, invoking `dlerror()` a second time, immediately following a prior invocation, results in `NULL` being returned.

Usage The `dlerror()` function is one of a family of functions that give the user direct access to the dynamic linking facilities. These facilities are available to dynamically-linked processes only. See *Linker and Libraries Guide*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `ld(1)`, `dladdr(3C)`, `dlclose(3C)`, `dldump(3C)`, `dlopen(3C)`, `dlsym(3C)`, `attributes(5)`, `standards(5)`

Linker and Libraries Guide

Notes The messages returned by `dlerror()` can reside in a static buffer that is overwritten on each call to `dlerror()`. Application code should not write to this buffer. Programs wanting to preserve an error message should make their own copies of that message.

Name dlinfo – dynamic load information

Synopsis

```
#include <dlfcn.h>
#include <link.h>
#include <limits.h>
#include <sys/mman.h>

int dlinfo(void *handle, int request, void *p);
```

Description The `dlinfo()` function sets or extracts information from the runtime linker `ld.so.1(1)`. This function is loosely modeled after the `ioctl(2)` function. The *request* argument and a third argument of varying type are passed to `dlinfo()`. The action taken by `dlinfo()` depends on the value of the *request* that is provided.

The *handle* argument is either the value that is returned from a `dlopen(3C)` or `dlmopen()` call, or the special handle `RTLD_SELF`. A *handle* argument is required for all requests except `RTLD_DI_CONFIGADDR`, `RTLD_DI_GETSIGNAL`, and `RTLD_DI_SETSIGNAL`. If *handle* is the value that is returned from a `dlopen()` or `dlmopen()` call, the information returned by the `dlinfo()` call pertains to the specified object. If *handle* is the special handle `RTLD_SELF`, the information returned by the `dlinfo()` call pertains to the caller.

The *request* argument can take the following values:

`RTLD_DI_ARGSINFO`
Obtain process argument information. The *p* argument is a pointer (`Dl_argsinfo_t *`*p*). The following elements from this structure are initialized:

`dla_argc` The number of arguments passed to the process.

`dla_argv` The argument array passed to the process.

`dla_envp` The active environment variable array that is available to the process. This element initially points to the environment variable array that is made available to `exec(2)`. This element can be updated should an alternative environment be established by the process. See `putenv(3C)`.

`dla_auxv` The auxiliary vector array passed to the process.

A process can be established from executing the runtime linker directly from the command line. See `ld.so.1(1)`. The `Dl_argsinfo_t` information reflects the information that is made available to the application regardless of how the runtime linker has been invoked.

`RTLD_DI_CONFIGADDR`
Obtain the configuration file information. The *p* argument is a `Dl_info_t` pointer (`Dl_info_t *`*p*). The following elements from this structure are initialized:

`dli_fname` The full name of the configuration file.

`dli_fbase` The base address of the configuration file loaded into memory.

RTLD_DI_LINKMAP
Obtain the Link_map for the *handle* that is specified. The *p* argument points to a Link_map pointer (Link_map **p*). The actual storage for the Link_map structure is maintained by ld.so.1.

The Link_map structure includes the following members:

```
unsigned long l_addr;      /* base address */
char          *l_name;     /* object name */
Elf32_Dyn     *l_ld;       /* .dynamic section */
Link_map      *l_next;     /* next link object */
Link_map      *l_prev;     /* previous link object */
char          *l_refname;  /* filter reference name */
```

l_addr — The base address of the object loaded into memory.

l_name — The full name of the loaded object. This full name is the filename of the object as referenced by ld.so.1.

l_ld — Points to the SHT_DYNAMIC structure.

l_next — The next Link_map on the link-map list. Other objects on the same link-map list as the current object can be examined by following the l_next and l_prev members.

l_prev — The previous Link_map on the link-map list.

l_refname — If the object that is referenced is a *filter*, this member points to the name of the object being filtered. If the object is not a *filter*, this member is 0. See the *Linker and Libraries Guide*.

RTLD_DI_LMID
Obtain the ID for the link-map list upon which the *handle* is loaded. The *p* argument is a Lmid_t pointer (Lmid_t **p*).

RTLD_DI_MMAPCNT
Determine the number of segment mappings for the *handle* that is specified, for use in a RTLD_DI_MMAPS request. The *p* argument is a uint_t pointer (uint_t **p*). On return from a RTLD_DI_MMAPCNT request, the uint_t value that is pointed to by *p* contains the number of segment mappings that the associated object uses.

To obtain the complete mapping information for an object, a mmapobj_result_t array for RTLD_DI_MMAPCNT entries must be provided. This array is assigned to the dlm_maps member, and the number of entries available in the array are assigned to the dlm_acnt member. This initialized structure is then passed to a RTLD_DI_MMAPS request. See EXAMPLES.

RTLD_DI_MMAPS
Obtain segment mapping information for the *handle* that is specified. The *p* argument is a Dl_mapinfo_t pointer (Dl_mapinfo_t *p). This structure can be initialized from the mapping count obtained from a previous RTLD_DI_MMAPCNT request.

Segment mapping information is provided in an array of mmapobj_result_t structures that originate from the mmapobj(2) of the associated object. The dlm_acnt member, typically initialized from a previous RTLD_DI_MMAPCNT request, indicates the number of entries in a mmapobj_result_t array. This array is assigned to the dlm_maps member. This initialized structure is then passed to a RTLD_DI_MMAPS request, where the segment mapping information is copied to the mmapobj_result_t array. The dlm_rcnt member indicates the number of mmapobj_result_t element entries that are returned. See EXAMPLES.

RTLD_DI_SERINFO
Obtain the library search paths for the *handle* that is specified. The *p* argument is a Dl_serinfo_t pointer (Dl_serinfo_t *p). A user must first initialize the Dl_serinfo_t structure with a RTLD_DI_SERINFOSIZE request. See EXAMPLES.

The returned Dl_serinfo_t structure contains dls_cnt Dl_serpath_t entries. Each entry's dlp_name member points to the search path. The corresponding dlp_info member contains one of more flags indicating the origin of the path. See the LA_SER_* flags that are defined in <link.h>.

RTLD_DI_SERINFOSIZE
Initialize a Dl_serinfo_t structure for the *handle* that is specified, for use in a RTLD_DI_SERINFO request. Both the dls_cnt and dls_size members are returned. The dls_cnt member indicates the number of search paths that are applicable to the *handle*. The dls_size member indicates the total size of a Dl_serinfo_t buffer required to hold dls_cnt Dl_serpath_t entries and the associated search path strings. The *p* argument is a Dl_serinfo_t pointer (Dl_serinfo_t *p).

To obtain the complete path information, a new Dl_serinfo_t buffer of size dls_size should be allocated. This new buffer should be initialized with the dls_cnt and dls_size entries. The initialized buffer is then passed to a RTLD_DI_SERINFO request. See EXAMPLES.

RTLD_DI_ORIGIN
Obtain the origin of the dynamic object that is associated with the *handle*. The *p* argument is a char pointer (char *p). The dirname(3C) of the associated object's realpath(3C), which can be no larger than {PATH_MAX}, is copied to the pointer *p*.

RTLD_DI_GETSIGNAL
Obtain the numeric signal number used by the runtime linker to kill the process in the event of a fatal runtime error. The *p* argument is an int pointer (int *p). The signal number is copied to the pointer *p*.

By default, the signal used by the runtime linker to terminate a process is SIGKILL. See thr_kill(3C). This default can be changed by calling dlinfo() with RTLD_DI_SETSIGNAL or by setting the environment variable LD_SIGNAL. See ld.so.1(1).

RTLD_DI_SETSIGNAL

Provide a numeric signal number used by the runtime linker to kill the process in the event of a fatal runtime error. The *p* argument is an int pointer (int *p). The value pointed to by *p* is established as the terminating signal value.

The current signal number used by the runtime linker to terminate a process can be obtained from dlinfo() using RTLD_DI_GETSIGNAL. Use of the RTLD_DI_SETSIGNAL option is equivalent to setting the environment variable LD_SIGNAL. See ld.so.1(1).

RTLD_DI_DEFERRED

Assign a new dependency name to an existing deferred dependency. The *p* argument is a Dl_definfo_t pointer (Dl_definfo *p). The dlv_refname field defines an existing dependency name. The dlv_depname field defines the new dependency name.

Dependency names are defined by DT_NEEDED dynamic entries, which can be displayed using the -d option of elfdump(1). Individual dependencies can be tagged as deferred. See the -z deferred option of ld(1). Deferred dependencies are only loaded during process execution, when the first binding to a deferred reference is made. Prior to a deferred dependency being loaded, the dependency name can be changed with RTLD_DI_DEFERRED. See also RTLD_DI_DEFERRED_SYM.

Once a deferred dependency is loaded, any attempt to change the dependency name with dlinfo() results in an error return of −1.

RTLD_DI_DEFERRED_SYM

Assign a new dependency name to an existing deferred symbol, using a symbol reference that exists to the dependency. The *p* argument is a Dl_definfo_t pointer (Dl_definfo *p). The dlv_refname field defines a symbol reference to the deferred dependency. The dlv_depname field defines the new dependency name.

RTLD_DI_DEFERRED_SYM provides an alternative means of modifying a deferred dependency to using RTLD_DI_DEFERRED. One, or more symbol references can be associated with a deferred dependency. RTLD_DI_DEFERRED_SYM allows one of these deferred symbol references to be used to select the associated deferred dependency. Prior to a deferred dependency being loaded, the dependency name can be changed with RTLD_DI_DEFERRED_SYM. See EXAMPLES.

Once a deferred dependency is loaded, any attempt to change the dependency name with dlinfo() results in an error return of −1.

Return Values The dlinfo() function returns −1 if the *request* is invalid, the parameter *p* is NULL, or the Dl_serinfo_t structure is uninitialized for a RTLD_DI_SERINFO request. dlinfo() also

returns –1 if the *handle* argument does not refer to a valid object opened by `dlopen()`, or is not the special handle `RTLD_SELF`. Detailed diagnostic information is available with `dlerror(3C)`.

Examples

EXAMPLE 1 Use `dlinfo()` to obtain library search paths.

The following example demonstrates how a dynamic object can inspect the library search paths that would be used to locate a simple filename with `dlopen()`. For simplicity, error checking has been omitted.

```
Dl_serinfo_t   _info, *info = &_info;
Dl_serpath_t   *path;
uint_t         cnt;

/* determine search path count and required buffer size */
dlinfo(RTLD_SELF, RTLD_DI_SERINFOSIZE, info);

/* allocate new buffer and initialize */
info = malloc(_info.dls_size);
info->dls_size = _info.dls_size;
info->dls_cnt = _info.dls_cnt;

/* obtain sarch path information */
dlinfo(RTLD_SELF, RTLD_DI_SERINFO, info);

path = &info->dls_serpath[0];

for (cnt = 1; cnt <= info->dls_cnt; cnt++, path++) {
    (void) printf("%2d: %s\n", cnt, path->dls_name);
}
```

EXAMPLE 2 Use `dlinfo()` to obtain segment information.

The following example demonstrates how a dynamic object can inspect its segment mapping information. For simplicity, error checking has been omitted

```
Dl_mapinfo_t   mi;
uint_t         cnt;

/* determine the number of segment mappings */
dlinfo(RTLD_SELF, RTLD_DI_MMAPCNT, &mi.dlm_acnt);

/* allocate the appropriate mapping array */
mi.dlm_maps = malloc(mi.dlm_acnt * sizeof (mmapobj_result_t));

/* obtain the mapping information */
dlinfo(RTLD_SELF, RTLD_DI_MMAPS, &mi);

for (cnt = 0; cnt < mi.dlm_rcnt; cnt++) {
```

EXAMPLE 2 Use `dlinfo()` to obtain segment information. *(Continued)*

```
        (void) printf("addr=%x - memory size=%x\n",
            mi.dlm_maps[cnt].mr_addr, mi.dlm_maps[cnt].mr_msize);
    }
```

EXAMPLE 3 Use `dlinfo()` to change a deferred dependency.

The following program defines a deferred dependency, `bar.so`, and an associated deferred symbol reference.

```
$ elfdump -d main | egrep "NEEDED|POSFLAG
    [0]  POSFLAG_1          0x5                 [ LAZY DEFERRED ]
    [1]  NEEDED             0x17e               bar.so
$ elfdump -y main | fgrep foo
    [12]  DBLP              [1] bar.so          foo
```

The program probes the existence of the symbol `foo()`, and if the symbol can not be found, exchanges the deferred dependency associated with the symbol for a new dependency name `foo.so`.

```
    if (dlsym(RTLD_PROBE, "foo") == NULL) {
        Dl_definfo_t    info;

        info.dld_refname = "foo";
        info.dld_depname = "foo.so";

        if (dlinfo(RTLD_SELF, RTLD_DI_DEFERRED_SYM, &info) == -1)
            return (1);
        if (dlsym(RTLD_PROBE, "foo") == NULL)
            return (1);
    }
    return (foo());
```

Usage

The `dlinfo()` function is one of a family of functions that give the user direct access to the dynamic linking facilities. These facilities are available to dynamically-linked processes only. See the *Linker and Libraries Guide*.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also elfdump(1), ld(1), ld.so.1(1), exec(2), ioctl(2), mmapobj(2), dirname(3C), dlclose(3C), dldump(3C), dlerror(3C), dlopen(3C), dlsym(3C), putenv(3C), realpath(3C), thr_kill(3C), attributes(5).

Linker and Libraries Guide

Name dl_iterate_phdr – walk through a list of objects

Synopsis

```
#include <link.h>

int dl_iterate_phdr(int (*callback)(struct dl_phdr_info *info,
     size_t size, void *data), void *data);
```

Description The `dl_iterate_phdr()` function returns information regarding each ELF object currently resident in the process address space.

The `dl_iterate_phdr()` function calls the function *callback* once for each object, until either all objects have been processed or *callback* returns a non-zero value.

Each call to *callback* receives three arguments: *info*, which is a pointer to a structure containing information about the object; *size*, which is the size of the structure pointed to by *info*; and the *data* argument passed to `dl_iterate_phdr()` by the caller.

The *info* argument is a pointer to a structure of the following type:

```
struct dl_phdr_info {
        /* Fields present in all implementations */
        ElfW(Addr)       dlpi_addr;
        const char       *dlpi_name;
        const ElfW(Phdr) *dlpi_phdr;
        ElfW(Half)       dlpi_phnum;

        /* Additional fields present in this implementation */
        u_longlong_t     dlpi_adds;
        u_longlong_t     dlpi_subs;
};
```

The `ElfW()` macro definition turns its argument into the name of an ELF data type suitable for the hardware architecture, by adding the `Elf32_` prefix for 32-bit code, or `Elf64_` for 64-bit code.

The first four fields (`dlpi_addr`, `dlpi_name`, `dlpi_phdr`, `dlpi_phnum`) are present in all implementations of `dl_iterate_phdr()`, and can be accessed on any system that provides this function. The callback function must use the *size* argument to determine if the remaining fields (`dlpi_adds`, `dlpi_subs`) are present. See EXAMPLES.

The `dlpi_addr` field is 0 for executable objects (`ET_EXEC`), and is the base address at which the object is mapped otherwise. Therefore, the address of any loadable segment in the program header array can be calculated as:

```
addr == info->dlpi_addr + info->dlpi_phdr[x].p_vaddr
```

`dlpi_name` gives the pathname of the object.

`dlpi_phdr` provides a pointer to the program header array for the object, and `dlpi_phnum` specifies the number of program headers found in the array.

`dlpi_adds` provides the number of objects that have been mapped into the current process since it started, and `dlpi_subs` provides the number of objects that have been unmapped. See NOTES.

See the *Linker and Libraries Guide* for more information about ELF objects, and the information contained in program headers.

Examples

EXAMPLE 1 Display all currently mapped object

The following program displays the pathnames of currently mapped objects. For each object, the virtual address of each loadable segment is shown.

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

static int
callback(struct dl_phdr_info *info, size_t size, void *data)
{
        int j;

        printf("name=%s (%d program headers)\n", info->dlpi_name,
            info->dlpi_phnum);
        for (j = 0; j < info->dlpi_phnum; j++) {
                if (info->dlpi_phdr[j].p_type == PT_LOAD)
                        printf("\t[%d] 0x%p\n", j,
                            (void *) (info->dlpi_addr +
                            info->dlpi_phdr[j].p_vaddr));
        }
        return 0;
}

int
main(int argc, char *argv[])
{
        dl_iterate_phdr(callback, NULL);
        return(0);
}
```

EXAMPLE 2 Testing for optional `dl_phdr_info` fields

Every implementation of dl_iterate_phdr is required to supply the first four fields in struct dl_phdr_info described above. The callback is allowed to assume that they are present and to access them without first testing for their presence. Additional fields may be present. The callback must use the size argument to test for their presence before accessing them. This example demonstrates how a callback function can detect the presence of the dlpi_adds and dlpi_subs fields described above:

```
static int
callback(struct dl_phdr_info *info, size_t size, void *data)
```

EXAMPLE 2 Testing for optional `dl_phdr_info` fields *(Continued)*

```
{
        /*
         * This must match the definition of dl_phdr_info, as
         * defined in <link.h>. It is used to determine whether
         * the info structure contains optional fields.
         */
        struct dl_phdr_info_test {
                ElfW(Addr)              dlpi_addr;
                const char              *dlpi_name;
                const ElfW(Phdr)        *dlpi_phdr;
                ElfW(Half)              dlpi_phnum;
                u_longlong_t            dlpi_adds;
                u_longlong_t            dlpi_subs;
        };

        printf("object:  %s\n", info->dlpi_name);
        printf("    addr:  0x%p\n", (u_longlong_t) info->dlpi_addr);
        printf("    phdr:  0x%p\n", (u_longlong_t) info->dlpi_phdr);
        printf("    phnum: %d\n", (int) info->dlpi_phnum);
        if (size >= sizeof (struct dl_phdr_info_test)) {
                printf("    adds:  %llu\n", info->dlpi_adds);
                printf("    subs:  %llu\n", info->dlpi_subs);
        }
        return (0);
}
```

Return Values The `dl_iterate_phdr()` function returns whatever value was returned by the last call to callback.

Usage The `dl_iterate_phdr()` function is a member of a family of functions that give the user direct access to the dynamic linking facilities. This family of functions is available only to dynamically-linked processes. See the *Linker and Libraries Guide*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `ld(1)`, `ld.so.1(1)`, `dladdr(3C)`, `dlclose(3C)`, `dldump(3C)`, `dlerror(3C)`, `dlinfo(3C)`, `dlopen(3C)`, `dlsym(3C)`, `attributes(5)`, `standards(5)`

Linker and Libraries Guide

Notes `dl_iterate_phdr()` was originally defined by the Linux operating system, and is contained in the Linux Standard Base (LSB).

The behavior of `dl_iterate_phdr()` when a callback function causes a new object to be loaded, either via lazy loading or a call to `dlopen()`, is undefined. The call to `dl_iterate_phdr()` that triggers the load may or may not issue a callback for the new object. This depends on the current position of `dl_iterate_phdr()` in the list of known objects when the new object is added. The caller must make no assumptions about this case.

`dl_iterate_phdr()` callbacks must not unload objects. If a call to `dlclose()` is detected from within the callback function, `dl_iterate_phdr()` immediately terminates the iteration operation and returns a value of -1.

If two separate calls to `dl_iterate_phdr()` provide the same two values for `dlpi_adds` and `dlpi_subs`, the caller may safely assume that the process object state has not changed between the two calls. An application can use this information to cache object data, and avoid unnecessary iteration. In such a scenario, the first call to the callback function would check to see if a cache exists, and that `dlpi_adds` and `dlpi_subs` have not changed since the last call to `dl_iterate_phdr()`, and if so, return a non-zero value to terminate the iteration operation immediately.

Name dlopen, dlmopen – gain access to an executable object file

Synopsis

```
#include <dlfcn.h>
#include <link.h>

void * dlopen(const char *pathname, int mode);

void * dlmopen(Lmid_t lmid, const char *pathname, int mode);
```

Description The `dlopen()` function makes an executable object file available to a running process. `dlopen()` returns to the process a *handle* that the process can use on subsequent calls to `dlsym(3C)`, `dladdr(3C)`, `dlinfo(3C)`, and `dlclose(3C)`. The value of this *handle* should not be interpreted in any way by the process. The *pathname* argument is the path name of the object to be opened. A path name containing an embedded `'/'` is interpreted as an absolute path or relative to the current directory. Otherwise, the set of search paths currently in effect by the runtime linker are used to locate the specified file. See `NOTES`.

If the object file referenced by `dlopen()` is not already loaded as part of the process, then the object file is added to the process address space. A handle for this object is created and returned to the caller. If the object file is already part of the process, a handle is also returned to the caller. Multiple references to the same object result in returning the same handle. A reference count within the handle maintains the number of callers. The `dlclose()` of a handle results in decrementing the handles reference count. When the reference count reaches 0 the object file is a candidate for unloading. Any `init` section within an object is called once when the object is loaded. Any `fini` section within an object is called once when the object is unloaded.

When `dlopen()` causes an object to be loaded, it also loads any non-lazy dependencies that are recorded within the object given by *pathname*. These dependencies are searched in the order in which the dependencies were loaded to locate any additional dependencies. This process continues until all the dependencies of *pathname* are loaded. This dependency tree is referred to as a group.

If the value of *pathname* is `0`, `dlopen()` provides a *handle* on a set of global symbol objects. These objects consist of the original program image file, any dependencies loaded at program startup, and any objects loaded using `dlopen()` with the `RTLD_GLOBAL` flag. Because the latter set of objects can change during process execution, the set identified by *handle* can also change dynamically.

The *mode* argument describes how `dlopen()` operates on *pathname* with respect to the processing of reference relocations. The *mode* also affects the scope of visibility of the symbols provided by *pathname* and its dependencies. This visibility can affect how the resulting *handle* is used.

When an object is loaded, the object can contain references to symbols whose addresses are not known until the object is loaded. These references must be relocated before the symbols can be accessed. References are categorized as either *immediate* or *lazy*. Immediate references

are typically references to data items used by the object code. Immediate references include pointers to functions and calls to functions made from position-dependent shared objects. Lazy references are typically calls to global functions that are made from position-independent shared objects.

Lazy references can also be identified as *deferred*. See the `-z deferred` option of `ld(1)`. Deferred dependencies are only loaded during process execution, when the first binding to a deferred reference is made. These references are unaffected by the *mode*.

The *mode* argument governs when non-deferred references take place. The *mode* argument can be one of the following values.

`RTLD_LAZY` Only immediate symbol references are relocated when the object is first loaded. Lazy references are not relocated until a given function is called for the first time. This value for *mode* should improve performance, since a process might not require all lazy references in any given object. This behavior mimics the normal loading of dependencies during process initialization. See `NOTES`.

`RTLD_NOW` All non-deferred relocations are performed when the object is first loaded. This process might waste some processing if relocations are performed for lazy references that are never used. However, this mode ensures that when an object is loaded, all non-deferred symbols that are referenced during execution are available. This behavior mimics the loading of dependencies when the environment variable `LD_BIND_NOW` is in effect.

See the *Linker and Libraries Guide* for more information about symbol references.

The visibility of symbols that are available for relocation can be affected by *mode*. To specify the scope of visibility for symbols that are loaded with a `dlopen()` call, *mode* should be a bitwise-inclusive OR with one of the following values:

`RTLD_GLOBAL` The object's global symbols are made available for the relocation processing of any other object. In addition, symbol lookup using `dlopen(0,` *mode*`)` and an associated `dlsym()` allows objects that are loaded with `RTLD_GLOBAL` to be searched.

`RTLD_LOCAL` The object's globals symbols are only available for the relocation processing of other objects that include the same group.

The program image file and any objects loaded at program startup have the mode `RTLD_GLOBAL`. The mode `RTLD_LOCAL` is the default mode for any objects that are acquired with `dlopen()`. A local object can be a dependency of more then one group. Any object of mode `RTLD_LOCAL` that is referenced as a dependency of an object of mode `RTLD_GLOBAL` is promoted to `RTLD_GLOBAL`. In other words, the `RTLD_LOCAL` mode is ignored.

Any object loaded by `dlopen()` that requires relocations against global symbols can reference the symbols in any `RTLD_GLOBAL` object. Objects of this mode are at least the program image

file and any objects loaded at program startup. A loaded object can also reference symbols from itself, and from any dependencies the object references. However, the *mode* parameter can also be a bitwise–inclusive OR with one of the following values to affect the scope of symbol availability:

`RTLD_GROUP` Only symbols from the associated group are made available for relocation. A group is established from the defined object and all the dependencies of that object. A group must be completely self-contained. All dependency relationships between the members of the group must be sufficient to satisfy the relocation requirements of each object that defines the group.

`RTLD_PARENT` The symbols of the object initiating the `dlopen()` call are made available to the objects obtained by `dlopen()`. This option is useful when hierarchical `dlopen()` families are created. Although the parent object can supply symbols for the relocation of this object, the parent object is not available to `dlsym()` through the returned *handle*.

`RTLD_WORLD` Only symbols from `RTLD_GLOBAL` objects are made available for relocation.

The default modes for `dlopen()` are both `RTLD_WORLD` and `RTLD_GROUP`. If an object is requires additional modes, the *mode* parameter can be the bitwise-inclusive OR of the required modes together with the default modes.

The following modes provide additional capabilities outside of relocation processing:

`RTLD_NODELETE` The specified object is tagged to prevent its deletion from the address space as part of a `dlclose()`.

`RTLD_NOLOAD` The specified object is not loaded as part of the `dlopen()`. However, a valid *handle* is returned if the object already exists as part of the process address space. Additional modes can be specified as a bitwise–inclusive OR with the present mode of the object and its dependencies. The `RTLD_NOLOAD` mode provides a means of querying the presence or promoting the modes of an existing dependency.

The default use of a *handle* with `dlsym()` allows a symbol search to inspect all objects that are associated with the group of objects that are loaded from `dlopen()`. The *mode* parameter can also be a bitwise–inclusive OR with the following value to restrict this symbol search:

`RTLD_FIRST` Use of this *handle* with `dlsym()`, restricts the symbol search to the first object associated with the *handle*.

An object can be accessed from a process both with and without `RTLD_FIRST`. Although the object will only be loaded once, two different *handle*s are created to provide for the different `dlsym()` requirements.

The `dlmopen()` function is identical to `dlopen()`, except that an identifying link-map ID *(lmid)* is provided. This link-map ID informs the dynamic linking facilities upon which link-map list to load the object. See the *Linker and Libraries Guide* for details about link-maps.

The *lmid* passed to `dlmopen()` identifies the link-map list on which the object is loaded. This parameter can be any valid `Lmid_t` returned by `dlinfo()` or one of the following special values:

`LM_ID_BASE` Load the object on the applications link-map list.

`LM_ID_LDSO` Load the object on the dynamic linkers (`ld.so.1`) link-map list.

`LM_ID_NEWLM` Cause the object to create a new link-map list as part of loading. Objects that are opened on a new link-map list must express all of their dependencies.

Return Values The `dlopen()` function returns `NULL` if *pathname* cannot be found, cannot be opened for reading, or is not a shared object or a relocatable object. `dlopen()` also returns `NULL` if an error occurs during the process of loading *pathname* or relocating its symbolic references. See `NOTES`. Additional diagnostic information is available through `dlerror()`.

Usage The `dlopen()` and `dlmopen()` functions are members of a family of functions that give the user direct access to the dynamic linking facilities. This family of functions is available only to dynamically-linked processes. See the *Linker and Libraries Guide*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT–Level	MT–Safe
Standard	See `standards(5)`.

See Also `ld(1)`, `ld.so.1(1)`, `dladdr(3C)`, `dlclose(3C)`, `dldump(3C)`, `dlerror(3C)`, `dlinfo(3C)`, `dlsym(3C)`, `attributes(5)`, `standards(5)`

Linker and Libraries Guide

Notes If *pathname* has dependencies on other objects, these objects are automatically loaded by `dlopen()`. The directory search path used to find *pathname* and any dependencies can be affected by setting the environment variable `LD_LIBRARY_PATH`. Any `LD_LIBRARY_PATH` variable is analyzed once at process startup. The search path can also be affected from a runpath setting within the object from which the call to `dlopen()` originates. These search rules will only be applied to path names that do not contain an embedded `'/'`. Objects whose names resolve to the same absolute path name or relative path name can be opened any number of times using `dlopen()`. However, the object that is referenced will only be loaded once into the address space of the current process.

When loading shared objects, the application should open a specific version of the shared object. Do not rely on the version of the shared object pointed to by the symbolic link.

When building objects to be loaded on a new link-map list, some precautions need to be taken. In general, all dependencies must be included when building an object. Also, include `/usr/lib/libmapmalloc.so.1` before `/lib/libc.so.1` when building an object.

When an object is loaded on a new link-map list, the object is isolated from the main running program. Certain global resources are only usable from one link-map list. A few examples are the `sbrk()` based `malloc()`, `libthread()`, and the signal vectors. Care must be taken not to use any of these resources other than from the primary link-map list. These issues are discussed in further detail in the *Linker and Libraries Guide*.

Some symbols defined in dynamic executables or shared objects can not be available to the runtime linker. The symbol table created by `ld` for use by the runtime linker might contain only a subset of the symbols that are defined in the object.

As part of loading a new object, initialization code within the object is called *before* the `dlopen()` returns. This initialization is user code, and as such, can produce errors that can not be caught by `dlopen()`. For example, an object loaded using `RTLD_LAZY` that attempts to call a function that can not be located results in process termination. Erroneous programming practices within the initialization code can also result in process termination. The runtime linkers debugging facility can offer help identifying these types of error. See the `LD_DEBUG` environment variable of `ld.so.1(1)`.

Loading relocatable objects is an expensive operation that requires converting the relocatable object into a shared object memory image. This capability may be useful in a debugging environment, but is not recommended for production software.

Name dlsym – get the address of a symbol in a shared object or executable

Synopsis

```
#include <dlfcn.h>

void *dlsym(void *restrict handle, const char *restrict name);
```

Description The `dlsym()` function allows a process to obtain the address of a symbol that is defined within a shared object or executable. The *handle* argument is either the value returned from a call to `dlopen()` or one of a family of special handles. The *name* argument is the symbol's name as a character string.

If *handle* is returned from `dlopen()`, the associated shared object must not have been closed using `dlclose()`. A *handle* can be obtained from `dlopen()` using the `RTLD_FIRST` mode. With this mode, the `dlsym()` function searches for the named symbol in the initial object referenced by *handle*. Without this mode, the `dlsym()` function searches for the named symbol in the group of shared objects loaded automatically as a result of loading the object referenced by *handle*. See `dlopen(3C)` and NOTES.

The following special handles are supported.

`RTLD_DEFAULT` Instructs `dlsym()` to search for the named symbol starting with the first object loaded, typically the dynamic executable. The search continues through the list of initial dependencies that are loaded with the process, followed by any objects obtained with `dlopen(3C)`. This search follows the default model that is used to relocate all objects within the process.

This model also provides for transitioning into a lazy loading environment. If a symbol can not be found in the presently loaded objects, any pending lazy loaded objects are processed in an attempt to locate the symbol. This loading compensates for objects that have not fully defined their dependencies. However, this compensation can undermine the advantages of lazy loading.

`RTLD_PROBE` Instructs `dlsym()` to search for the named symbol in the same manner as occurs with a *handle* of `RTLD_DEFAULT`. However, `RTLD_PROBE` only searches for symbol definitions in the presently loaded objects, together with any lazy loadable objects specifically identified by the caller to provide the named symbol. This handle does not trigger an exhaustive load of any lazy loadable symbols in an attempt to find the named symbol. This handle can provide a more optimal search than would occur using `RTLD_DEFAULT`.

`RTLD_NEXT` Instructs `dlsym()` to search for the named symbol in the objects that were loaded following the object from which the `dlsym()` call is being made.

`RTLD_SELF` Instructs `dlsym()` to search for the named symbol in the objects that were loaded starting with the object from which the `dlsym()` call is being made.

When used with a special handle, `dlsym()` is selective in searching objects that have been loaded using `dlopen()`. These objects are searched for symbols if one of the following conditions are true.

- The object is part of the same local `dlopen()` dependency hierarchy as the calling object. See the *Linker and Libraries Guide* for a description of `dlopen()` dependency hierarchies.
- The object has global search access. See `dlopen(3C)` for a discussion of the `RTLD_GLOBAL` mode.

Return Values The `dlsym()` function returns `NULL` if *handle* does not refer to a valid object opened by `dlopen()` or is not one of the special handles. The function also returns `NULL` if the named symbol cannot be found within any of the objects associated with *handle*. Additional diagnostic information is available through `dlerror(3C)`.

Examples **EXAMPLE 1** Use `dlopen()` and `dlsym()` to access a function or data objects.

The following code fragment demonstrates how to use `dlopen()` and `dlsym()` to access either function or data objects. For simplicity, error checking has been omitted.

```
void      *handle;
int       *iptr, (*fptr)(int);

/* open the needed object */
handle = dlopen("/usr/home/me/libfoo.so.1", RTLD_LAZY);

/* find the address of function and data objects */
fptr = (int (*)(int))dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");

/* invoke function, passing value of integer as a parameter */
(*fptr)(*iptr);
```

EXAMPLE 2 Use `dlsym()` to verify that a particular function is defined.

The following code fragment shows how to use `dlsym()` to verify that a function is defined. If the function exists, the function is called.

```
int (*fptr)();

if ((fptr = (int (*)())dlsym(RTLD_DEFAULT,
    "my_function")) != NULL) {
        (*fptr)();
}
```

Usage The `dlsym()` function is one of a family of functions that give the user direct access to the dynamic linking facilities. These facilities are available to dynamically-linked processes only. See the *Linker and Libraries Guide*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `ld(1)`, `ld.so.1(1)`, `dladdr(3C)`, `dlclose(3C)`, `dldump(3C)`, `dlerror(3C)`, `dlinfo(3C)`, `dlopen(3C)`, `attributes(5)`, `standards(5)`

Linker and Libraries Guide

Notes If an object is acting as a filter, care should be taken when interpreting the address of any symbol obtained using a handle to this object. For example, using `dlsym(3C)` to obtain the symbol *_end* for this object, results in returning the address of the symbol *_end* within the filtee, not the filter. For more information on filters see the *Linker and Libraries Guide*.

Name door_bind, door_unbind – bind or unbind the current thread with the door server pool

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <door.h>

int door_bind(int did);

int door_unbind(void);
```

Description The `door_bind()` function associates the current thread with a door server pool. A door server pool is a private pool of server threads that is available to serve door invocations associated with the door *did*.

The `door_unbind()` function breaks the association of `door_bind()` by removing any private door pool binding that is associated with the current thread.

Normally, door server threads are placed in a global pool of available threads that invocations on any door can use to dispatch a door invocation. A door that has been created with `DOOR_PRIVATE` only uses server threads that have been associated with the door by `door_bind()`. It is therefore necessary to bind at least one server thread to doors created with `DOOR_PRIVATE`.

The server thread create function, `door_server_create()`, is initially called by the system during a `door_create()` operation. See `door_server_create(3C)` and `door_create(3C)`.

The current thread is added to the private pool of server threads associated with a door during the next `door_return()` (that has been issued by the current thread after an associated `door_bind()`). See `door_return(3C)`. A server thread performing a `door_bind()` on a door that is already bound to a different door performs an implicit `door_unbind()` of the previous door.

If a process containing threads that have been bound to a door calls `fork(2)`, the threads in the child process will be bound to an invalid door, and any calls to `door_return(3C)` will result in an error.

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

Errors The `door_bind()` and `door_unbind()` functions fail if:

`EBADF` The *did* argument is not a valid door.

`EBADF` The `door_unbind()` function was called by a thread that is currently not bound.

`EINVAL` *did* was not created with the `DOOR_PRIVATE` attribute.

Examples

EXAMPLE 1 Use `door_bind()` to create private server pools for two doors.

The following example shows the use of `door_bind()` to create private server pools for two doors, `d1` and `d2`. Function `my_create()` is called when a new server thread is needed; it creates a thread running function, `my_server_create()`, which binds itself to one of the two doors.

```
#include <door.h>
#include <thread.h>
#include <pthread.h>
thread_key_t door_key;
int d1 = -1;
int d2 = -1;
cond_t cv;       /* statically initialized to zero */
mutex_t lock;    /* statically initialized to zero */

extern void foo(void *, char *, size_t, door_desc_t *, uint_t);
extern void bar(void *, char *, size_t, door_desc_t *, uint_t);

static void *
my_server_create(void *arg)
{
        /* wait for d1 & d2 to be initialized */
        mutex_lock(&lock);
        while (d1 == -1 || d2 == -1)
                cond_wait(&cv, &lock);
        mutex_unlock(&lock);

        if (arg == (void *)foo){
                /* bind thread with pool associated with d1 */
                thr_setspecific(door_key, (void *)foo);
                if (door_bind(d1) < 0) {
                        perror("door_bind"); exit (-1);
                }
        } else if (arg == (void *)bar) {
                /* bind thread with pool associated with d2 */
                thr_setspecific(door_key, (void *)bar);
                if (door_bind(d2) < 0) {
                /* bind thread to d2 thread pool */
                        perror("door_bind"); exit (-1);
                }
        }
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        door_return(NULL, 0, NULL, 0);  /* Wait for door invocation */
}

static void
my_create(door_info_t *dip)
```

EXAMPLE 1 Use `door_bind()` to create private server pools for two doors. *(Continued)*

```
{
        /* Pass the door identity information to create function */
        thr_create(NULL, 0, my_server_create, (void *)dip->di_proc,
                THR_BOUND | THR_DETACHED, NULL);
}

main()
{
        (void) door_server_create(my_create);
        if (thr_keycreate(&door_key, NULL) != 0) {
                perror("thr_keycreate");
                exit(1);
        }
        mutex_lock(&lock);
        d1 = door_create(foo, NULL, DOOR_PRIVATE); /* Private pool */
        d2 = door_create(bar, NULL, DOOR_PRIVATE); /* Private pool */
        cond_signal(&cv);
        mutex_unlock(&lock);
        while (1)
                pause();
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs
Interface Stability	Committed
MT-Level	Safe

See Also

`fork(2)`, `door_create(3C)`, `door_return(3C)`, `door_server_create(3C)`, `attributes(5)`

Name door_call – invoke the function associated with a door descriptor

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <door.h>

int door_call(int d, door_arg_t *params);
```

Description The `door_call()` function invokes the function associated with the door descriptor *d*, and passes the arguments (if any) specified in *params*. All of the *params* members are treated as in/out parameters during a door invocation and may be updated upon returning from a door call. Passing `NULL` for *params* indicates there are no arguments to be passed and no results expected.

Arguments are specified using the `data_ptr` and `desc_ptr` members of *params*. The size of the argument data in bytes is passed in `data_size` and the number of argument descriptors is passed in `desc_num`.

Results from the door invocation are placed in the buffer, `rbuf`. See `door_return(3C)`. The `data_ptr` and `desc_ptr` members of *params* are updated to reflect the location of the results within the `rbuf` buffer. The size of the data results and number of descriptors returned are updated in the `data_size` and `desc_num` members. It is acceptable to use the same buffer for input argument data and results, so `door_call()` may be called with `data_ptr` and `desc_ptr` pointing to the buffer `rbuf`.

If the results of a door invocation exceed the size of the buffer specified by `rsize`, the system automatically allocates a new buffer in the caller's address space and updates the `rbuf` and `rsize` members to reflect this location. In this case, the caller is responsible for reclaiming this area using `munmap(rbuf, rsize)` when the buffer is no longer required. See `munmap(2)`.

Descriptors passed in a `door_desc_t` structure are identified by the `d_attributes` member. The client marks the `d_attributes` member with the type of object being passed by logically OR-ing the value of object type. Currently, the only object type that can be passed or returned is a file descriptor, denoted by the `DOOR_DESCRIPTOR` attribute. Additionally, the `DOOR_RELEASE` attribute can be set, causing the descriptor to be closed in the caller's address space after it is passed to the target. The descriptor will be closed even if `door_call()` returns an error, unless that error is `EFAULT` or `EBADF`.

The `door_desc_t` structure includes the following members:

```
typedef struct {
        door_attr_t d_attributes;   /* Describes the parameter */
        union {
                struct {
                        int d_descriptor;   /* Descriptor */
                        door_id_t d_id;    /* Unique door id */
                        } d_desc;
                } d_data;
        } door_desc_t;
```

When file descriptors are passed or returned, a new descriptor is created in the target address space and the `d_descriptor` member in the target argument is updated to reflect the new descriptor. In addition, the system passes a system-wide unique number associated with each door in the `door_id` member and marks the `d_attributes` member with other attributes associated with a door including the following:

`DOOR_LOCAL`	The door received was created by this process using `door_create()`. See `door_create(3C)`.
`DOOR_PRIVATE`	The door received has a private pool of server threads associated with the door.
`DOOR_UNREF`	The door received is expecting an unreferenced notification.
`DOOR_UNREF_MULTI`	Similar to `DOOR_UNREF`, except multiple unreferenced notifications may be delivered for the same door.
`DOOR_REFUSE_DESC`	This door does not accept argument descriptors.
`DOOR_NO_CANCEL`	This door does not cancel the server thread upon client abort.
`DOOR_REVOKED`	The door received has been revoked by the server.

The `door_call()` function is not a restartable system call. It returns `EINTR` if a signal was caught and handled by this thread. If the door invocation is not idempotent the caller should mask any signals that may be generated during a `door_call()` operation. If the client aborts in the middle of a `door_call()` and the door was not created with the `DOOR_NO_CANCEL` flag, the server thread is notified using the POSIX (see `standards(5)`) thread cancellation mechanism. See `cancellation(5)`.

The descriptor returned from `door_create()` is marked as close on `exec(FD_CLOEXEC)`. Information about a door is available for all clients of a door using `door_info()`. Applications concerned with security should not place secure information in door data that is accessible by `door_info()`. In particular, secure data should not be stored in the data item *cookie*. See `door_info(3C)`.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `door_call()` function will fail if:

`E2BIG`	Arguments were too big for server thread stack.
`EAGAIN`	Server was out of available resources.
`EBADF`	Invalid door descriptor was passed.
`EFAULT`	Argument pointers pointed outside the allocated address space.
`EINTR`	A signal was caught in the client, the client called `fork(2)`, or the server exited during invocation.

EINVAL	Bad arguments were passed.
EMFILE	The client or server has too many open descriptors.
ENFILE	The *desc_num* argument is larger than the door's DOOR_PARAM_DESC_MAX parameter (see door_getparam(3C)), and the door does not have the DOOR_REFUSE_DESC set.
ENOBUFS	The *data_size* argument is larger than the door's DOOR_PARAM_DATA_MAX parameter, or smaller than the door's DOOR_PARAM_DATA_MIN parameter (see door_getparam(3C)).
ENOTSUP	The *desc_num* argument is non-zero and the door has the DOOR_REFUSE_DESC flag set.
EOVERFLOW	System could not create overflow area in caller for results.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs
Interface Stability	Committed
MT-Level	Safe

See Also

munmap(2), door_create(3C), door_getparam(3C), door_info(3C), door_return(3C), libdoor(3LIB), attributes(5), cancellation(5), standards(5)

Name door_create – create a door descriptor

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <door.h>

int door_create(void (*server_procedure) (void *cookie, char *argp,
     size_t arg_size, door_desc_t *dp, uint_t n_desc), void *cookie,
     uint_t attributes);
```

Description The `door_create()` function creates a door descriptor that describes the procedure specified by the function *server_procedure*. The data item, *cookie*, is associated with the door descriptor, and is passed as an argument to the invoked function *server_procedure* during `door_call(3C)` invocations. Other arguments passed to *server_procedure* from an associated `door_call()` are placed on the stack and include *argp* and *dp*. The *argp* argument points to *arg_size* bytes of data and the *dp* argument points to *n_desc* `door_desc_t` structures. The *attributes* argument specifies attributes associated with the newly created door. Valid values for *attributes* are constructed by OR-ing one or more of the following values:

`DOOR_UNREF`
: Delivers a special invocation on the door when the number of descriptors that refer to this door drops to one. In order to trigger this condition, more than one descriptor must have referred to this door at some time. `DOOR_UNREF_DATA` designates an unreferenced invocation, as the *argp* argument passed to *server_procedure*. In the case of an unreferenced invocation, the values for *arg_size*, *dp* and *n_did* are `0`. Only one unreferenced invocation is delivered on behalf of a door.

`DOOR_UNREF_MULTI`
: Similar to `DOOR_UNREF`, except multiple unreferenced invocations can be delivered on the same door if the number of descriptors referring to the door drops to one more than once. Since an additional reference may have been passed by the time an unreferenced invocation arrives, the `DOOR_IS_UNREF` attribute returned by the `door_info(3C)` call can be used to determine if the door is still unreferenced.

`DOOR_PRIVATE`
: Maintains a separate pool of server threads on behalf of the door. Server threads are associated with a door's private server pool using `door_bind(3C)`.

`DOOR_REFUSE_DESC`
: Any attempt to call `door_call(3C)` on this door with argument descriptors will fail with `ENOTSUP`. When this flag is set, the door's server procedure will always be invoked with an *n_desc* argument of 0.

`DOOR_NO_CANCEL`
: Clients which abort calls to `door_call()` on this door will not cause the cancellation of the server thread handling the request. See `cancellation(5)`.

The descriptor returned from `door_create()` will be marked as close on exec (`FD_CLOEXEC`). Information about a door is available for all clients of a door using `door_info()`. Applications

concerned with security should not place secure information in door data that is accessible by `door_info()`. In particular, secure data should not be stored in the data item *cookie*.

By default, additional threads are created as needed to handle concurrent `door_call()` invocations. See `door_server_create(3C)` for information on how to change this behavior.

A process can advertise a door in the file system name space using `fattach(3C)`.

After creation, `door_setparam(3C)` can be used to set limits on the amount of data and descriptors clients can send over the door.

Return Values Upon successful completion, `door_create()` returns a non-negative value. Otherwise, `door_create` returns `−1` and sets `errno` to indicate the error.

Errors The `door_create()` function will fail if:

`EINVAL` Invalid attributes are passed.

`EMFILE` The process has too many open descriptors.

Examples **EXAMPLE 1** Create a door and use `fattach()` to advertise the door in the file system namespace.

The following example creates a door and uses `fattach()` to advertise the door in the file system namespace.

```
void
server(void *cookie, char *argp, size_t arg_size, door_desc_t *dp,
    uint_t n_desc)
{
    door_return(NULL, 0, NULL, 0);
    /* NOTREACHED */
}

int
main(int argc, char *argv[])
{
    int did;
    struct stat buf;

    if ((did = door_create(server, 0, 0)) < 0) {
        perror("door_create");
        exit(1);
    }

    /* make sure file system location exists */
    if (stat("/tmp/door", &buf) < 0) {
        int newfd;
        if ((newfd = creat("/tmp/door", 0444)) < 0) {
            perror("creat");
            exit(1);
```

EXAMPLE 1 Create a door and use `fattach()` to advertise the door in the file system namespace. *(Continued)*

```
        }
        (void) close(newfd);
    }

    /* make sure nothing else is attached */
    (void) fdetach("/tmp/door");

    /* attach to file system */
    if (fattach(did, "/tmp/door") < 0) {
        perror("fattach");
        exit(2);
    }
    [...]
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs
Interface Stability	Committed
MT-Level	Safe

See Also

`door_bind(3C)`, `door_call(3C)`, `door_info(3C)`, `door_revoke(3C)`, `door_setparam(3C)`, `door_server_create(3C)`, `fattach(3C)`, `libdoor(3LIB)`, `attributes(5)`, `cancellation(5)`

Name door_cred – return credential information associated with the client

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <door.h>

int door_cred(door_cred_t *info);
```

Description The `door_cred()` function returns credential information associated with the client (if any) of the current door invocation.

The contents of the *info* argument include the following fields:

```
uid_t   dc_euid;        /* Effective uid of client */
gid_t   dc_egid;        /* Effective gid of client */
uid_t   dc_ruid;        /* Real uid of client */
gid_t   dc_rgid;        /* Real gid of client */
pid_t   dc_pid;         /* pid of client */
```

The credential information associated with the client refers to the information from the immediate caller; not necessarily from the first thread in a chain of door calls.

Return Values Upon successful completion, `door_cred()` returns `0`. Otherwise, `door_cred()` returns `−1` and sets `errno` to indicate the error.

Errors The `door_cred()` function will fail if:

`EFAULT` The address of the *info* argument is invalid.

`EINVAL` There is no associated door client.

Usage The `door_cred()` function is obsolete. Applications should use the `door_ucred`(3C) function in place of `door_cred()`.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs
Interface Stability	Obsolete
MT-Level	Safe

See Also `door_call`(3C), `door_create`(3C), `door_ucred`(3C), `attributes`(5)

Name door_getparam, door_setparam – retrieve and set door parameters

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <door.h>

int door_getparam(int d, int param, size_t *out);

int door_setparam(int d, int param, size_t val);
```

Description The `door_getparam()` function retrieves the value of *param* for the door descriptor *d* and writes it through the pointer *out*. The `door_setparam()` function sets the value of *param* for the door descriptor *d* to *val*. The *param* argument names the parameter to view or change and can be one of the following values:

`DOOR_PARAM_DATA_MAX` This parameter represents the maximum amount of data that can be passed to the door routine. Any attempt to call `door_call(3C)` on a door with a *data_size* value larger than the door's `DOOR_PARAM_DATA_MAX` parameter will fail with `ENOBUFS`. At door creation time, this parameter is initialized to `SIZE_MAX` and can be set to any value from 0 to `SIZE_MAX`, inclusive. This parameter must be greater than or equal to the `DOOR_PARAM_DATA_MIN` parameter.

`DOOR_PARAM_DATA_MIN` This parameter represents the the minimum amount of data that can be passed to the door routine. Any attempt to call `door_call(3C)` on a door with a *data_size* value smaller than the door's `DOOR_PARAM_DATA_MIN` parameter will fail with `ENOBUFS`. At door creation time, this parameter is initialized to 0, and can be set to any value from 0 to `SIZE_MAX`, inclusive. This parameter must be less than or equal to the `DOOR_PARAM_DATA_MAX` parameter.

`DOOR_PARAM_DESC_MAX` This parameter represents the the maximum number of argument descriptors that can be passed to the door routine. Any attempt to call `door_call(3C)` on a door with a *desc_nu*m value larger than the door's `DOOR_PARAM_DESC_MAX` parameter will fail with `ENFILE`. If the door was created with the `DOOR_REFUSE_DESC` flag, this parameter is initialized to 0 and cannot be changed to any other value. Otherwise, it is initialized to `INT_MAX` and can be set to any value from 0 to `INT_MAX`, inclusive.

The `door_setparam()` function can only affect doors that were created by the current process.

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

Errors The `door_getparam()` function will fail if:

`EBADF` The *d* argument is not a door descriptor.

EFAULT	The *out* argument is not a valid address.
EINVAL	The *param* argument is not a recognized parameter.
EOVERFLOW	The value of the parameter is larger than the SIZE_MAX. This condition can occur only if the calling process is 32-bit and the door targets a 64-bit process or the kernel.

The door_setparam() function will fail if:

EBADF	The *d* argument is not a door descriptor or has been revoked.
EINVAL	The *param* argument is not a recognized parameter, or the requested change would make DOOR_PARAM_DATA_MIN greater than DOOR_PARAM_DATA_MAX.
ENOTSUP	The *param* argument is DOOR_PARAM_DESC_MAX, *d* was created with the DOOR_REFUSE_DESC flag, and *val* is not zero.
EPERM	The *d* argument was not created by this process.
ERANGE	The *val* argument is not in supported range of *param*.

Examples

EXAMPLE 1 Set up a door with a fixed request size.

```
typedef struct my_request {
     int request;
ar buffer[4096];
} my_request_t;

fd = door_create(my_handler, DOOR_REFUSE_DESC);
if (fd < 0)
     /* handle error */

if (door_setparam(fd, DOOR_PARAM_DATA_MIN,
    sizeof (my_request_t)) < 0 ||
    door_setparam(fd, DOOR_PARAM_DATA_MAX,
    sizeof (my_request_t)) < 0)
 /* handle error */

/*
 * the door will only accept door_call(3DOOR)s with a
 * data_size which is exactly sizeof (my_request_t).
 */
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also door_call(3C), door_create(3C), attributes(5)

Notes The parameters that can be manipulated by door_setparam() are not the only limitation on the size of requests. If the door server thread's stack size is not large enough to hold all of the data requested plus room for processing the request, the door call will fail with E2BIG.

The DOOR_PARAM_DATA_MIN parameter will not prevent DOOR_UNREF_DATA notifications from being sent to the door.

Name door_info – return information associated with a door descriptor

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <door.h>

int door_info(int d, struct door_info *info);
```

Description The `door_info()` function returns information associated with a door descriptor. It obtains information about the door descriptor *d* and places the information that is relevant to the door in the structure pointed to by the *info* argument.

The `door_info` structure pointed to by the *info* argument contains the following members:

```
pid_t           di_target;      /* door server pid */
door_ptr_t      di_proc;        /* server function */
door_ptr_t      di_data;        /* data cookie for invocation */
door_attr_t     di_attributes;  /* door attributes */
door_id_t       di_uniquifier;  /* unique id among all doors */
```

The `di_target` member is the process ID of the door server, or −1 if the door server process has exited.

The values for `di_attributes` may be composed of the following:

`DOOR_LOCAL`	The door descriptor refers to a service procedure in this process.
`DOOR_UNREF`	The door has requested notification when all but the last reference has gone away.
`DOOR_UNREF_MULTI`	Similar to `DOOR_UNREF`, except multiple unreferenced notifications may be delivered for this door.
`DOOR_IS_UNREF`	There is currently only one descriptor referring to the door.
`DOOR_REFUSE_DESC`	The door refuses any attempt to `door_call(3C)` it with argument descriptors.
`DOOR_NO_CANCEL`	Clients who abort a `door_call(3C)` call on this door will not cause the `cancellation(5)` of the server thread handling the request.
`DOOR_REVOKED`	The door descriptor refers to a door that has been revoked.
`DOOR_PRIVATE`	The door has a separate pool of server threads associated with it.

The `di_proc` and `di_data` members are returned as `door_ptr_t` objects rather than `void *` pointers to allow clients and servers to interoperate in environments where the pointer sizes may vary in size (for example, 32-bit clients and 64-bit servers). Each door has a system-wide unique number associated with it that is set when the door is created by `door_create()`. This number is returned in `di_uniquifier`.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `door_info()` function will fail if:

`EFAULT` The address of argument *info* is an invalid address.

`EBADF` *d* is not a door descriptor.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs
Interface Stability	Committed
MT-Level	Safe

See Also `door_bind(3C)`, `door_call(3C)`, `door_create(3C)`, `door_server_create(3C)`, `attributes(5)`, `cancellation(5)`

Name door_return – return from a door invocation

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <door.h>

int door_return(char *data_ptr, size_t data_size, door_desc_t *desc_ptr,
     uint_t num_desc);
```

Description The `door_return()` function returns from a door invocation. It returns control to the thread that issued the associated `door_call()` and blocks waiting for the next door invocation. See `door_call(3C)`. Results, if any, from the door invocation are passed back to the client in the buffers pointed to by *data_ptr* and *desc_ptr*. If there is not a client associated with the `door_return()`, the calling thread discards the results, releases any passed descriptors with the `DOOR_RELEASE` attribute, and blocks waiting for the next door invocation.

Return Values Upon successful completion, `door_return()` does not return to the calling process. Otherwise, `door_return()` returns −1 to the calling process and sets `errno` to indicate the error.

Errors The `door_return()` function fails and returns to the calling process if:

`E2BIG` Arguments were too big for client.

`EFAULT` The address of *data_ptr* or *desc_ptr* is invalid.

`EINVAL` Invalid `door_return()` arguments were passed or a thread is bound to a door that no longer exists.

`EMFILE` The client has too many open descriptors.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs
Interface Stability	Committed
MT-Level	Safe

See Also `door_call(3C)`,`attributes(5)`

Name door_revoke – revoke access to a door descriptor

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <door.h>

int door_revoke(int d);
```

Description The `door_revoke()` function revokes access to a door descriptor. Door descriptors are created with `door_create(3C)`. The `door_revoke()` function performs an implicit call to `close(2)`, marking the door descriptor *d* as invalid.

A door descriptor can be revoked only by the process that created it. Door invocations that are in progress during a `door_revoke()` invocation are allowed to complete normally.

Return Values Upon successful completion, `door_revoke()` returns `0`. Otherwise, `door_revoke()` returns −1 and sets `errno` to indicate the error.

Errors The `door_revoke()` function will fail if:

`EBADF` An invalid door descriptor was passed.

`EPERM` The door descriptor was not created by this process (with `door_create(3C)`).

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs
Interface Stability	Committed
MT-Level	Safe

See Also `close(2)`, `door_create(3C)`, `attributes(5)`

Name door_server_create – specify an alternative door server thread creation function

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <door.h>

void (*) () door_server_create(void (*create_proc)(door_info_t*));
```

Description Normally, the doors library creates new door server threads in response to incoming concurrent door invocations automatically. There is no pre-defined upper limit on the number of server threads that the system creates in response to incoming invocations (1 server thread for each active door invocation). These threads are created with the default thread stack size and POSIX (see standards(5)) threads cancellation disabled. The created threads also have the THR_BOUND | THR_DETACHED attributes for Solaris threads and the PTHREAD_SCOPE_SYSTEM | PTHREAD_CREATE_DETACHED attributes for POSIX threads. The signal disposition, and scheduling class of the newly created thread are inherited from the calling thread (initially from the thread calling door_create(), and subsequently from the current active door server thread).

The door_server_create() function allows control over the creation of server threads needed for door invocations. The procedure *create_proc* is called every time the available server thread pool is depleted. In the case of private server pools associated with a door (see the DOOR_PRIVATE attribute in door_create()), information on which pool is depleted is passed to the create function in the form of a door_info_t structure. The di_proc and di_data members of the door_info_t structure can be used as a door identifier associated with the depleted pool. The *create_proc* procedure may limit the number of server threads created and may also create server threads with appropriate attributes (stack size, thread-specific data, POSIX thread cancellation, signal mask, scheduling attributes, and so forth) for use with door invocations.

The overall amount of data and argument descriptors that can be sent through a door is limited by both the server thread's stack size and by the parameters of the door itself. See door_setparam(3C).

The specified server creation function should create user level threads using thr_create() with the THR_BOUND flag, or in the case of POSIX threads, pthread_create() with the PTHREAD_SCOPE_SYSTEM attribute. The server threads make themselves available for incoming door invocations on this process by issuing a door_return(NULL, 0, NULL, 0). In this case, the door_return() arguments are ignored. See door_return(3C) and thr_create(3C).

The server threads created by default are enabled for POSIX thread cancellations which may lead to unexpected thread terminations while holding resources (such as locks) if the client aborts the associated door_call(). See door_call(3C). Unless the server code is truly interested in notifications of client aborts during a door invocation and is prepared to handle such notifications using cancellation handlers, POSIX thread cancellation should be disabled for server threads using pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, NULL). If all doors are created with the DOOR_NO_CANCEL flag (see door_create(3C)), the threads will never be cancelled by an aborted door_call() call

The *create_proc* procedure need not create any additional server threads if there is at least one server thread currently active in the process (perhaps handling another door invocation) or it may create as many as seen fit each time it is called. If there are no available server threads during an incoming door invocation, the associated `door_call()` blocks until a server thread becomes available. The *create_proc* procedure must be MT-Safe.

Return Values

Upon successful completion, `door_server_create()` returns a pointer to the previous server creation function. This function has no failure mode (it cannot fail).

Examples

EXAMPLE 1 Creating door server threads.

The following example creates door server threads with cancellation disabled and an 8k stack instead of the default stack size:

```
#include <door.h>
#include <pthread.h>
#include <thread.h>

void *
my_thread(void *arg)
{
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        door_return(NULL, 0, NULL, 0);
}
void
my_create(door_info_t *dip)
{
        thr_create(NULL, 8192, my_thread, NULL,
                   THR_BOUND | THR_DETACHED, NULL);
}
main( )
{
        (void)door_server_create(my_create);
         . . .
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs
Interface Stability	Committed
MT-Level	Safe

See Also door_bind(3C), door_call(3C), door_create(3C), door_return(3C), pthread_create(3C), pthread_setcancelstate(3C), thr_create(3C), attributes(5), cancellation(5), standards(5)

Name door_ucred – return credential information associated with the client

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <door.h>

int door_ucred(ucred_t **info);
```

Description The `door_ucred()` function returns credential information associated with the client, if any, of the current door invocation.

When successful, `door_ucred()` writes a pointer to a user credential to the location pointed to by *info* if that location was previously `NULL`. If that location was non-null, `door_ucred()` assumes that *info* points to a previously allocated `ucred_t` which is then reused. The location pointed to by *info* can be used multiple times before being freed. The value returned in *info* must be freed using `ucred_free(3C)`.

The resulting user credential includes information about the effective user and group ID, the real user and group ID, all privilege sets and the calling PID.

The credential information associated with the client refers to the information from the immediate caller, not necessarily from the first thread in a chain of door calls.

Return Values Upon successful completion, `door_ucred()` returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error, in which case the memory location pointed to by the *info* argument is unchanged.

Errors The `door_ucred()` function will fail if:

`EAGAIN` The location pointed to by *info* was `NULL` and allocating memory sufficient to hold a ucred failed.

`EFAULT` The address of the *info* argument is invalid.

`EINVAL` There is no associated door client.

`ENOMEM` The location pointed to by *info* was `NULL` and allocating memory sufficient to hold a ucred failed.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `door_call(3C)`, `door_create(3C)`, `ucred_get(3C)`, `attributes(5)`

Name door_xcreate – create a door descriptor for a private door with per-door control over thread creation

Synopsis

```
#include <door.h>

typedef void door_server_procedure_t(void *, char *, size_t,
     door_desc_t *, uint_t);

typedef int door_xcreate_server_func_t(door_info_t *,
     void *(*)(void *), void *, void *);

typedef void door_xcreate_thrsetup_func_t(void *);

int door_xcreate(door_server_procedure_t *server_proceduere,
     void *cookie, uint_t attributes,
     door_xcreate_server_func_t *thr_create_func,
     door_xcreate_thrsetup_func_t *thr_setup_func, void *crcookie,
     int nthread);
```

Description The `door_xcreate()` function creates a private door to the given *server_procedure*, with per-door control over the creation of threads that will service invocations of that door. A private door is a door that has a private pool of threads that service calls to that door alone; non-private doors share a pool of service threads (see `door_create`(3C)).

Creating private doors using `door_create()`

Prior to the introduction of `door_xcreate()`, a private door was created using `door_create()` specifying attributes including `DOOR_PRIVATE` after installing a suitable door server thread creation function using `door_server_create()`. During such a call to `door_create()`, the first server thread for that door is created by calling the door server function; you must therefore already have installed a custom door server creation function using `door_server_create()`. The custom server creation function is called at initial creation of a private door, and again whenever a new invocation uses the last available thread for that door. The function must decide whether it wants to increase the level of concurrency by creating an additional thread - if it decides not to then further invocations may have to wait for an existing active invocation to complete before they can proceed. Additional threads may be created using whatever thread attributes are desired in the application, and the application must specify a thread start function (to `thr_create`(3C) or `pthread_create`(3C)) which will perform a `door_bind()` to the newly-created door before calling `door_return`(`NULL`, 0, `NULL`, 0) to enter service. See `door_server_create`(3C) and `door_bind`(3C) for more information and for an example.

This “legacy” private door API is adequate for many uses, but has some limitations:

- The server thread creation function appointed via the `door_server_create()` is shared by all doors in the process. Private doors are distinguished from non-private in that the `door_info_t` pointer argument to the thread creation function is non-null for private doors; from the `door_info_t` the associated door server procedure is available via the `di_proc` member.
- If a library wishes to create a private door of which the application is essentially unaware it has no option but to inherit any function appointed with `door_server_create()` which may render the library door inoperable.
- Newly-created server threads must bind to the door they will service, but the door file descriptor to quote in `door_bind()` is not available in the `door_info_t` structure we receive a pointer to. The door file descriptor is returned as the result of `door_create()`, but the initial service thread is created during the call to `door_create()`. This leads to complexity in the startup of the service thread, and tends to force the use of global variables for the door file descriptors as per the example in `door_bind()`.

Creating private doors with `door_xcreate()`

The `door_xcreate()` function is purpose-designed for the creation of private doors and simplifies their use by moving responsibility for binding the new server thread and synchronizing with it into a library-provided thread startup function:

- The first three arguments to `door_xcreate()` are as you would use in `door_create()`: the door *server_procedure*, a private cookie to pass to that procedure whenever it is invoked for this door, and desired door attributes. The `DOOR_PRIVATE` attribute is implicit, and an additional attribute of `DOOR_NO_DEPLETION_CB` is available.
- Four additional arguments specify a server thread creation function to use for this door (must not be `NULL`), a thread setup function for new server threads (can be `NULL`), a cookie to pass to those functions, and the initial number of threads to create for this door.
- The `door_xcreate_server_func_t()` for creating server threads has differing semantics to those of a `door_server_func_t()` used in `door_server_create()`. In addition to a `door_info_t` pointer it also receives as arguments a library-provided thread start function and thread start argument that it must use, and the private cookie registered in the call to `door_xcreate()`. The nominated `door_xcreate_server_func_t()` must:
  - Return 0 if no additional thread is to be created, for example if it decides the current level of concurrency is sufficient. When the server thread creation function is invoked as part of a depletion callback (as opposed to during initial `door_xcreate()`) the `door_info_t` `di_attributes` member includes `DOOR_DEPLETION_CB`.
  - Otherwise attempt to create exactly one new thread using `thr_create()` or `pthread_create()`, with whatever thread attributes (stack size) are desired and quoting the implementation-provided thread start function and opaque data cookie. If the call to `thr_create()` or `pthread_create()` is successful then return 1, otherwise return -1.
  - Do not call `door_bind()` or request to enter service via `door_return`(`NULL`, 0, `NULL`, 0).

As in door_server_create() new server threads must be created PTHREAD_SCOPE_SYSTEM and PTHREAD_CREATE_DETACHED for POSIX threads, and THR_BOUND and THR_DETACHED for Solaris threads. The signal disposition and scheduling class of newly-created threads are inherited from the calling thread, initially from the thread calling door_xcreate() and subsequently from the current active door server thread.

- The library-provided thread start function performs the following operations in the order presented:
  - Calls the door_xcreate_thrsetup_func_t() if it is not NULL, passing the *crcookie*. You can use this setup function to perform custom service thread configuration that must be done from the context of the new thread. Typically this is to configure cancellation preferences, and possibly to associate application thread-specific-data with the newly-created server thread.

    If thr_setup_func() was NULL then a default is applied which will configure the new thread with pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL) and pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL). If the server code is truly interested in notifications of client aborts during a door invocation then you will need to provide a thr_setup_func() that does not disable cancellations, and use pthread_cleanup_push(3C)and pthread_cleanup_pop(3C)as appropriate.
  - Binds the new thread to the door file descriptor using door_bind().
  - Synchronizes with door_xcreate() so that the new server thread is known to have successfully completed door_bind() before door_xcreate() returns.
- The number of service threads to create at initial door creation time can be controlled through the *nthread* argument to door_xcreate(). The nominated door_xcreate_server_func_t() will be called *nthread* times. All *nthread* new server threads must be created successfully (thr_create_func() returns 1 for each) and all must succeed in binding to the new door; if fewer than *nthread* threads are created, or fewer than *nthread* succeed in binding, then door_xcreate() fails and any threads that were created are made to exit.

  No artificial maximum value is imposed on the *nthread* argument: it may be as high as system resources and available virtual memory permit. There is a small amount of additional stack usage in the door_xcreate() stack frame for each thread - up to 16 bytes in a 64-bit application. If there is unsufficient room to extend the stack for this purpose then door_xcreate() fails with E2BIG.

  The door attributes that can be selected in the call to door_xcreate() are the same as in door_create(), with DOOR_PRIVATE implied and DOOR_NO_DEPLETION_CB added:

  DOOR_PRIVATE
  : It is not necessary to include this attribute. The door_xcreate() interfaces only creates private doors.

DOOR_NO_DEPLETION_CB
: Create the initial pool of *nthread* service threads, but do not perform further callbacks to the `thr_create_func()` for this door when the thread pool appears to be depleted at the start of a new door invocation. This allows you to select a fixed level of concurrency.

  Another `di_attribute` is defined during thread depletion callbacks:

DOOR_DEPLETION_CB
: This call to the server thread creation function is the result of a depletion callback. This attribute is not set when the function is called during initial `door_xcreate()`.

The descriptor returned from `door_xcreate()` will be marked as close on `exec` (`FD_CLOEXEC`). Information about a door is available for all clients of a door using `door_info`(3C). Applications concerned with security should not place secure information in door data that is accessible by `door_info()`. In particular, secure data should not be stored in the data item cookie.

A process can advertise a door in the file system name space using `fattach`(3C).

After creation, `door_setparam`(3C)can be used to set limits on the amount of data and descriptors clients can send over the door.

A door created with `door_xcreate()` may be revoked using `door_revoke`(3C). This closes the associated file descriptor, and acts as a barrier to further door invocations, but existing active invocations are not guaranteed to have completed before `door_revoke()` returns. Server threads bound to a revoked door do not wakeup or exit automatically when the door is revoked.

Return Values Upon successful completion, `door_xcreate()` returns a non-negative value. Otherwise, `door_xcreate()` returns -1 and sets `errno` to indicate the error.

Errors The `door_xcreate()` function will fail if:

`E2BIG` The requested *nthread* is too large. A small amount of stack space is required for each thread we must start and synchronize with. If extending the `door_xcreate()` stack by the required amount will exceed the stack bounds then `E2BIG` is returned.

`EBADF` The attempt to `door_bind()` within the library-provided thread start function failed.

`EINVAL` Invalid attributes are passed, *nthread* is less than 1, or `thr_create_func()` is `NULL`. This is also returned if `thr_create_func()` returns 0 (no thread creation attempted) during `door_xcreate()`.

`EMFILE` The process has too many open descriptors.

`ENOMEM` Insufficient memory condition while creating the door.

`ENOTSUP` A `door_xcreate()` call was attempted from a fork handler.

EPIPE A call to the nominated `thr_create_func()` returned -1 indicating that `pthread_create()` or `thr_create()` failed.

Examples

EXAMPLE 1 Create a private door with an initial pool of 10 server threads

Create a private door with an initial pool of 10 server threads. Threads are created with the minimum required attributes and there is no thread setup function. Use `fattach()` to advertise the door in the filesystem namespace.

```
static pthread_attr_t tattr;

/*
 * Simplest possible door_xcreate_server_func_t.  Always attempt to
 * create a thread, using the previously initialized attributes for
 * all threads. We must use the start function and argument provided,
 * and make no use of our private mycookie argument.
 */
int
thrcreatefunc(door_info_t *dip, void *(*startf)(void *),
    void *startfarg, void *mycookie)
{
        if (pthread_create(NULL, &tattr, startf, startfarg) != 0) {
                perror("thrcreatefunc: pthread_create");
                return (-1);
        }

        return (1);
}

/*
 * Dummy door server procedure - does no processing.
 */
void
door_proc(void *cookie, char *argp, size_t argsz, door_desc_t *descp,
    uint_t n)
{
    door_return (NULL, 0, NULL, 0);
}

int
main(int argc, char *argv[])
{
    struct stat buf;
    int did;

    /*
     * Setup thread attributes - minimum required.
     */
```

EXAMPLE 1 Create a private door with an initial pool of 10 server threads *(Continued)*

```
    (void) pthread_attr_init(&tattr);
    (void) pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED);
    (void) pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);

    /*
     * Create a private door with an initial pool of 10 server threads.
     */
    did = door_xcreate(door_proc, NULL, 0, thrcreatefunc, NULL, NULL,
                       10);

    if (did == -1) {
        perror("door_xcreate");
        exit(1);
    }

    if (stat(DOORPATH, &buf) < 0) {
        int newfd;

        if ((newfd = creat(DOORPATH, 0644)) < 0) {
            perror("creat");
            exit(1);
        }
        (void) close(newfd);
    }

    (void) fdetach(DOORPATH);

    (void) fdetach(DOORPATH);
    if (fattach(did, DOORPATH) < 0) {
        perror("fattach");
        exit(1);
    }

    (void) fprintf(stderr, "Pausing in main\n");
    (void) pause();
}
```

EXAMPLE 2 Create a private door with exactly one server thread and no callbacks for additional threads

Create a private door with exactly one server thread and no callbacks for additional threads. Use a server thread stacksize of 32K, and specify a thread setup function.

```
#define DOORPATH        "/tmp/grmdoor"

static pthread_attr_t tattr;
```

EXAMPLE 2 Create a private door with exactly one server thread and no callbacks for additional threads *(Continued)*

```
/*
 * Thread setup function - configuration that must be performed from
 * the conext of the new thread.  The mycookie argument is the
 * second-to-last argument from door_xcreate.
 */
void
thrsetupfunc(void *mycookie)
{
    /*
     * If a thread setup function is specified it must do the
     * following at minimum.
     */
    (void) pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

    /*
     * The default thread setup functions also performs the following
     * to disable thread cancellation notifications, so that server
     * threads are not cancelled when a client aborts a door call.
     * This is not a requirement.
     */
    (void) pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);

    /*
     * Now we can go on to perform other thread initialization,
     * for example to allocate and initialize some thread-specific data
     * for this thread;  for thread-specific data you can use a
       destructor function in pthread_key_create if you want to perform
       any actions if/when a door server thread exits.
     */
}

/*
 * The door_xcreate_server_func_t we will use for server thread
 * creation. The mycookie argument is the second-to-last argument
 * from door_xcreate.
 */
int
thrcreatefunc(door_info_t *dip, void *(*startf)(void *),
    void *startfarg, void *mycookie)
{
    if (pthread_create(NULL, &tattr, startf, startfarg) != 0) {
            perror("thrcreatefunc: pthread_create");
            return (-1);
    }
```

EXAMPLE 2 Create a private door with exactly one server thread and no callbacks for additional threads *(Continued)*

```
    return (1);
}

/*
 * Door procedure.  The cookie received here is the second arg to
 * door_xcreate.
 */
void
door_proc(void *cookie, char *argp, size_t argsz, door_desc_t *descp,
    uint_t n)
{
    (void) door_return(NULL, 0, NULL, 0);
}

int
main(int argc, char *argv[])
{
    struct stat buf;
    int did;

    /*
     * Configure thread attributes we will use in thrcreatefunc.
     * The PTHREAD_CREATE_DETACHED and PTHREAD_SCOPE_SYSTEM are
     * required.
     */
    (void) pthread_attr_init(&tattr);
    (void) pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED);
    (void) pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
    (void) pthread_attr_setstacksize(&tattr, 16 * 1024);

    /*
     * Create a private door with just one server thread and asking for
         * no further callbacks on thread pool depletion during an
         * invocation.
         */
    did = door_xcreate(door_proc, NULL, DOOR_NO_DEPLETION_CB,
        thrcreatefunc, thrsetupfunc, NULL, 1);

    if (did == -1) {
            perror("door_xcreate");
            exit(1);
    }
```

EXAMPLE 2 Create a private door with exactly one server thread and no callbacks for additional threads *(Continued)*

```
    if (stat(DOORPATH, &buf) < 0) {
            int newfd;

            if ((newfd = creat(DOORPATH, 0644)) < 0) {
                    perror("creat");
                    exit(1);
            }
            (void) close(newfd);
    }

    (void) fdetach(DOORPATH);
    if (fattach(did, DOORPATH) < 0) {
            perror("fattach");
            exit(1);
    }

    (void) fprintf(stderr, "Pausing in main\n");
    (void) pause();
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs
Interface Stability	Committed
MT-Level	Safe

See Also

`door_bind(3C)`, `door_call(3C)`, `door_create(3C)`, `door_info(3C)`, `door_revoke(3C)`, `door_server_create(3C)`, `door_setparam(3C)`, `fattach(3C)`, `libdoor(3LIB)`, `pthread_create(3C)`, `pthread_cleanup_pop(3C)`, `pthread_cleanup_push(3C)`, `thr_create(3C)`, `attributes(5)`, `cancellation(5)`

Name drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

Synopsis

```
#include <stdlib.h>

double drand48(void)

double erand48(unsigned short xi[3]);

long lrand48(void)

long nrand48(unsigned short xi[3]);

long mrand48(void)

long jrand48(unsigned short xi[3]);

void srand48(long seedval);

unsigned short *seed48(unsigned short seed16v[3]);

void lcong48(unsigned short param[7]);
```

Description This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions `drand48()` and `erand48()` return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions `lrand48()` and `nrand48()` return non-negative long integers uniformly distributed over the interval $[0, 2^{31}]$.

Functions `mrand48()` and `jrand48()` return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31}]$.

Functions `srand48()`, `seed48()`, and `lcong48()` are initialization entry points, one of which should be invoked before either `drand48()`, `lrand48()`, or `mrand48()` is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if `drand48()`, `lrand48()`, or `mrand48()` is called without a prior call to an initialization entry point.) Functions `erand48( )`, `nrand48( )`, and `jrand48()` do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m}\ n >= 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless `lcong48()` has been invoked, the multiplier value *a* and the addend value *c* are given by

$$a = 5\mathrm{DEECE66D}_{16} = 273673163155_8$$

$c = B_{16} = 13_8$

The value returned by any of the functions `drand48()`, `erand48()`, `lrand48()`, `nrand48()`, `mrand48()`, or `jrand48()` is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions `drand48()`, `lrand48()`, and `mrand48()` store the last 48-bit X_i generated in an internal buffer. X_i must be initialized prior to being invoked. The functions `erand48()`, `nrand48()`, and `jrand48()` require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. These routines do not have to be initialized; the calling program must place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions `erand48()`, `nrand48()`, and `jrand48()` allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, that is, the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function `srand48()` sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function `seed48()` sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by `seed48()`, and a pointer to this buffer is the value returned by `seed48()`. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize using `seed48()` when the program is restarted.

The initialization function `lcong48()` allows the user to specify the initial X_i the multiplier value *a*, and the addend value *c*. Argument array elements *param[0-2]* specify X_i, *param[3-5]* specify the multiplier *a*, and *param*[6] specifies the 16-bit addend *c*. After `lcong48()` has been called, a subsequent call to either `srand48()` or `seed48()` will restore the "standard" multiplier and addend values, *a* and *c*, specified above.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Safe

See Also

`rand(3C)`, `attributes(5)`, `standards(5)`

Name dup2 – duplicate an open file descriptor

Synopsis

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
```

Description The `dup2()` function causes the file descriptor *fildes2* to refer to the same file as *fildes*. The *fildes* argument is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than the current value for the maximum number of open file descriptors allowed the calling process. See `getrlimit(2)`. If *fildes2* already refers to an open file, not *fildes*, it is closed first. If *fildes2* refers to *fildes*, or if *fildes* is not a valid open file descriptor, *fildes2* will not be closed first.

The `dup2()` function is equivalent to `fcntl`(*fildes*, `F_DUP2FD`, *fildes2*).

Return Values Upon successful completion a non-negative integer representing the file descriptor is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `dup2()` function will fail if:

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

`EBADF` The *fildes2* argument is negative or is not less than the current resource limit returned by `getrlimit(RLIMIT_NOFILE, . . .)`.

`EINTR` A signal was caught during the `dup2()` call.

`EMFILE` The process has too many open files. See `fcntl(2)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `close(2)`, `creat(2)`, `exec(2)`, `fcntl(2)`, `getrlimit(2)`, `open(2)`, `pipe(2)`, `attributes(5)`, `standards(5)`

Name econvert, fconvert, gconvert, seconvert, sfconvert, sgconvert, qeconvert, qfconvert, qgconvert – output conversion

Synopsis

```
#include <floatingpoint.h>

char *econvert(double value, int ndigit, int *decpt, int *sign,
     char *buf);

char *fconvert(double value, int ndigit, int *decpt, int *sign,
     char *buf);

char *gconvert(double value, int ndigit, int trailing, char *buf);

char *seconvert(single *value, int ndigit, int *decpt, int *sign,
     char *buf);

char *sfconvert(single *value, int ndigit, int *decpt, int *sign,
     char *buf);

char *sgconvert(single *value, int ndigit, int trailing, char *buf);

char *qeconvert(quadruple *value, int ndigit, int *decpt, int *sign,
     char *buf);

char *qfconvert(quadruple *value, int ndigit, int *decpt, int *sign
     char *buf);

char *qgconvert(quadruple *value, int ndigit, int trailing, char *buf);
```

Description The `econvert()` function converts the *value* to a null-terminated string of *ndigit* ASCII digits in *buf* and returns a pointer to *buf*. *buf* should contain at least *ndigit+1* characters. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt*. Thus *buf* == "314" and **decpt* == 1 corresponds to the numerical value 3.14, while *buf* == "314" and **decpt* == −1 corresponds to the numerical value .0314. If the sign of the result is negative, the word pointed to by *sign* is nonzero; otherwise it is zero. The least significant digit is rounded.

The `fconvert()` function works much like `econvert()`, except that the correct digit has been rounded as if for `sprintf(%w.nf)` output with *n=ndigit* digits to the right of the decimal point. *ndigit* can be negative to indicate rounding to the left of the decimal point. The return value is a pointer to *buf*. *buf* should contain at least *310+max(0,ndigit)* characters to accomodate any double-precision *value*.

The `gconvert()` function converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It produces *ndigit* significant digits in fixed-decimal format, like `sprintf(%w.nf)`, if possible, and otherwise in floating-decimal format, like `sprintf(%w.ne)`; in either case *buf* is ready for printing, with sign and exponent. The result corresponds to that obtained by

```
(void) sprintf(buf,"%w.ng'',value) ;
```

If *trailing* = 0, trailing zeros and a trailing point are suppressed, as in `sprintf(%g)`. If *trailing* != 0, trailing zeros and a trailing point are retained, as in `sprintf(%#g)`.

The `seconvert()`, `sfconvert()`, and `sgconvert()` functions are single-precision versions of these functions, and are more efficient than the corresponding double-precision versions. A pointer rather than the value itself is passed to avoid C's usual conversion of single-precision arguments to double.

The `qeconvert()`, `qfconvert()`, and `qgconvert()` functions are quadruple-precision versions of these functions. The `qfconvert()` function can overflow the *decimal_record* field *ds* if *value* is too large. In that case, *buf*[0] is set to zero.

The `ecvt()`, `fcvt()` and `gcvt()` functions are versions of `econvert()`, `fconvert()`, and `gconvert()`, respectively, that are documented on the ecvt(3C) manual page. They constitute the default implementation of these functions and conform to the X/Open CAE Specification, System Interfaces and Headers, Issue 4, Version 2.

Usage IEEE Infinities and NaNs are treated similarly by these functions. "NaN" is returned for NaN, and "Inf" or "Infinity" for Infinity. The longer form is produced when *ndigit* >= 8.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also ecvt(3C),`sprintf(3C)`, `attributes(5)`

Name ecvt, fcvt, gcvt – convert floating-point number to string

Synopsis

```
#include <stdlib.h>

char *ecvt(double value, int ndigit, int *restrict decpt, int *restrict sign);

char *fcvt(double value, int ndigit, int *restrict decpt, int *restrict sign);

char *gcvt(double value, int ndigit, char *buf);
```

Description The `ecvt()`, `fcvt()` and `gcvt()` functions convert floating-point numbers to null-terminated strings.

`ecvt()` The `ecvt()` function converts *value* to a null-terminated string of *ndigit* digits (where *ndigit* is reduced to an unspecified limit determined by the precision of a `double`) and returns a pointer to the string. The high-order digit is non-zero, unless the value is 0. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored in the integer pointed to by *decpt* (negative means to the left of the returned digits). The radix character is not included in the returned string. If the sign of the result is negative, the integer pointed to by *sign* is non-zero, otherwise it is 0.

If the converted value is out of range or is not representable, the contents of the returned string are unspecified.

`fcvt()` The `fcvt()` function is identical to `ecvt()` except that *ndigit* specifies the number of digits desired after the radix point. The total number of digits in the result string is restricted to an unspecified limit as determined by the precision of a `double`.

`gcvt()` The `gcvt()` function converts *value* to a null-terminated string (similar to that of the `%g` format of `printf(3C)`) in the array pointed to by *buf* and returns *buf*. It produces *ndigit* significant digits (limited to an unspecified value determined by the precision of a `double`) in `%f` if possible, or `%e` (scientific notation) otherwise. A minus sign is included in the returned string if *value* is less than 0. A radix character is included in the returned string if *value* is not a whole number. Trailing zeros are suppressed where *value* is not a whole number. The radix character is determined by the current locale. If `setlocale(3C)` has not been called successfully, the default locale, POSIX, is used. The default locale specifies a period (`.`) as the radix character. The `LC_NUMERIC` category determines the value of the radix character within the current locale.

Return Values The `ecvt()` and `fcvt()` functions return a pointer to a null-terminated string of digits.

The `gcvt()` function returns *buf*.

Errors No errors are defined.

Usage The return values from `ecvt()` and `fcvt()` might point to thread-specific data that can be overwritten by subsequent calls to these functions by the same thread.

For portability to implementations conforming to earlier versions of Solaris, `sprintf(3C)` is preferred over this function.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See standards(5).

See Also printf(3C), setlocale(3C), sprintf(3C), attributes(5), standards(5)

Name enable_extended_FILE_stdio – enable extended FILE facility within standard I/O

Synopsis

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

int enable_extended_FILE_stdio(int low_fd, int signal_action);
```

Description The `enable_extended_FILE_stdio()` function enables the use of the extended FILE facility (see NOTES) and determines which, if any, signal will be sent when an application uses `FILE->_file` inappropriately.

The *low_fd* argument specifies the lowest file descriptor in the range 3 through 255 that the application wants to be selected as the unallocatable file descriptor. File descriptors 0, 1, and 2 cannot be used because they are reserved for use as the default file descriptors underlying the `stdin`, `stdout`, and `stderr` standard I/O streams. The *low_fd* argument can also be set to −1 to request that `enable_extended_FILE_stdio()` select a "reasonable" unallocatable file descriptor. In this case, `enable_extended_FILE_stdio()` will first attempt to reserve a relatively large file descriptor, but will keep trying to find an unallocatable file descriptor until it is known that no file descriptor can be reserved.

The *signal_action* argument specifies the signal that will be sent to the process when the unallocatable file descriptor is used as a file descriptor argument to any system call except `close(2)`. If *signal_action* is −1, the default signal (`SIGABRT`) will be sent. If *signal_action* is 0, no signal will be sent. Otherwise, the signal specified by *signal_action* will be sent.

The `enable_extended_FILE_stdio()` function calls

```
unallocatablefd = fcntl(low_fd, F_BADFD, action);
```

to reserve the unallocatable file descriptor and set the signal to be sent if the unallocatable file descriptor is used in a system call. If the `fcntl(2)` call succeeds, the extended FILE facility is enabled and the unallocatable file descriptor is saved for later use by the standard I/O functions. When an attempt is made to open a standard I/O stream (see `fdopen(3C)`, `fopen(3C)`, and `popen(3C)`) with an underlying file descriptor greater than 255, the file descriptor is stored in an auxiliary location and the field formerly known as `FILE->_file` is set to the unallocatable file descriptor.

If the file descriptor limit for the process is less than or equal to 256 (the system default), the application needs to raise the limit (see `getrlimit(2)`) for the extended FILE facility to be useful. The `enable_extended_FILE_stdio()` function does not attempt to change the file descriptor limit.

This function is used by the `extendedFILE(5)` preloadable library to enable the extended FILE facility.

Return Values Upon successful completion, `enable_extended_FILE_stdio()` returns 0. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `enable_extended_FILE_stdio()` function will fail if:

`EAGAIN` All file descriptors in the inclusive range 3 through 255 refer to files that are currently open in the process.

`EBADF` The *low_fd* argument is greater than 255, or is less than 3 and not equal to -1.

`EEXIST` A file descriptor has already been marked by an earlier call to `fcntl()`.

`EINVAL` The *signal_action* argument is not −1, is not 0, and is not a valid signal number.

Usage The `enable_extended_FILE_stdio()` function is available only in the 32-bit compilation environment.

The `fdopen(3C)`, `fopen(3C)`, and `popen(3C)` functions all enable the use of the extended FILE facility. For source changes, a trailing `F` character in the *mode* argument can be used with any of these functions if the `FILE` **fptr* is used only within the context of a single function or group of functions and not meant to be returned to a caller. All of the source code to the application must then be recompiled, thereby exposing any improper usage of the `FILE` structure fields.

The `F` character must not be used if the `FILE` **fptr* is to be returned to a caller. The calling application might not understand how to process it. Alternatively, the `enable_extended_FILE_stdio()` function can be used at a higher level in the code.

Use `extendedFILE(5)` for binary relief.

Examples **EXAMPLE 1** Increase the file limit and enable the extended FILE facility.

The following example demonstrates how to programmatically increase the file limit and enable extended FILE facility.

```
(void) getrlimit(RLIMIT_NOFILE, &rlp);
rlp.rlim_cur = 1000;  /* set the desired number of file descriptors */
retval = setrlimit(RLIMIT_NOFILE, &lrp);
if (retval == -1) {
    /* error */
}

/* enable extended FILE facility */
retval = enable_extended_FILE_stdio(-1, SIGABRT);
if (retval == -1) {
    /* error */
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library (32-bit)
Interface Stability	Committed
MT-Level	Safe

See Also close(2), fcntl(2), getrlimit(2), fdopen(3C), fopen(3C), popen(3C), signal.h(3HEAD), stdio(3C), attributes(5), extendedFILE(5)

Notes Historically, 32-bit Solaris applications have been limited to using only the file descriptors 0 through 255 with the standard I/O functions (see stdio(3C)) in the C library. The extended FILE facility allows well-behaved 32-bit applications to use any valid file descriptor with the standard I/O functions.

For the purposes of the extended FILE facility, a well-behaved application is one that:

- does not directly access any fields in the FILE structure pointed to by the FILE pointer associated with any standard I/O stream,
- checks all return values from standard I/O functions for error conditions, and
- behaves appropriately when an error condition is reported.

The extended FILE facility generates EBADF error returns and optionally delivers a signal to the calling process on most attempts to use the file descriptor formerly stored in FILE->*_file* as an argument to a system call when a file descriptor value greater than 255 is being used to access the file underlying the corresponding FILE pointer. The only exception is that calls to the close() system call will return an EBADF error in this case, but will not deliver the signal. The FILE->*_file* has been renamed to help applications quickly detect code that needs to be updated.

The extended FILE facility should only be used by well-behaved applications. Although the extended FILE facility reports errors, applications that directly reference FILE->*_file* should be updated to use public interfaces rather than rely on implementation details that no longer work as the application expects (see __fbufsize(3C) and fileno(3C).

This facility takes great care to avoid problems in well-behaved applications while maintaining maximum compatibility. It also attempts to catch dangerous behavior in applications that are not well-behaved as soon as possible and to notify those applications as soon as bad behavior is detected.

There are, however, limitations. For example, if an application enables this facility and is linked with an object file that had a standard I/O stream using an extended FILE pointer, and then used the sequence

```
(void) close(FILE->_file);
FILE->_file = myfd;
```

to attempt to change the file descriptor associated with the stream, undesired results can occur. The `close()` function will fail, but since this usage ignores the return status, the application proceeds to perform low level I/O on `FILE->`*_file* while calls to standard I/O functions would continue to use the original, extended `FILE` pointer. If the application continues using standard I/O functions after changing `FILE->`*_file*, silent data corruption could occur because the application thinks it has changed file descriptors with the above assignment but the actual standard I/O file descriptor is stored in the auxiliary location. The chances for corruption are even higher if *myfd* has a value greater than 255 and is truncated by the assignment to the 8-bit *_file* field.

Since the*_file* field has been renamed, attempts to recompile this code will fail. The application should be changed not to use this field in the `FILE` structure.

The application should not use this facility if it uses *_file* directly, including using the `fileno()` macro that was provided in `stdio.h`(3HEAD) in Solaris 2.0 through 2.7.

Name encrypt – encoding function

Synopsis

```
#include <crypt.h>

void encrypt(char block[64], int edflag);
```

Standard conforming

```
#include <unistd.h>

void encrypt(char block[64], int edflag);
```

Description The `encrypt()` function provides (rather primitive) access to the hashing algorithm employed by the `crypt(3C)` function. The key generated by `setkey(3C)` is used to encrypt the string *block* with `encrypt()`.

The *block* argument to `encrypt()` is an array of length 64 bytes containing only the bytes with numerical value of 0 and 1. The array is modified in place to a similar array using the key set by `setkey(3C)`. If *edflag* is 0, the argument is encoded. If *edflag* is 1, the argument may be decoded (see the `USAGE` section below); if the argument is not decoded, `errno` will be set to `ENOSYS`.

Return Values The `encrypt()` function returns no value.

Errors The `encrypt()` function will fail if:

`ENOSYS` The functionality is not supported on this implementation.

Usage In some environments, decoding may not be implemented. This is related to U.S. Government restrictions on encryption and decryption routines: the DES decryption algorithm cannot be exported outside the U.S.A. Historical practice has been to ship a different version of the encryption library without the decryption feature in the routines supplied. Thus the exported version of `encrypt()` does encoding but not decoding.

Because `encrypt()` does not return a value, applications wishing to check for errors should set `errno` to 0, call `encrypt()`, then test `errno` and, if it is non-zero, assume an error has occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `crypt(3C)`, `setkey(3C)`, `attributes(5)`

Name end, _end, etext, _etext, edata, _edata – last locations in program

Synopsis

```
extern int _etext;

extern int _edata;

extern int _end;
```

Description These names refer neither to routines nor to locations with interesting contents; only their addresses are meaningful.

`_etext` The address of `_etext` is the first location after the last read-only loadable segment.

`_edata` The address of `_edata` is the first location after the last read-write loadable segment.

`_end` If the address of `_edata` is greater than the address of `_etext`, the address of _end is same as the address of `_edata`.

If the address of `_etext` is greater than the address of `_edata`, the address of `_end` is set to the page boundary after the address pointed to by `_etext`.

Usage When execution begins, the program break (the first location beyond the data) coincides with `_end`, but the program break can be reset by the `brk(2)`, `malloc(3C)`, and the standard input/output library (see `stdio(3C)`), functions by the profile (`-p`) option of `cc`, and so on. Thus, the current value of the program break should be determined by `sbrk((char *)0)`.

References to `end`, `etext`, and `edata`, without a preceding underscore will be aliased to the associated symbol that begins with the underscore.

See Also `brk(2)`, `malloc(3C)`, `stdio(3C)`

Name err, verr, errx, verrx, warn, vwarn, warnx, vwarnx – formatted error messages

Synopsis

```
#include <err.h>

void err(int eval, const char *fmt, ...);

void verr(int eval, const char *fmt, va_list args);

void errx(int eval, const char *fmt, ...);

void verrx(int eval, const char *fmt, va_list args);

void warn(const char *fmt, ...);

void vwarn(const char *fmt, va_list args);

void warnx(const char *fmt, ...);

void vwarnx(const char *fmt, va_list args);
```

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

The `err()`, `verr()`, `errx()`, and `verrx()` functions do not return, but instead cause the program to terminate with the status value given by the argument status.

Examples **EXAMPLE 1** Display the current `errno` information string and terminate with status indicating failure.

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

EXAMPLE 2 Display an error message and terminate with status indicating failure.

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

EXAMPLE 3 Warn of an error.

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

Warnings It is important never to pass a string with user-supplied data as a format without using '%s'. An attacker can put format specifiers in the string to mangle the stack, leading to a possible security hole. This holds true even if the string has been built "by hand" using a function like `snprintf(3C)`, as the resulting string can still contain user-supplied conversion specifiers for later interpolation by the `err()` and `warn()` functions.

Always be sure to use the proper secure idiom:

```
err(1, "%s", string);
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe with Exceptions

These functions are safe to use in multithreaded applications as long as `setlocale(3C)` is not being called to change the locale.

See Also `exit(3C)`, `getexecname(3C)`, `setlocale(3C)`, `strerror(3C)`, `attributes(5)`

Name euclen, euccol, eucscol – get byte length and display width of EUC characters

Synopsis

```
#include <euc.h>

int euclen(const unsigned char *s);

int euccol(const unsigned char *s);

int eucscol(const unsigned char *str);
```

Description The `euclen()` function returns the length in bytes of the Extended Unix Code (EUC) character pointed to by *s*, including single-shift characters, if present.

The `euccol()` function returns the screen column width of the EUC character pointed to by *s*.

The `eucscol()` function returns the screen column width of the EUC string pointed to by *str*.

For the `euclen()` and `euccol()`, functions, *s* points to the first byte of the character. This byte is examined to determine its codeset. The character type table for the current *locale* is used for codeset byte length and display width information.

Usage These functions will work only with EUC locales.

These functions can be used safely in multithreaded applications, as long as `setlocale(3C)` is not called to change the locale.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe with exceptions

See Also `getwidth(3C)`, `setlocale(3C)`, `attributes(5)`

Name exit, _exithandle – terminate process

Synopsis

```
#include <stdlib.h>

void exit(int status);

void _exithandle(void);
```

Description The `exit()` function terminates a process by calling first `_exithandle()` and then `_exit()` (see `exit(2)`).

The `_exithandle()` function calls any functions registered through the `atexit(3C)` function in the reverse order of their registration. This action includes executing all finalization code from the *.fini* sections of all objects that are part of the process.

The `_exithandle()` function is intended for use *only* with `_exit()`, and allows for specialized processing such as `dldump(3C)` to be performed. Normal process execution should not be continued after a call to `_exithandle()` has occurred, as internal data structures may have been torn down due to `atexit()` or *.fini* processing.

The symbols `EXIT_SUCCESS` and `EXIT_FAILURE` are defined in the header `<stdlib.h>` and may be used as the value of *status* to indicate successful or unsuccessful termination, respectively.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `exit(2)`, `atexit(3C)`, `dldump(3C)`, `attributes(5)`, `standards(5)`

Name fattach – attach a STREAMS– or doors-based file descriptor to an object in the file system name space

Synopsis

```
#include <stropts.h>

int fattach(int fildes, const char *path);
```

Description The `fattach()` function attaches a STREAMS- or doors-based file descriptor to an object in the file system name space, effectively associating a name with *fildes*. The *fildes* argument must be a valid open file descriptor representing a STREAMS or doors file. The *path* argument is a path name of an existing object and the user must have appropriate privileges or be the owner of the file and have write permissions. All subsequent operations on *path* will operate on the STREAMS or doors file until the STREAMS or doors file is detached from the node. The *fildes* argument can be attached to more than one *path*, that is, a stream or door can have several names associated with it.

The attributes of the named stream or door (see `stat(2)`), are initialized as follows: the permissions, user ID, group ID, and times are set to those of *path*, the number of links is set to 1, and the size and device identifier are set to those of the streams or doors device associated with *fildes*. If any attributes of the named stream or door are subsequently changed (for example, `chmod(2)`), the attributes of the underlying object are not affected.

Return Values Upon successful completion, `fattach()` returns `0`. Otherwise it returns `−1` and sets `errno` to indicate an error.

Errors The `fattach()` function will fail if:

`EACCES`	The user is the owner of *path* but does not have write permissions on *path* or *fildes* is locked.
`EBADF`	The *fildes* argument is not a valid open file descriptor.
`EBUSY`	The *path* argument is currently a mount point or has a STREAMS or doors file descriptor attached to it.
`EINVAL`	The *path* argument is a file in a remotely mounted directory.
`EINVAL`	The *fildes* argument does not represent a STREAMS or doors file.
`ELOOP`	Too many symbolic links were encountered in translating *path*.
`ENAMETOOLONG`	The size of *path* exceeds {`PATH_MAX`}, or the component of a path name is longer than {`NAME_MAX`} while {`_POSIX_NO_TRUNC`} is in effect.
`ENOENT`	The *path* argument does not exist.
`ENOTDIR`	A component of a path prefix is not a directory.
`EPERM`	The effective user ID is not the owner of *path* or a user with the appropriate privileges.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also fdetach(1M), chmod(2), mount(2), stat(2), door_create(3C), fdetach(3C), isastream(3C), attributes(5), standards(5), streamio(7I)

STREAMS Programming Guide

Name __fbufsize, __flbf, __fpending, __fpurge, __freadable, __freading, __fsetlocking, __fwritable, __fwriting, _flushlbf – interfaces to stdio FILE structure

Synopsis

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

size_t __fbufsiz(FILE *stream);

int __flbf(FILE *stream);

size_t __fpending(FILE *stream);

void __fpurge(FILE *stream);

int __freadable(FILE *stream);

int __freading(FILE *stream);

int __fsetlocking(FILE *stream, int type);

int __fwritable(FILE *stream);

int __fwriting(FILE *stream);

void _flushlbf(void);
```

Description These functions provide portable access to the members of the stdio(3C) FILE structure.

The __fbufsize() function returns in bytes the size of the buffer currently in use by the given stream.

The __flbf() function returns non-zero if the stream is line-buffered.

The __fpending function returns in bytes the amount of output pending on a stream.

The __fpurge() function discards any pending buffered I/O on the stream.

The __freadable() function returns non-zero if it is possible to read from a stream.

The __freading() function returns non-zero if the file is open readonly, or if the last operation on the stream was a read operation such as fread(3C) or fgetc(3C). Otherwise it returns 0.

The __fsetlocking() function allows the type of locking performed by stdio on a given stream to be controlled by the programmer.

If *type* is FSETLOCKING_INTERNAL, stdio performs implicit locking around every operation on the given stream. This is the default system behavior on that stream.

If *type* is FSETLOCKING_BYCALLER, stdio assumes that the caller is responsible for maintaining the integrity of the stream in the face of access by multiple threads. If there is only one thread accessing the stream, nothing further needs to be done. If multiple threads are accessing the stream, then the caller can use the flockfile(), funlockfile(), and ftrylockfile()

functions described on the flockfile(3C) manual page to provide the appropriate locking. In both this and the case where *type* is FSETLOCKING_INTERNAL, __fsetlocking() returns the previous state of the stream.

If *type* is FSETLOCKING_QUERY, __fsetlocking() returns the current state of the stream without changing it.

The __fwritable() function returns non-zero if it is possible to write on a stream.

The __fwriting() function returns non-zero if the file is open write-only or append-only, or if the last operation on the stream was a write operation such as fwrite(3C) or fputc(3C). Otherwise it returns 0.

The _flushlbf() function flushes all line-buffered files. It is used when reading from a line-buffered file.

Usage Although the contents of the stdio FILE structure have always been private to the stdio implementation, some applications have needed to obtain information about a stdio stream that was not accessible through a supported interface. These applications have resorted to accessing fields of the FILE structure directly, rendering them possibly non-portable to new implementations of stdio, or more likely, preventing enhancements to stdio that would cause those applications to break.

In the 64-bit environment, the FILE structure is opaque. The functions described here are provided as a means of obtaining the information that up to now has been retrieved directly from the FILE structure. Because they are based on the needs of existing applications (such as mh and emacs), they may be extended as other programs are ported. Although they may still be non-portable to other operating systems, they will be compatible from each Solaris release to the next. Interfaces that are more portable are under development.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	__fsetlocking() is Unsafe; all others are MT-Safe
Interface Stability	Committed

See Also fgetc(3C), flockfile(3C), fputc(3C), fread(3C), fwrite(3C), stdio(3C), attributes(5)

Name fclose, fcloseall – close a stream

Synopsis

```
#include <stdio.h>

int fclose(FILE *stream);

int fcloseall(void);
```

Description The `fclose()` function causes the stream pointed to by *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream is written to the file; any unread buffered data is discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated.

The `fclose()` function marks for update the `st_ctime` and `st_mtime` fields of the underlying file if the stream is writable and if buffered data has not yet been written to the file. It will perform a `close(2)` operation on the file descriptor that is associated with the stream pointed to by *stream*.

After the call to `fclose()`, any use of *stream* causes undefined behavior.

The `fclose()` function is performed automatically for all open files upon calling `exit(2)`.

The `fcloseall()` function calls `fclose()` on all open streams.

Return Values Upon successful completion, 0 is returned. Otherwise, `EOF` is returned and `errno` is set to indicate the error.

Errors The `fclose()` function will fail if:

`EAGAIN` The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

`EBADF` The file descriptor underlying stream is not valid.

`EFBIG` An attempt was made to write a file that exceeds the maximum file size or the process's file size limit; or the file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

`EINTR` The `fclose()` function was interrupted by a signal.

`EIO` The process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU` and the process group of the process is orphaned.

`ENOSPC` There was no free space remaining on the device containing the file.

`EPIPE` An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the calling thread.

The `fclose()` function may fail if:

ENXIO A request was made of a non-existent device, or the request was beyond the limits of the device.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also close(2), exit(2), getrlimit(2), ulimit(2), fopen(3C), stdio(3C), attributes(5), standards(5)

Name fdatasync – synchronize a file's data

Synopsis

```
#include <unistd.h>

int fdatasync(int fildes);
```

Description The `fdatasync()` function forces all currently queued I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O completion state.

The functionality is as described for `fsync(3C)` (with the symbol `_XOPEN_REALTIME` defined), with the exception that all I/O operations are completed as defined for synchronised I/O data integrity completion.

Return Values If successful, the `fdatasync()` function returns `0`. Otherwise, the function returns `−1` and sets `errno` to indicate the error. If the `fdatasync()` function fails, outstanding I/O operations are not guaranteed to have been completed.

Errors The `fdatasync()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor open for writing.

`EINVAL` The system does not support synchronized I/O for this file.

`ENOSYS` The function `fdatasync()` is not supported by the system.

In the event that any of the queued I/O operations fail, `fdatasync()` returns the error conditions defined for `read(2)` and `write(2)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `fcntl(2)`, `open(2)`, `read(2)`, `write(2)`, `fsync(3C)`, `aio_fsync(3C)`, `fcntl.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name fdetach – detach a name from a STREAMS-based file descriptor

Synopsis

```
#include <stropts.h>

int fdetach(const char *path);
```

Description The `fdetach()` function detaches a STREAMS-based file from the file to which it was attached by a previous call to `fattach(3C)`. The *path* argument points to the pathname of the attached STREAMS file. The process must have appropriate privileges or be the owner of the file. A successful call to `fdetach()` causes all pathnames that named the attached STREAMS file to again name the file to which the STREAMS file was attached. All subsequent operations on *path* will operate on the underlying file and not on the STREAMS file.

All open file descriptions established while the STREAMS file was attached to the file referenced by *path*, will still refer to the STREAMS file after the `fdetach()` has taken effect.

If there are no open file descriptors or other references to the STREAMS file, then a successful call to `fdetach()` has the same effect as performing the last `close(2)` on the attached file.

Return Values Upon successful completion, `fdetach()` returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `fdetach()` function will fail if:

`EACCES`	Search permission is denied on a component of the path prefix.
`EPERM`	The effective user ID is not the owner of *path* and the process does not have appropriate privileges.
`ENOTDIR`	A component of the path prefix is not a directory.
`ENOENT`	A component of *path* does not name an existing file or *path* is an empty string.
`EINVAL`	The *path* argument names a file that is not currently attached.
`ENAMETOOLONG`	The size of a pathname exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
`ELOOP`	Too many symbolic links were encountered in resolving *path*.

The `fdetach()` function may fail if:

`ENAMETOOLONG`	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE

Interface Stability	Standard

See Also `fdetach`(1M), `close`(2), `fattach`(3C), `attributes`(5), `standards`(5), `streamio`(7I)

STREAMS Programming Guide

Name fdopen – associate a stream with a file descriptor

Synopsis

```
#include <stdio.h>

FILE *fdopen(int fildes, const char *mode);
```

Description The `fdopen()` function associates a stream with a file descriptor *fildes*.

The *mode* argument is a character string having one of the following values:

`r` or `rb`	Open a file for reading.
`w` or `wb`	Open a file for writing.
`a` or `ab`	Open a file for writing at end of file.
`r+`, `rb+` or `r+b`	Open a file for update (reading and writing).
`w+`, `wb+` or `w+b`	Open a file for update (reading and writing).
`a+`, `ab+` or `a+b`	Open a file for update (reading and writing) at end of file.

The meaning of these flags is exactly as specified for the `fopen(3C)` function, except that modes beginning with *w* do not cause truncation of the file. A trailing `F` character can also be included in the *mode* argument as described in `fopen(3C)` to enable extended FILE facility.

The mode of the stream must be allowed by the file access mode of the open file. The file position indicator associated with the new stream is set to the position indicated by the file offset associated with the file descriptor.

The `fdopen()` function preserves the offset maximum previously set for the open file description corresponding to *fildes*.

The error and end-of-file indicators for the stream are cleared. The `fdopen()` function may cause the `st_atime` field of the underlying file to be marked for update.

If *fildes* refers to a shared memory object, the result of the `fdopen()` function is unspecified.

Return Values Upon successful completion, `fdopen()` returns a pointer to a stream. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

The `fdopen()` function may fail and not set `errno` if there are no free `stdio` streams.

Errors The `fdopen()` function may fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

`EINVAL` The *mode* argument is not a valid mode.

`EMFILE` {`FOPEN_MAX`} streams are currently open in the calling process.

{`STREAM_MAX`} streams are currently open in the calling process.

ENOMEM There is insufficient space to allocate a buffer.

Usage A process is allowed to have at least {FOPEN_MAX} stdio streams open at a time. For 32-bit applications, however, the underlying ABIs formerly required that no file descriptor used to access the file underlying a stdio stream have a value greater than 255. To maintain binary compatibility with earlier Solaris releases, this limit still constrains 32-bit applications.

File descriptors are obtained from calls like open(2), dup(2), creat(2) or pipe(2), which open files but do not return streams. Streams are necessary input for almost all of the standard I/O library functions.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See below.

For all aspects of this function except the F character in the *mode* argument, see standards(5)

See Also creat(2), dup(2), open(2), pipe(2), fclose(3C), fopen(3C), attributes(5), standards(5)

Name ferror, feof, clearerr, fileno – stream status inquiries

Synopsis

```
#include <stdio.h>

int ferror(FILE *stream);

int feof(FILE *stream);

void clearerr(FILE *stream);

int fileno(FILE *stream);
```

Description The `ferror()` function returns a non-zero value when an error has previously occurred reading from or writing to the named *stream* (see `Intro(3)`). It returns `0` otherwise.

The `feof()` function returns a non-zero value when `EOF` has previously been detected reading the named input *stream*. It returns `0` otherwise.

The `clearerr()` function resets the error indicator and `EOF` indicator to 0 on the named *stream*.

The `fileno()` function returns the integer file descriptor associated with the named *stream*; see `open(2)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `open(2)`, `Intro(3)`, `fopen(3C)`, `stdio(3C)`, `attributes(5)`, `standards(5)`

Name fflush – flush a stream

Synopsis

```
#include <stdio.h>

int fflush(FILE *stream);
```

Description If *stream* points to an output stream or an update stream in which the most recent operation was not input, `fflush()` causes any unwritten data for that stream to be written to the file, and the `st_ctime` and `st_mtime` fields of the underlying file are marked for update.

If *stream* points to an input stream or an update stream into which the most recent operation was input, that stream is flushed if it is seekable and is not already at end-of-file. Flushing an input stream discards any buffered input and adjusts the file pointer such that the next input operation accesses the byte after the last one read. A stream is seekable if the underlying file is not a pipe, FIFO, socket, or TTY device.

If *stream* is a null pointer, `fflush()` performs this flushing action on all streams for which the behavior is defined above.

An input stream, seekable or non-seekable, can be flushed by explicitly calling `fflush()` with a non-null argument specifying that stream.

Return Values Upon successful completion, `fflush()` returns `0`. Otherwise, it returns `EOF` and sets `errno` to indicate the error.

Errors The `fflush()` function will fail if:

`EAGAIN` The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

`EBADF` The file descriptor underlying *stream* is not valid.

`EFBIG` An attempt was made to write a file that exceeds the maximum file size or the process's file size limit; or the file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

`EINTR` The `fflush()` function was interrupted by a signal.

`EIO` The process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU`, and the process group of the process is orphaned.

`ENOSPC` There was no free space remaining on the device containing the file.

`EPIPE` An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the calling process.

The `fflush()` function may fail if:

`ENXIO` A request was made of a non-existent device, or the request was beyond the limits of the device.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also getrlimit(2), ulimit(2), attributes(5), standards(5)

Name ffs, ffsl, ffsll, fls, flsl, flsll – find first or last bit set in a bit string

Synopsis

```
#include <strings.h>

int ffs(int value);

int ffsl(long value);

int ffsll(long long value);

int fls(int value);

int flsl(long value);

flsll(long long value);
```

Description The `ffs()`, `ffsl()`, and `ffsll()` functions find the first bit set in *value* and return the position of that bit.

The `fls()`, `fssl()`, and `flsll()` functions find the last bit set in *value* and return the position of that bit.

Bits are numbered starting at one (the least significant bit).

Return Values These functions return the position of the first bit set, or 0 if no bits are set in *value*.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `attributes(5)`, `standards(5)`

Name fgetattr, fsetattr, getattrat, setattrat – get and set system attributes

Synopsis

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

int fgetattr(int fildes, xattr_view_t view,nvlist_t **response);

int fsetattr(int fildes, xattr_view_t view,nvlist_t *request)

int getattrat(int fildes, xattr_view_t view, const char *filename,
     nvlist_t **response);

int setattrat(int fildes, xattr_view_t view, const char *filename,
     nvlist_t *request);
```

Description The `fgetattr()` function obtains an nvlist of system attribute information about an open file object specified by the file descriptor *fildes*, obtained from a successful `open(2)`, `creat(2)`, `dup(2)`, `fcntl(2)`, or `pipe(2)` function.

The `getattrat()` function first opens the extended attribute file specified by *filename* in the already opened file directory object specified by *fildes*. It then retrieves an nvlist of system attributes and their values from *filename*.

The *response* argument is allocated by either `fgetattr()` or `getattrat()`. The application must call `nvlist_free(3NVPAIR)` to deallocate the memory.

Upon successful completion, the nvlist will contain one nvpair for each of the system attributes associated with *view*. The list of views and the attributes associated with each view are listed below. Not all underlying file systems support all views and all attributes. The nvlist will not contain an nvpair for any attribute not supported by the underlying filesystem.

The `fsetattr()` function uses the nvlist pointed to by *request* to update one or more of the system attribute's information about an open file object specified by the file descriptor *fildes*, obtained from a successful `open()`, `creat()`, `dup()`, `fcntl()`, or `pipe()` function. The `setattrat()` function first opens the extended attribute file specified by *filename* in the already opened file directory object specified by *fildes*. It then uses the nvlist pointed to by *request* to update one or more of the system attributes of filename.

If completion is not successful then no system attribute information is updated.

The following chart lists the supported views, attributes, and data types for each view:

View	Attribute	Data type
XATTR_VIEW_READONLY	A_FSID	uint64_value
	A_OPAQUE	boolean_value

View	Attribute	Data type
	A_AV_SCANSTAMP	uint8_array[]
XATTR_VIEW_READWRITE	A_READONLY	boolean_value
	A_HIDDEN	boolean_value
	A_SYSTEM	boolean_value
	A_ARCHIVE	boolean_value
	A_CRTIME	uint64_array[2]
	A_NOUNLINK	boolean_value
	A_IMMUTABLE	boolean_value
	A_APPENDONLY	boolean_value
	A_NODUMP	boolean_value
	A_AV_QUARANTINED	boolean_value
	A_AV_MODIFIED	boolean_value
	A_OWNERSID	nvlist composed of uint32_value and string
	A_GROUPSID	nvlist composed of uint32_value and string

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

Errors The `fgetattr()`, `getattrat()`, `fsetattr()`, and `setattrat()`, functions will fail if:

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

`EINVAL` The underlying file system does not support extended file attributes.

`EIO` An error occurred while reading from the file system.

The `getattrat()` and `setattrat()` functions will fail if:

`EACCES` Search permission or write permission for *filename* is denied.

`ENOENT` The *filename* argument does not name an existing file in the extended attribute directory represented by *fildes*.

`EPERM` There are insufficient privileges to manipulate attributes.

Examples **EXAMPLE 1** Obtain an nvlist of readonly system attributes for an open file object.

Use `fgetattr()` to obtain an nvlist of the readonly system attributes for the open file object represented by file descriptor *fildes*.

EXAMPLE 1 Obtain an nvlist of readonly system attributes for an open file object. *(Continued)*

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

nvlist_t *response;
nvpair_t *pair = NULL;

if (fgetattr(fildes, XATTR_VIEW_READONLY, &response)) {
             exit(1);
}
while (pair = nvlist_next_nvpair(response, pair)) {
    .
    .
    .
}
nvlist_free(response);
```

EXAMPLE 2 Set the `A_READONLY` system attribute on an open file object.

Use `fsetattr()` to set the `A_OPAQUE` system attribute on the open file object represented by file descriptor *fildes*.

```
nvlist_t *request;
nvpair_t *pair = NULL;

if (nvlist_alloc(&request, NV_UNIQUE_NAME, 0) != 0) {
            exit(1);
}
if (nvlist_add_boolean_value(request, A_READONLY, 1) != 0) {
            exit(1);
}
if (fsetattr(fildes, XATTR_VIEW_READWRITE, request)) {
            exit(1);
}
```

EXAMPLE 3 Obtain an nvlist of the read/write system attributes for a file.

Use `getattrat()` to obtain an nvlist of the read/write system attributes for the file named `xattrfile` in the extended attribute directory of the open file represented by file descriptor *fildes*.

```
nvlist_t *response;
nvpair_t *pair = NULL;

if (getattrat(fildes, XATTR_VIEW_READWRITE, "file", &response)) {
             exit(1);
```

EXAMPLE 3 Obtain an nvlist of the read/write system attributes for a file. *(Continued)*

```
}
while (pair = nvlist_next_nvpair(response, pair)) {
    .
    .
    .
}
nvlist_free(response);
```

EXAMPLE 4 Set the `A_APPENDONLY` system attribute on a file.

Use `setattrat()` to set the `A_APPENDONLY` system attribute on the file named `file` in the extended attribute directory of the open file represented by file descriptor *fildes*.

```
nvlist_t *request;
nvpair_t *pair = NULL;

if (nvlist_alloc(&request, NV_UNIQUE_NAME, 0) != 0) {
            exit(1);
}
if (nvlist_add_boolean_value(request, A_APPENDONLY, 1) != 0) {
            exit(1);
}
if (setattrat(fildes, XATTR_VIEW_READWRITE, "file", request)) {
            exit(1);
     }
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`creat(2)`, `dup(2)`, `fcntl(2)`, `fstat(2)`, `fstatat(2)`, `open(2)`, `pipe(2)`, `libnvpair(3LIB)`, `attributes(5)`, `fsattr(5)`

Name fgetc, getc, getc_unlocked, getchar, getchar_unlocked, getw – get a byte from a stream

Synopsis

```
#include <stdio.h>

int fgetc(FILE *stream);

int getc(FILE *stream);

int getc_unlocked(FILE *stream);

int getchar(void);

int getchar_unlocked(void);

int getw(FILE *stream);
```

Description The `fgetc()` function obtains the next byte (if present) as an `unsigned char` converted to an `int`, from the input stream pointed to by *stream*, and advances the associated file position indicator for the stream (if defined).

For standard-conforming (see `standards(5)`) applications, if the end-of-file indicator for the stream is set, `fgetc()` returns `EOF` whether or not a next byte is present.

The `fgetc()` function may mark the `st_atime` field of the file associated with *stream* for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc()`, `fgets(3C)`, `fread(3C)`, `fscanf(3C)`, `getc()`, `getchar()`, `getdelim(3C)`, `getline(3C)`, `gets(3C)` or `scanf(3C)` using *stream* that returns data not supplied by a prior call to `ungetc(3C)` or `ungetwc(3C)`.

The `getc()` function is functionally identical to `fgetc()`, except that it is implemented as a macro. It runs faster than `fgetc()`, but it takes up more space per invocation and its name cannot be passed as an argument to a function call.

The `getchar()` routine is equivalent to `getc(stdin)`. It is implemented as a macro.

The `getc_unlocked()` and `getchar_unlocked()` routines are variants of `getc()` and `getchar()`, respectively, that do not lock the stream. It is the caller's responsibility to acquire the stream lock before calling these routines and releasing the lock afterwards; see `flockfile(3C)` and `stdio(3C)`. These routines are implemented as macros.

The `getw()` function reads the next word from the *stream*. The size of a word is the size of an `int` and may vary from environment to environment. The `getw()` function presumes no special alignment in the file.

The `getw()` function may mark the `st_atime` field of the file associated with *stream* for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc()`, `fgets(3C)`, `fread(3C)`, `getc()`, `getchar()`, `gets(3C)`, `fscanf(3C)` or `scanf(3C)` using *stream* that returns data not supplied by a prior call to `ungetc(3C)`.

Return Values Upon successful completion, `fgetc()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, and `getw()` return the next byte from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and these functions return `EOF`. For standard-conforming (see `standards(5)`) applications, if the end-of-file indicator for the stream is set, these functions return `EOF` whether or not the stream is at end-of-file. If a read error occurs, the error indicator for the stream is set, EOF is returned, and `errno` is set to indicate the error.

Errors The `fgetc()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, and `getw()` functions will fail if data needs to be read and:

`EAGAIN` The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the `fgetc()` operation.

`EBADF` The file descriptor underlying *stream* is not a valid file descriptor open for reading.

`EINTR` The read operation was terminated due to the receipt of a signal, and no data was transferred.

`EIO` A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group is orphaned. This error may also be generated for implementation-dependent reasons.

`EOVERFLOW` The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

The `fgetc()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, and `getw()` functions may fail if:

`ENOMEM` Insufficient storage space is available.

`ENXIO` A request was made of a non-existent device, or the request was outside the capabilities of the device.

Usage If the integer value returned by `fgetc()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, and `getw()` is stored into a variable of type `char` and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a variable of type `char` on widening to integer is implementation-dependent.

The `ferror(3C)` or `feof(3C)` functions must be used to distinguish between an error condition and an end-of-file condition.

Functions exist for the `getc()`, `getc_unlocked()`, `getchar()`, and `getchar_unlocked()` macros. To get the function form, the macro name must be undefined (for example, `#undef getc`).

When the macro forms are used, `getc()` and `getc_unlocked()` evaluate the *stream* argument more than once. In particular, `getc(*f++);` does not work sensibly. The `fgetc()` function should be used instead when evaluating the *stream* argument has side effects.

Because of possible differences in word length and byte ordering, files written using `getw()` are machine-dependent, and may not be read using `getw()` on a different processor.

The `getw()` function is inherently byte stream-oriented and is not tenable in the context of either multibyte character streams or wide-character streams. Application programmers are recommended to use one of the character-based input functions instead.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	`fgetc()`, `getc()`, `getc_unlocked()`, `getchar()`, and `getchar_unlocked()` are Standard.
MT-Level	See `NOTES` below.

See Also `Intro(3)`, `__fsetlocking(3C)`, `fclose(3C)`, `feof(3C)`, `fgets(3C)`, `fgetwc(3C)`, `fgetws(3C)`, `flockfile(3C)`, `fopen(3C)`, `fread(3C)`, `fscanf(3C)`, `getdelim(3C)`, `getline(3C)`, `gets(3C)`, `putc(3C)`, `scanf(3C)`, `stdio(3C)`, `ungetc(3C)`, `ungetwc(3C)`, `attributes(5)`, `standards(5)`

Notes The `fgetc()`, `getc()`, `getchar()`, and `getw()` routines are MT-Safe in multithreaded applications. The `getc_unlocked()` and `getchar_unlocked()` routines are unsafe in multithreaded applications.

Name fgetpos – get current file position information

Synopsis

```
#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);
```

Description The `fgetpos()` function stores the current value of the file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The value stored contains unspecified information usable by `fsetpos(3C)` for repositioning the stream to its position at the time of the call to `fgetpos()`.

Return Values Upon successful completion, `fgetpos()` returns `0`. Otherwise, it returns a non-zero value and sets `errno` to indicate the error.

Errors The `fgetpos()` function may fail if:

`EBADF` The file descriptor underlying *stream* is not valid.

`ESPIPE` The file descriptor underlying *stream* is associated with a pipe, a FIFO, or a socket.

`EOVERFLOW` The current value of the file position cannot be represented correctly in an object of type `fpos_t`.

Usage The `fgetpos()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `fopen(3C)`, `fsetpos(3C)`, `ftell(3C)`, `rewind(3C)`, `ungetc(3C)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Name fgetwc – get a wide-character code from a stream

Synopsis

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

wint_t fgetwc(FILE*stream);
```

Description The `fgetwc()` function obtains the next character (if present) from the input stream pointed to by *stream*, converts that to the corresponding wide-character code and advances the associated file position indicator for the stream (if defined).

If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

The `fgetwc()` function may mark the `st_atime` field of the file associated with *stream* for update. The `st_atime` field will be marked for update by the first successful execution of `fgetwc()`, `fgetc`(3C), `fgets`(3C), `fgetws`(3C), `fread`(3C), `fscanf`(3C), `getc`(3C), `getchar`(3C), `gets`(3C), or `scanf`(3C) using *stream* that returns data not supplied by a prior call to `ungetc`(3C) or `ungetwc`(3C).

Return Values Upon successful completion the `fgetwc()` function returns the wide-character code of the character read from the input stream pointed to by *stream* converted to a type `wint_t`.

For standard-conforming (see `standards`(5)) applications, if the end-of-file indicator for the stream is set, `fgetwc()` returns `WEOF` whether or not the stream is at end-of-file.

If a read error occurs, the error indicator for the stream is set, `fgetwc()` returns `WEOF` and sets `errno` to indicate the error.

If an encoding error occurs, the error indicator for the stream is set, `fgetwc()` returns `WEOF`, and `errno` is set to indicate the error.

Errors The `fgetwc()` function will fail if data needs to be read and:

`EAGAIN` The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the `fgetwc()` operation.

`EBADF` The file descriptor underlying *stream* is not a valid file descriptor open for reading.

`EINTR` The read operation was terminated due to the receipt of a signal, and no data was transferred.

`EIO` A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal and either the process is ignoring or blocking the `SIGTTIN` signal or the process group is orphaned.

`EOVERFLOW` The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding *stream*.

The `fgetwc()` function may fail if:

`ENOMEM` Insufficient storage space is available.

`ENXIO` A request was made of a non-existent device, or the request was outside the capabilities of the device.

`EILSEQ` The data obtained from the input stream does not form a valid character.

Usage The `ferror`(3C) or `feof`(3C) functions must be used to distinguish between an error condition and an end-of-file condition.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards`(5).

See Also `feof`(3C), `ferror`(3C), `fgetc`(3C), `fgets`(3C), `fgetws`(3C), `fopen`(3C), `fread`(3C), `fscanf`(3C), `getc`(3C), `getchar`(3C), `gets`(3C), `scanf`(3C), `setlocale`(3C), `ungetc`(3C), `ungetwc`(3C), `attributes`(5), `standards`(5)

Name floating_to_decimal, single_to_decimal, double_to_decimal, extended_to_decimal, quadruple_to_decimal – convert floating-point value to decimal record

Synopsis

```
#include <floatingpoint.h>

void single_to_decimal(single *px, decimal_mode *pm,
     decimal_record *pd, fp_exception_field_type *ps);

void double_to_decimal(double *px, decimal_mode *pm,
     decimal_record *pd, fp_exception_field_type *ps);

void extended_to_decimal(extended *px, decimal_mode *pm,
     decimal_record *pd, fp_exception_field_type *ps);

void quadruple_to_decimal(quadruple *px, decimal_mode *pm,
     decimal_record *pd, fp_exception_field_type *ps);
```

Description The `floating_to_decimal` functions convert the floating-point value at **px* into a decimal record at **pd*, observing the modes specified in **pm* and setting exceptions in **ps*. If there are no IEEE exceptions, **ps* will be zero.

If **px* is zero, infinity, or NaN, then only *pd→sign* and *pd→fpclass* are set. Otherwise *pd→exponent* and *pd→ds* are also set so that

```
(sig)*(pd->ds)*10**(pd->exponent)
```

is a correctly rounded approximation to **px*, where *sig* is +1 or −1, depending upon whether *pd→sign* is 0 or −1. *pd→ds* has at least one and no more than `DECIMAL_STRING_LENGTH`−1 significant digits because one character is used to terminate the string with a null.

pd→ds is correctly rounded according to the IEEE rounding modes in *pm→rd*. **ps* has *fp_inexact* set if the result was inexact, and has *fp_overflow* set if the string result does not fit in *pd→ds* because of the limitation `DECIMAL_STRING_LENGTH`.

If *pm→df* == *floating_form*, then *pd→ds* always contains *pm→ndigits* significant digits. Thus if **px* == 12.34 and *pm→ndigits* == 8, then *pd→ds* will contain 12340000 and *pd→exponent* will contain −6.

If *pm→df* == *fixed_form* and *pm→ndigits* >= 0, then the decimal value is rounded at *pm→ndigits* digits to the right of the decimal point. For example, if **px* == 12.34 and *pm→ndigits* == 1, then *pd→ds* will contain 123 and *pd→exponent* will be set to −1.

If *pm→df* == *fixed_form* and *pm→ndigits*< 0, then the decimal value is rounded at −*pm→ndigits* digits to the left of the decimal point, and *pd→ds* is padded with trailing zeros up to the decimal point. For example, if **px* == 12.34 and *pm→n* digits == −1, then *pd→ds* will contain 10 and *pd→exponent* will be set to 0.

When *pm→df* == *fixed_form* and the value to be converted is large enough that the resulting string would contain more than `DECIMAL_STRING_LENGTH`−1 digits, then the string placed in

pd→ds is limited to exactly `DECIMAL_STRING_LENGTH`-1 digits (by moving the place at which the value is rounded further left if need be), *pd→exponent* is adjusted accordingly and the overflow flag is set in **ps*.

pd->more is not used.

The `econvert`(3C), `fconvert`(3C), `gconvert`(3C), `printf`(3C), and `sprintf`(3C) functions all use `double_to_decimal()`.

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also

`econvert`(3C), `fconvert`(3C), `gconvert`(3C), `printf`(3C), `sprintf`(3C), `attributes`(5)

Name flockfile, funlockfile, ftrylockfile – acquire and release stream lock

Synopsis

```
#include <stdio.h>

void flockfile(FILE *stream);

void funlockfile(FILE *stream);

int ftrylockfile(FILE *stream);
```

Description The `flockfile()` function acquires an internal lock of a stream *stream*. If the lock is already acquired by another thread, the thread calling `flockfile()` is suspended until it can acquire the lock. In the case that the stream lock is available, `flockfile()` not only acquires the lock, but keeps track of the number of times it is being called by the current thread. This implies that the stream lock can be acquired more than once by the same thread.

The `funlockfile()` function releases the lock being held by the current thread. In the case of recursive locking, this function must be called the same number of times `flockfile()` was called. After the number of `funlockfile()` calls is equal to the number of `flockfile()` calls, the stream lock is available for other threads to acquire.

The `ftrylockfile()` function acquires an internal lock of a stream *stream*, only if that object is available. In essence `ftrylockfile()` is a non-blocking version of `flockfile()`.

Return Values The `ftrylockfile()` function returns `0` on success and non-zero to indicate a lock cannot be acquired.

Examples **EXAMPLE 1** A sample program of `flockfile()`.

The following example prints everything out together, blocking other threads that might want to write to the same file between calls to `fprintf(3C)`:

```
FILE iop;
flockfile(iop);
fprintf(iop, "hello ");
fprintf(iop, "world);
fputc(iop, 'a');
funlockfile(iop);
```

An unlocked interface is available in case performance is an issue. For example:

```
flockfile(iop);
while (!feof(iop)) {
        *c++ = getc_unlocked(iop);
}
funlockfile(iop);
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `Intro(3)`, `__fsetlocking(3C)`, `ferror(3C)`, `fprintf(3C)`, `getc(3C)`, `putc(3C)`, `stdio(3C)`, `ungetc(3C)`, `attributes(5)`, `standards(5)`

Notes The interfaces on this page are as specified in IEEE Std 1003.1:2001. See `standards(5)`.

Name fmtmsg – display a message on stderr or system console

Synopsis

```
#include <fmtmsg.h>

int fmtmsg(long classification, const  char *label, int severity,
const char *text, const char *action, const char *tag);
```

Description The `fmtmsg()` function writes a formatted message to `stderr`, to the console, or to both, on a message's classification component. It can be used instead of the traditional `printf(3C)` interface to display messages to `stderr`, and in conjunction with `gettxt(3C)`, provides a simple interface for producing language-independent applications.

A formatted message consists of up to five standard components (*label*, *severity*, *text*, *action*, and *tag*) as described below. The *classification* component is not part of the standard message displayed to the user, but rather defines the source of the message and directs the display of the formatted message.

classification Contains identifiers from the following groups of major classifications and subclassifications. Any one identifier from a subclass may be used in combination by ORing the values together with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both `stderr` and the system console).

- "Major classifications" identify the source of the condition. Identifiers are: `MM_HARD` (hardware), `MM_SOFT` (software), and `MM_FIRM` (firmware).
- "Message source subclassifications" identify the type of software in which the problem is spotted. Identifiers are: `MM_APPL` (application), `MM_UTIL` (utility), and `MM_OPSYS` (operating system).
- "Display subclassifications" indicate where the message is to be displayed. Identifiers are: `MM_PRINT` to display the message on the standard error stream, `MM_CONSOLE` to display the message on the system console. Neither, either, or both identifiers may be used.
- "Status subclassifications" indicate whether the application will recover from the condition. Identifiers are: `MM_RECOVER` (recoverable) and `MM_NRECOV` (non-recoverable).
- An additional identifier, `MM_NULLMC`, indicates that no classification component is supplied for the message.

label Identifies the source of the message. The format of this component is two fields separated by a colon. The first field is up to 10 characters long; the second is up to 14 characters. Suggested usage is that *label* identifies the package in which the application resides as well as the program or application name. For example, the *label* `UX:cat` indicates the UNIX System V package and the `cat(1)` utility.

severity Indicates the seriousness of the condition. Identifiers for the standard levels of *severity* are:

- `MM_HALT` indicates that the application has encountered a severe fault and is halting. Produces the print string `HALT`.
- `MM_ERROR` indicates that the application has detected a fault. Produces the print string `ERROR`.
- `MM_WARNING` indicates a condition out of the ordinary that might be a problem and should be watched. Produces the print string `WARNING`.
- `MM_INFO` provides information about a condition that is not in error. Produces the print string `INFO`.
- `MM_NOSEV` indicates that no severity level is supplied for the message.

Other severity levels may be added by using the `addseverity()` routine.

text Describes the condition that produced the message. The *text* string is not limited to a specific size.

action Describes the first step to be taken in the error recovery process. `fmtmsg()` precedes each action string with the prefix: `TOFIX:`. The *action* string is not limited to a specific size.

tag An identifier which references on-line documentation for the message. Suggested usage is that *tag* includes the *label* and a unique identifying number. A sample *tag* is `UX:cat:146`.

Environment Variables

The `MSGVERB` and `SEV_LEVEL` environment variables control the behavior of `fmtmsg()` as follows:

`MSGVERB` This variable determines which message components `fmtmsg()` selects when writing messages to `stderr`. Its value is a colon-separated list of optional keywords and can be set as follows:

```
MSGVERB=[keyword[:keyword[: . . .]]]
export MSGVERB
```

Valid *keywords* are: `label`, `severity`, `text`, `action`, and `tag`. If `MSGVERB` contains a keyword for a component and the component's value is not the component's null value, `fmtmsg()` includes that component in the message when writing the message to `stderr`. If `MSGVERB` does not include a keyword for a message component, that component is not included in the display of the message. The keywords may appear in any order. If `MSGVERB` is not defined, if its value is the null string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed above, `fmtmsg()` selects all components.

The first time `fmtmsg()` is called, it examines `MSGVERB` to determine which message components are to be selected when generating a message to write to the standard error stream, `stderr`. The values accepted on the initial call are saved for future calls.

The `MSGVERB` environment variable affects only those components that are selected for display to the standard error stream. All message components are included in console messages.

`SEV_LEVEL` This variable defines severity levels and associates print strings with them for use by `fmtmsg()`. The standard severity levels listed below cannot be modified. Additional severity levels can also be defined, redefined, and removed using `addseverity()` (see `addseverity(3C)`). If the same severity level is defined by both `SEV_LEVEL` and `addseverity()`, the definition by `addseverity()` takes precedence.

0 (no severity is used)

1 `HALT`

2 `ERROR`

3 `WARNING`

4 `INFO`

The `SEV_LEVEL` variable can be set as follows:

```
SEV_LEVEL=[description[:description[: . . .]]]
export SEV_LEVEL
```

where *description* is a comma-separated list containing three fields:

description=severity_keyword,level,printstring

The *severity_keyword* field is a character string that is used as the keyword on the `-s` *severity* option to the `fmtmsg(1)` utility. (This field is not used by the `fmtmsg()` function.)

The *level* field is a character string that evaluates to a positive integer (other than 0, 1, 2, 3, or 4, which are reserved for the standard severity levels). If the keyword *severity_keyword* is used, *level* is the severity value passed on to the `fmtmsg()` function.

The *printstring* field is the character string used by `fmtmsg()` in the standard message format whenever the severity value *level* is used.

If a *description* in the colon list is not a three-field comma list, or if the second field of a comma list does not evaluate to a positive integer, that *description* in the colon list is ignored.

The first time `fmtmsg()` is called, it examines the `SEV_LEVEL` environment variable, if defined, to determine whether the environment expands the levels of severity beyond the five standard levels and those defined using `addseverity()`. The values accepted on the initial call are saved for future calls.

Use in Applications

One or more message components may be systematically omitted from messages generated by an application by using the null value of the argument for that component.

The table below indicates the null values and identifiers for `fmtmsg()` arguments.

Argument	Type	Null-Value	Identifier
label	`char*`	`(char*) NULL`	`MM_NULLLBL`
severity	`int`	`0`	`MM_NULLSEV`
class	`long`	`0L`	`MM_NULLMC`
text	`char*`	`(char*) NULL`	`MM_NULLTXT`
action	`char*`	`(char*) NULL`	`MM_NULLACT`
tag	`char*`	`(char*) NULL`	`MM_NULLTAG`

Another means of systematically omitting a component is by omitting the component keyword(s) when defining the `MSGVERB` environment variable (see the `Environment Variables` section above).

Return Values

The `fmtmsg()` returns the following values:

`MM_OK` The function succeeded.

`MM_NOTOK` The function failed completely.

`MM_NOMSG` The function was unable to generate a message on the standard error stream, but otherwise succeeded.

`MM_NOCON` The function was unable to generate a console message, but otherwise succeeded.

Examples

EXAMPLE 1 The following example of `fmtmsg()`:

```
fmtmsg(MM_PRINT, "UX:cat", MM_ERROR, "invalid syntax",
"refer to manual", "UX:cat:001")
```

EXAMPLE 1 The following example of `fmtmsg()`: *(Continued)*

produces a complete message in the standard message format:

```
UX:cat: ERROR: invalid syntax
TO FIX: refer to manual   UX:cat:001
```

EXAMPLE 2 When the environment variable `MSGVERB` is set as follows:

```
MSGVERB=severity:text:action
```

and the Example 1 is used, `fmtmsg()` produces:

```
ERROR: invalid syntax
TO FIX: refer to manual
```

EXAMPLE 3 When the environment variable `SEV_LEVEL` is set as follows:

```
SEV_LEVEL=note,5,NOTE
```

the following call to `fmtmsg()`

```
fmtmsg(MM_UTIL | MM_PRINT, "UX:cat", 5, "invalid syntax",
"refer to manual", "UX:cat:001")
```

produces

```
UX:cat: NOTE: invalid syntax
TO FIX: refer to manual   UX:cat:001
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also

`fmtmsg(1)`, `addseverity(3C)`, `gettxt(3C)`, `printf(3C)`, `attributes(5)`, `standards(5)`

Name fnmatch – match filename or path name

Synopsis

```
#include <fnmatch.h>

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

Description The `fnmatch()` function matches patterns as described on the `fnmatch(5)` manual page. It checks the *string* argument to see if it matches the *pattern* argument.

The *flags* argument modifies the interpretation of *pattern* and *string*. It is the bitwise inclusive OR of zero or more of the following flags defined in the header <`fnmatch.h`>.

`FNM_PATHNAME` If set, a slash (`/`) character in *string* will be explicitly matched by a slash in *pattern*; it will not be matched by either the asterisk (`*`) or question-mark (`?`) special characters, nor by a bracket (`[ ]`) expression.

If not set, the slash character is treated as an ordinary character.

`FNM_NOESCAPE` If not set, a backslash character (`\`) in *pattern* followed by any other character will match that second character in *string*. In particular, "`\\`" will match a backslash in *string*.

If set, a backslash character will be treated as an ordinary character.

`FNM_PERIOD` If set, a leading period in *string* will match a period in *pattern*; where the location of "leading" is indicated by the value of `FNM_PATHNAME`:

- If `FNM_PATHNAME` is set, a period is "leading" if it is the first character in *string* or if it immediately follows a slash.
- If `FNM_PATHNAME` is not set, a period is "leading" only if it is the first character of *string*.

If not set, no special restrictions are placed on matching a period.

Return Values If *string* matches the pattern specified by *pattern*, then `fnmatch()` returns `0`. If there is no match, `fnmatch()` returns `FNM_NOMATCH`, which is defined in the header <`fnmatch.h`>. If an error occurs, `fnmatch()` returns another non-zero value.

Usage The `fnmatch()` function has two major uses. It could be used by an application or utility that needs to read a directory and apply a pattern against each entry. The `find(1)` utility is an example of this. It can also be used by the `pax(1)` utility to process its *pattern* operands, or by applications that need to match strings in a similar manner.

The name `fnmatch()` is intended to imply *filename* match, rather than *pathname* match. The default action of this function is to match filenames, rather than path names, since it gives no special significance to the slash character. With the `FNM_PATHNAME` flag, `fnmatch()` does match path names, but without tilde expansion, parameter expansion, or special treatment for period at the beginning of a filename.

The `fnmatch()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also

`find(1)`, `pax(1)`, `glob(3C)`, `setlocale(3C)`, `wordexp(3C)`, `attributes(5)`, `fnmatch(5)`, `standards(5)`

Name fopen – open a stream

Synopsis

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
```

Description The `fopen()` function opens the file whose pathname is the string pointed to by *filename*, and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences:

`r` or `rb`	Open file for reading.
`w` or `wb`	Truncate to zero length or create file for writing.
`a` or `ab`	Append; open or create file for writing at end-of-file.
`r+` or `rb+` or `r+b`	Open file for update (reading and writing).
`w+` or `wb+` or `w+b`	Truncate to zero length or create file for update.
`a+` or `ab+` or `a+b`	Append; open or create file for update, writing at end-of-file.

The character `b` has no effect, but is allowed for ISO C standard conformance (see `standards(5)`). Opening a file with read mode (`r` as the first character in the *mode* argument) fails if the file does not exist or cannot be read.

Opening a file with append mode (`a` as the first character in the *mode* argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to `fseek(3C)`. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

When a file is opened with update mode (`+` as the second or third character in the *mode* argument), both input and output may be performed on the associated stream. However, output must not be directly followed by input without an intervening call to `fflush(3C)` or to a file positioning function (`fseek(3C)`, `fsetpos(3C)` or `rewind(3C)`), and input must not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

If *mode* begins with *w* or `a` and the file did not previously exist, upon successful completion, `fopen()` function will mark for update the `st_atime`, `st_ctime` and `st_mtime` fields of the file and the `st_ctime` and `st_mtime` fields of the parent directory.

If *mode* begins with *w* and the file did previously exist, upon successful completion, `fopen()` will mark for update the `st_ctime` and `st_mtime` fields of the file. The `fopen()` function will allocate a file descriptor as `open(2)` does.

Normally, 32-bit applications return an `EMFILE` error when attempting to associate a stream with a file accessed by a file descriptor with a value greater than 255. If the last character of *mode* is `F`, 32-bit applications will be allowed to associate a stream with a file accessed by a file descriptor with a value greater than 255. A `FILE` pointer obtained in this way must never be used by any code that might directly access fields in the `FILE` structure. If the fields in the `FILE` structure are used directly by 32-bit applications when the last character of mode is `F`, data corruption could occur. See the USAGE section of this manual page and the `enable_extended_FILE_stdio(3C)` manual page for other options for enabling the extended FILE facility.

In 64-bit applications, the last character of *mode* is silently ignored if it is `F`. 64-bit applications are always allowed to associate a stream with a file accessed by a file descriptor with any value.

The largest value that can be represented correctly in an object of type `off_t` will be established as the offset maximum in the open file description.

Return Values Upon successful completion, `fopen()` returns a pointer to the object controlling the stream. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

The `fopen()` function may fail and not set `errno` if there are no free `stdio` streams.

Errors The `fopen()` function will fail if:

`EACCES`	Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *mode* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
`EINTR`	A signal was caught during the execution of `fopen()`.
`EISDIR`	The named file is a directory and *mode* requires write access.
`ELOOP`	Too many symbolic links were encountered in resolving *path*.
`EMFILE`	There are {`OPEN_MAX`} file descriptors currently open in the calling process.
`ENAMETOOLONG`	The length of the *filename* exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.
`ENFILE`	The maximum allowable number of files is currently open in the system.
`ENOENT`	A component of *filename* does not name an existing file or *filename* is an empty string.

ENOSPC	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The named file is a character special or block special file, and the device associated with this special file does not exist.
EOVERFLOW	The current value of the file position cannot be represented correctly in an object of type fpos_t.
EROFS	The named file resides on a read-only file system and *mode* requires write access.

The fopen() function may fail if:

EINVAL	The value of the *mode* argument is not valid.
EMFILE	{FOPEN_MAX} streams are currently open in the calling process.
	{STREAM_MAX} streams are currently open in the calling process.
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
ENOMEM	Insufficient storage space is available.
ETXTBSY	The file is a pure procedure (shared text) file that is being executed and *mode* requires write access.

Usage A process is allowed to have at least {FOPEN_MAX} stdio streams open at a time. For 32-bit applications, however, the underlying ABIs formerly required that no file descriptor used to access the file underlying a stdio stream have a value greater than 255. To maintain binary compatibility with earlier Solaris releases, this limit still constrains 32-bit applications. However, when a 32-bit application is aware that no code that has access to the FILE pointer returned by fopen() will use the FILE pointer to directly access any fields in the FILE structure, the F character can be used as the last character in the *mode* argument to circumvent this limit. Because it could lead to data corruption, the F character in *mode* must never be used when the FILE pointer might later be used by binary code unknown to the user. The F character in *mode* is intended to be used by library functions that need a FILE pointer to access data to process a user request, but do not need to pass the FILE pointer back to the user. 32-bit applications that have been inspected can use the extended FILE facility to circumvent this limit if the inspection shows that no FILE pointers will be used to directly access FILE structure contents.

The fopen() function has a transitional interface for 64-bit file offsets. See lf64(5).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See below.

For all aspects of this function except the `F` character in the *mode* argument, see `standards(5)`

See Also `enable_extended_FILE_stdio(3C)`, `fclose(3C)`, `fdopen(3C)`, `fflush(3C)`, `freopen(3C)`, `fsetpos(3C)`, `rewind(3C)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Name fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky – IEEE floating-point environment control

Synopsis

```
#include <ieeefp.h>

fp_rnd fpgetround(void);

fp_rnd fpsetround(fp_rnd rnd_dir);

fp_except fpgetmask(void);

fp_except fpsetmask(fp_except mask);

fp_except fpgetsticky(void);

fp_except fpsetsticky(fp_except sticky);
```

Description There are five floating-point exceptions:

- divide-by-zero,
- overflow,
- underflow,
- imprecise (inexact) result, and
- invalid operation.

When a floating-point exception occurs, the corresponding sticky bit is set (1), and if the mask bit is enabled (1), the trap takes place. These routines let the user change the behavior on occurrence of any of these exceptions, as well as change the rounding mode for floating-point operations.

The *mask* argument is formed by the logical OR operation of the following floating-point exception masks:

```
FP_X_INV       /* invalid operation exception */
FP_X_OFL       /* overflow exception */
FP_X_UFL       /* underflow exception */
FP_X_DZ        /* divide-by-zero exception */
FP_X_IMP       /* imprecise (loss of precision) */
```

The following floating-point rounding modes are passed to `fpsetround` and returned by `fpgetround()`.

```
FP_RN       /* round to nearest representative number */
FP_RP       /* round to plus infinity */
FP_RM       /* round to minus infinity */
FP_RZ       /* round to zero (truncate) */
```

The default environment is rounding mode set to nearest (`FP_RN`) and all traps disabled.

The `fpsetsticky()` function modifies all sticky flags. The `fpsetmask()` function changes all mask bits. The `fpsetmask()` function clears the sticky bit corresponding to any exception being enabled.

Return Values The `fpgetround()` function returns the current rounding mode.

The `fpsetround()` function sets the rounding mode and returns the previous rounding mode.

The `fpgetmask()` function returns the current exception masks.

The `fpsetmask()` function sets the exception masks and returns the previous setting.

The `fpgetsticky()` function returns the current exception sticky flags.

The `fpsetsticky()` function sets (clears) the exception sticky flags and returns the previous setting.

Usage The C programming language requires truncation (round to zero) for floating point to integral conversions. The current rounding mode has no effect on these conversions.

The sticky bit must be cleared to recover from the trap and proceed. If the sticky bit is not cleared before the next trap occurs, a wrong exception type may be signaled.

Individual bits may be examined using the constants defined in `<ieeefp.h>`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `isnand(3C)`, `attributes(5)`

Name fputc, putc, putc_unlocked, putchar, putchar_unlocked, putw – put a byte on a stream

Synopsis

```
#include <stdio.h>

int fputc(int c, FILE *stream);

int putc(int c, FILE *stream);

int putc_unlocked(int c, FILE *stream);

int putchar(int c);

int putchar_unlocked(int c);

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

Description The `fputc()` function writes the byte specified by *c* (converted to an `unsigned char`) to the output stream pointed to by *stream*, at the position indicated by the associated file-position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the byte is appended to the output stream.

The `st_ctime` and `st_mtime` fields of the file will be marked for update between the successful execution of `fputc()` and the next successful completion of a call to `fflush`(3C) or `fclose`(3C) on the same stream or a call to `exit`(3C) or `abort`(3C).

The `putc()` routine behaves like `fputc()`, except that it is implemented as a macro. It runs faster than `fputc()`, but it takes up more space per invocation and its name cannot be passed as an argument to a function call.

The call `putchar(`*c*`)` is equivalent to `putc(`*c*`, `*stdout*`)`. The `putchar()` routine is implemented as a macro.

The `putc_unlocked()` and `putchar_unlocked()` routines are variants of `putc()` and `putchar()`, respectively, that do not lock the stream. It is the caller's responsibility to acquire the stream lock before calling these routines and releasing the lock afterwards; see `flockfile`(3C) and `stdio`(3C). These routines are implemented as macros.

The `putw()` function writes the word (that is, type `int`) *w* to the output *stream* (at the position at which the file offset, if defined, is pointing). The size of a word is the size of a type `int` and varies from machine to machine. The `putw()` function neither assumes nor causes special alignment in the file.

The `st_ctime` and `st_mtime` fields of the file will be marked for update between the successful execution of `putw()` and the next successful completion of a call to `fflush`(3C) or `fclose`(3C) on the same stream or a call to `exit`(3C) or `abort`(3C).

Return Values Upon successful completion, `fputc()`, `putc()`, `putc_unlocked()`, `putchar()`, and `putchar_unlocked()` return the value that was written. Otherwise, these functions return EOF, the error indicator for the stream is set, and `errno` is set to indicate the error.

Upon successful completion, `putw()` returns `0`. Otherwise, it returns a non-zero value, sets the error indicator for the associated *stream*, and sets `errno` to indicate the error.

An unsuccessful completion will occur, for example, if the file associated with *stream* is not open for writing or if the output file cannot grow.

Errors The `fputc()`, `putc()`, `putc_unlocked()`, `putchar()`, `putchar_unlocked()`, and `putw()` functions will fail if either the *stream* is unbuffered or the *stream*'s buffer needs to be flushed, and:

`EAGAIN` The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

`EBADF` The file descriptor underlying *stream* is not a valid file descriptor open for writing.

`EFBIG` An attempt was made to write to a file that exceeds the maximum file size or the process' file size limit.

`EFBIG` The file is a regular file and an attempt was made to write at or beyond the offset maximum.

`EINTR` The write operation was terminated due to the receipt of a signal, and no data was transferred.

`EIO` A physical I/O error has occurred, or the process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU` and the process group of the process is orphaned. This error may also be returned under implementation-dependent conditions.

`ENOSPC` There was no free space remaining on the device containing the file.

`EPIPE` An attempt is made to write to a pipe or `FIFO` that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the calling thread.

The `fputc()`, `putc()`, `putc_unlocked()`, `putchar()`, `putchar_unlocked()`, and `putw()` functions may fail if:

`ENOMEM` Insufficient storage space is available.

`ENXIO` A request was made of a non-existent device, or the request was outside the capabilities of the device.

Usage Functions exist for the `putc()`, `putc_unlocked()`, `putchar()`, and `putchar_unlocked()` macros. To get the function form, the macro name must be undefined (for example, `#undef putc`).

When the macro forms are used, `putc()` and `putc_unlocked()` evaluate the *stream* argument more than once. In particular, `putc(c, *f++);` does not work sensibly. The `fputc()` function should be used instead when evaluating the *stream* argument has side effects.

Because of possible differences in word length and byte ordering, files written using `putw()` are implementation-dependent, and possibly cannot be read using `getw(3C)` by a different application or by the same application running in a different environment.

The `putw()` function is inherently byte stream oriented and is not tenable in the context of either multibyte character streams or wide-character streams. Application programmers are encouraged to use one of the character-based output functions instead.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	`fputc()`, `putc()`, `putc_unlocked()`, `putchar()`, and `putchar_unlocked()` are Standard.
MT-Level	See `NOTES` below.

See Also `getrlimit(2)`, `ulimit(2)` `write(2)`, `Intro(3)`, `abort(3C)`, `exit(3C)`, `fclose(3C)`, `ferror(3C)`, `fflush(3C)`, `flockfile(3C)`, `printf(3C)`, `putc(3C)`, `puts(3C)`, `setbuf(3C)`, `stdio(3C)`, `attributes(5)`, `standards(5)`

Notes The `fputc()`, `putc()`, `putchar()`, and `putw()` routines are MT-Safe in multithreaded applications. The `putc_unlocked()` and `putchar_unlocked()` routines are unsafe in multithreaded applications.

Name fputwc, putwc, putwchar – put wide-character code on a stream

Synopsis

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

wint_t fputwc(wchar_t wc, FILE*stream);

wint_t putwc(wchar_t wc, FILE*stream);

#include <wchar.h>

wint_t putwchar(wchar_t wc);
```

Description The `fputwc()` function writes the character corresponding to the wide-character code *wc* to the output stream pointed to by *stream*, at the position indicated by the associated file-position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream. If an error occurs while writing the character, the shift state of the output file is left in an undefined state.

The `st_ctime` and `st_mtime` fields of the file will be marked for update between the successful execution of `fputwc()` and the next successful completion of a call to `fflush`(3C) or `fclose`(3C) on the same stream or a call to `exit`(2) or `abort`(3C).

The `putwc()` function is equivalent to `fputwc()`, except that it is implemented as a macro.

The call `putwchar(`*wc*`)` is equivalent to `putwc(`*wc,* `stdout)`. The `putwchar()` routine is implemented as a macro.

Return Values Upon successful completion, `fputwc()`, `putwc()`, and `putwchar()` return *wc*. Otherwise, they return `WEOF`, the error indicator for the stream is set, and `errno` is set to indicate the error.

Errors The `fputwc()`, `putwc()`, and `putwchar()` functions will fail if either the stream is unbuffered or data in the *stream*'s buffer needs to be written, and:

`EAGAIN` The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

`EBADF` The file descriptor underlying *stream* is not a valid file descriptor open for writing.

`EFBIG` An attempt was made to write to a file that exceeds the maximum file size or the process's file size limit; or the file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

`EINTR` The write operation was terminated due to the receipt of a signal, and no data was transferred.

`EIO` A physical I/O error has occurred, or the process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU`, and the process group of the process is orphaned.

`ENOSPC` There was no free space remaining on the device containing the file.

`EPIPE` An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the calling thread.

The `fputwc()`, `putwc()`, and `putwchar()` functions may fail if:

`ENOMEM` Insufficient storage space is available.

`ENXIO` A request was made of a non-existent device, or the request was outside the capabilities of the device.

`EILSEQ` The wide-character code *wc* does not correspond to a valid character.

Usage Functions exist for the `putwc()` and `putwchar()` macros. To get the function form, the macro name must be undefined (for example, `#undef putc`).

When the macro form is used, `putwc()` evaluates the *stream* argument more than once. In particular, `putwc(`*wc*`, *`*f`*`++)` does not work sensibly. The `fputwc()` function should be used instead when evaluating the *stream* argument has side effects.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `exit(2)`, `ulimit(2)`, `abort(3C)`, `fclose(3C)`, `ferror(3C)`, `fflush(3C)`, `fopen(3C)`, `setbuf(3C)`, `attributes(5)`, `standards(5)`

Name fputws – put wide character string on a stream

Synopsis

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

int fputws(const wchar_t *restrict s, FILE *restrict stream);
```

Description The `fputws()` function writes a character string corresponding to the (null-terminated) wide character string pointed to by *ws* to the stream pointed to by *stream*. No character corresponding to the terminating null wide-character code is written, nor is a NEWLINE character appended.

The `st_ctime` and `st_mtime` fields of the file will be marked for update between the successful execution of `fputws()` and the next successful completion of a call to `fflush`(3C) or `fclose`(3C) on the same stream or a call to `exit`(2) or `abort`(3C).

Return Values Upon successful completion, `fputws()` returns a non-negative value. Otherwise, it returns −1, sets an error indicator for the stream, and sets `errno` to indicate the error.

Errors Refer to `fputwc`(3C).

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards`(5).

See Also `exit`(2), `abort`(3C), `fclose`(3C), `fflush`(3C), `fopen`(3C), `fputwc`(3C), `attributes`(5), `standards`(5)

Name fread – binary input

Synopsis

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

Description The `fread()` function reads into the array pointed to by *ptr* up to *nitems* elements whose size is specified by *size* in bytes, from the stream pointed to by *stream*. For each object, *size* calls are made to the `fgetc(3C)` function and the results stored, in the order read, in an array of `unsigned char` exactly overlaying the object. The file-position indicator for the stream (if defined) is advanced by the number of bytes successfully read. If an error occurs, the resulting value of the file-position indicator for the stream is unspecified. If a partial element is read, its value is unspecified.

The `fread()` function may mark the `st_atime` field of the file associated with *stream* for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc(3C)`, `fgets(3C)`, `fgetwc(3C)`, `fgetws(3C)`, `fread()`, `fscanf(3C)`, `getc(3C)`, `getchar(3C)`, `getdelim(3C)`, `getline(3C)`, `gets(3C)`, or `scanf(3C)` using *stream* that returns data not supplied by a prior call to `ungetc(3C)` or `ungetwc(3C)`.

Return Values Upon successful completion, `fread()` returns the number of elements successfully read, which is less than *nitems* only if a read error or end-of-file is encountered. If *size* or *nitems* is 0, `fread()` returns 0 and the contents of the array and the state of the stream remain unchanged. Otherwise, if a read error occurs, the error indicator for the stream is set and `errno` is set to indicate the error.

Errors Refer to `fgetc(3C)`.

Examples **EXAMPLE 1** Reading from a Stream

The following example reads a single element from the *fp* stream into the array pointed to by *buf*.

```
#include <stdio.h>
...
size_t bytes_read;
char buf[100];
FILE *fp;
...
bytes_read = fread(buf, sizeof(buf), 1, fp);
...
```

Usage The `ferror()` or `feof()` functions must be used to distinguish between an error condition and end-of-file condition. See `ferror(3C)`.

Because of possible differences in element length and byte ordering, files written using `fwrite(3C)` are application-dependent, and possibly cannot be read using `fread()` by a different application or by the same application on a different processor.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also read(2), fclose(3C), ferror(3C), fopen(3C), getc(3C), getdelim(3C), getline(3C), gets(3C), printf(3C), putc(3C), puts(3C), attributes(5), standards(5)

Name freopen – open a stream

Synopsis

```
#include <stdio.h>

FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

Description The `freopen()` function first attempts to flush the stream and close any file descriptor associated with *stream*. Failure to flush or close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.

The `freopen()` function opens the file whose pathname is the string pointed to by *filename* and associates the stream pointed to by *stream* with it. The *mode* argument is used just as in `fopen(3C)`.

If *filename* is a null pointer and the application comforms to SUSv3 (see `standards(5)`), the `freopen()` function attempts to change the mode of the stream to that specified by *mode*, as though the name of the file currently associated with the *stream* had been used. The following changes of mode are permitted, depending upon the access mode of the file descriptor underlying the stream:

- When `+` is specified, the file descriptor mode must be `O_RDWR`.
- When `r` is specified, the file descriptor mode must be `O_RDONLY` or `O_RDWR`.
- When `a` or `w` is specified, the file descriptor mode must be `O_WRONLY` or `O_RDWR`.

If the filename is a null pointer and the application does not comform to SUSv3, `freopen()` returns a null pointer.

The original stream is closed regardless of whether the subsequent open succeeds.

After a successful call to the `freopen()` function, the orientation of the stream is cleared, the encoding rule is cleared, and the associated `mbstate_t` object is set to describe an initial conversion state.

The largest value that can be represented correctly in an object of type `off_t` will be established as the offset maximum in the open file description.

Return Values Upon successful completion, `freopen()` returns the value of *stream*. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

Errors The `freopen()` function will fail if:

`EACCES` Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *mode* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

`EBADF` The application comforms to SUSv3, the *filename* argument is a null pointer, and either the underlying file descriptor is not valid or the mode specified when the underlying file descriptor was opened does not support the file access modes requested by the *mode* argument.

EFAULT	The application does not comform to SUSv3 and the *filename* argument is a null pointer.
EINTR	A signal was caught during `freopen()`.
EISDIR	The named file is a directory and *mode* requires write access.
ELOOP	Too many symbolic links were encountered in resolving *path*.
EMFILE	There are {`OPEN_MAX`} file descriptors currently open in the calling process.
ENAMETOOLONG	The length of the *filename* exceeds {`PATH_MAX`} or a pathname component is longer than {`NAME_MAX`}.
ENFILE	The maximum allowable number of files is currently open in the system.
ENOENT	A component of *filename* does not name an existing file or *filename* is an empty string.
ENOSPC	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The named file is a character special or block special file, and the device associated with this special file does not exist.
EOVERFLOW	The current value of the file position cannot be represented correctly in an object of type `off_t`.
EROFS	The named file resides on a read-only file system and *mode* requires write access.

The `freopen()` function may fail if:

EINVAL	The value of the *mode* argument is not valid.
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {`PATH_MAX`}.
ENOMEM	Insufficient storage space is available.
ENXIO	A request was made of a non-existent device, or the request was outside the capabilities of the device.
ETXTBSY	The file is a pure procedure (shared text) file that is being executed and *mode* requires write access.

Usage The `freopen()` function is typically used to attach the preopened *streams* associated with `stdin`, `stdout` and `stderr` to other files. By default `stderr` is unbuffered, but the use of `freopen()` will cause it to become buffered or line-buffered.

The `freopen()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also fclose(3C), fdopen(3C), fopen(3C), stdio(3C), attributes(5), lf64(5), standards(5)

Name fseek, fseeko – reposition a file-position indicator in a stream

Synopsis

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);

int fseeko(FILE *stream, off_t offset, int whence);
```

Description The `fseek()` function sets the file-position indicator for the stream pointed to by *stream*. The `fseeko()` function is identical to `fseek()` except for the type of *offset*.

The new position, measured in bytes from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*, whose values are defined in <`stdio.h`> as follows:

`SEEK_SET` Set position equal to *offset* bytes.

`SEEK_CUR` Set position to current location plus *offset*.

`SEEK_END` Set position to EOF plus *offset*.

If the stream is to be used with wide character input/output functions, *offset* must either be 0 or a value returned by an earlier call to `ftell(3C)` on the same stream and *whence* must be `SEEK_SET`.

A successful call to `fseek()` clears the end-of-file indicator for the stream and undoes any effects of `ungetc(3C)` and `ungetwc(3C)` on the same stream. After an `fseek()` call, the next operation on an update stream may be either input or output.

If the most recent operation, other than `ftell(3C)`, on a given stream is `fflush(3C)`, the file offset in the underlying open file description will be adjusted to reflect the location specified by `fseek()`.

The `fseek()` function allows the file-position indicator to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return bytes with the value 0 until data is actually written into the gap.

The value of the file offset returned by `fseek()` on devices which are incapable of seeking is undefined.

If the stream is writable and buffered data had not been written to the underlying file, `fseek()` will cause the unwritten data to be written to the file and mark the `st_ctime` and `st_mtime` fields of the file for update.

Return Values The `fseek()` and `fseeko()` functions return `0` on success; otherwise, they returned `−1` and set `errno` to indicate the error.

Errors The `fseek()` and `fseeko()` functions will fail if, either the *stream* is unbuffered or the *stream*'s buffer needed to be flushed, and the call to `fseek()` or `fseeko()` causes an underlying `lseek(2)` or `write(2)` to be invoked:

`EAGAIN` The `O_NONBLOCK` flag is set for the file descriptor and the process would be delayed in the write operation.

`EBADF` The file descriptor underlying the stream file is not open for writing or the stream's buffer needed to be flushed and the file is not open.

`EFBIG` An attempt was made to write a file that exceeds the maximum file size or the process's file size limit, or the file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

`EINTR` The write operation was terminated due to the receipt of a signal, and no data was transferred.

`EINVAL` The *whence* argument is invalid. The resulting file-position indicator would be set to a negative value.

`EIO` A physical I/O error has occurred; or the process is a member of a background process group attempting to perform a `write(2)` operation to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU`, and the process group of the process is orphaned.

`ENOSPC` There was no free space remaining on the device containing the file.

`ENXIO` A request was made of a non-existent device, or the request was outside the capabilities of the device.

`EPIPE` The file descriptor underlying *stream* is associated with a pipe or FIFO.

`EPIPE` An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the calling thread.

The `fseek()` function will fail if:

`EOVERFLOW` The resulting file offset would be a value which cannot be represented correctly in an object of type `long`.

The `fseeko()` function will fail if:

`EOVERFLOW` The resulting file offset would be a value which cannot be represented correctly in an object of type `off_t`.

Usage Although on the UNIX system an offset returned by `ftell()` or `ftello()` (see `ftell(3C)`) is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX systems requires that an offset be used by `fseek()` directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

The `fseeko()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `getrlimit(2)`, `ulimit(2)`, `ftell(3C)`, `rewind(3C)`, `ungetc(3C)`, `ungetwc(3C)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Name fsetpos – reposition a file pointer in a stream

Synopsis

```
#include <stdio.h>

int fsetpos(FILE *stream, const fpos_t *pos);
```

Description The `fsetpos()` function sets the file position indicator for the stream pointed to by *stream* according to the value of the object pointed to by *pos*, which must be a value obtained from an earlier call to `fgetpos(3C)` on the same stream.

A successful call to `fsetpos()` function clears the end-of-file indicator for the stream and undoes any effects of `ungetc(3C)` on the same stream. After an `fsetpos()` call, the next operation on an update stream may be either input or output.

Return Values The `fsetpos()` function returns `0` if it succeeds; otherwise it returns a non-zero value and sets `errno` to indicate the error.

Errors The `fsetpos()` function may fail if:

`EBADF` The file descriptor underlying *stream* is not valid.

`ESPIPE` The file descriptor underlying *stream* is associated with a pipe, a FIFO, or a socket.

Usage The `fsetpos()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `lseek(2)`, `fgetpos(3C)`, `fopen(3C)`, `fseek(3C)`, `ftell(3C)`, `rewind(3C)`, `ungetc(3C)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Name fsync – synchronize changes to a file

Synopsis

```
#include <unistd.h>

int fsync(int fildes);
```

Description The `fsync()` function moves all modified data and attributes of the file descriptor *fildes* to a storage device. When `fsync()` returns, all in-memory modified copies of buffers associated with *fildes* have been written to the physical medium. The `fsync()` function is different from `sync()`, which schedules disk I/O for all files but returns before the I/O completes. The `fsync()` function forces all outstanding data operations to synchronized file integrity completion (see `fcntl.h(3HEAD)` definition of `O_SYNC`.)

The `fsync()` function forces all currently queued I/O operations associated with the file indicated by the file descriptor *fildes* to the synchronized I/O completion state. All I/O operations are completed as defined for synchronized I/O file integrity completion.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error. If the `fsync()` function fails, outstanding I/O operations are not guaranteed to have been completed.

Errors The `fsync()` function will fail if:

`EBADF`	The *fildes* argument is not a valid file descriptor.
`EINTR`	A signal was caught during execution of the `fsync()` function.
`EIO`	An I/O error occurred while reading from or writing to the file system.
`ENOSPC`	There was no free space remaining on the device containing the file.
`ETIMEDOUT`	Remote connection timed out. This occurs when the file is on an NFS file system mounted with the *soft* option. See `mount_nfs(1M)`.

In the event that any of the queued I/O operations fail, `fsync()` returns the error conditions defined for `read(2)` and `write(2)`.

Usage The `fsync()` function should be used by applications that require that a file be in a known state. For example, an application that contains a simple transaction facility might use `fsync()` to ensure that all changes to a file or files caused by a given transaction were recorded on a storage medium.

The manner in which the data reach the physical medium depends on both implementation and hardware. The `fsync()` function returns when notified by the device driver that the write has taken place.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `mount_nfs(1M)`, `read(2)`, `sync(2)`, `write(2)`, `fcntl.h(3HEAD)`, `fdatasync(3C)`, `attributes(5)`, `standards(5)`

Name ftell, ftello – return a file offset in a stream

Synopsis

```
#include <stdio.h>

long ftell(FILE *stream);

off_t ftello(FILE *stream);
```

Description The `ftell()` function obtains the current value of the file-position indicator for the stream pointed to by *stream*. The `ftello()` function is identical to `ftell()` except for the return type.

Return Values Upon successful completion, the `ftell()` and `ftello()` functions return the current value of the file-position indicator for the stream measured in bytes from the beginning of the file. Otherwise, they return –1 and sets `errno` to indicate the error.

Errors The `ftell()` and `ftello()` functions will fail if:

`EBADF` The file descriptor underlying *stream* is not an open file descriptor.

`ESPIPE` The file descriptor underlying *stream* is associated with a pipe, a FIFO, or a socket.

The `ftell()` function will fail if:

`EOVERFLOW` The current file offset cannot be represented correctly in an object of type `long`.

The `ftello()` function will fail if:

`EOVERFLOW` The current file offset cannot be represented correctly in an object of type `off_t`.

Usage The `ftello()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `lseek(2)`, `fopen(3C)`, `fseek(3C)`, `attributes(5)`, , `standards(5)`, `lf64(5)`

Name ftime – get date and time

Synopsis

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

int ftime(struct timeb *tp);
```

Description The `ftime()` function sets the `time` and `millitm` members of the `timeb` structure pointed to by *tp*. The structure is defined in <`sys/timeb.h`> and contains the following members:

```
time_t           time;
unsigned short   millitm;
short            timezone;
short            dstflag;
```

The `time` and `millitm` members contain the seconds and milliseconds portions, respectively, of the current time in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970.

The `timezone` member contains the local time zone. The `dstflag` member contains a flag that, if non-zero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

The contents of the `timezone` and `dstflag` members of *tp* after a call to `ftime()` are unspecified.

Return Values Upon successful completion, the `ftime()` function returns `0`. Otherwise −`1` is returned.

Errors No errors are defined.

Usage For portability to implementations conforming to earlier versions of this document, `time(2)` is preferred over this function.

The millisecond value usually has a granularity greater than one due to the resolution of the system clock. Depending on any granularity (particularly a granularity of one) renders code non-portable.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `date(1)`, `time(2)`, `ctime(3C)`, `gettimeofday(3C)`, `timezone(4)`, `attributes(5)`, `standards(5)`

Name ftok – generate an IPC key

Synopsis

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

key_t ftok(const char *path, int id);
```

Description The `ftok()` function returns a key based on *path* and *id* that is usable in subsequent calls to `msgget(2)`, `semget(2)` and `shmget(2)`. The *path* argument must be the pathname of an existing file that the process is able to `stat(2)`.

The `ftok()` function will return the same key value for all paths that name the same file, when called with the same *id* value, and will return different key values when called with different *id* values.

If the file named by *path* is removed while still referred to by a key, a call to `ftok()` with the same *path* and *id* returns an error. If the same file is recreated, then a call to `ftok()` with the same *path* and *id* is likely to return a different key.

Only the low order 8-bits of `id` are significant. The behavior of `ftok()` is unspecified if these bits are 0.

Return Values Upon successful completion, `ftok()` returns a key. Otherwise, `ftok()` returns `(key_t)−1` and sets `errno` to indicate the error.

Errors The `ftok()` function will fail if:

`EACCES`	Search permission is denied for a component of the path prefix.
`ELOOP`	Too many symbolic links were encountered in resolving *path*.
`ENAMETOOLONG`	The length of the *path* argument exceeds `{PATH_MAX}` or a pathname component is longer than `{NAME_MAX}`.
`ENOENT`	A component of *path* does not name an existing file or *path* is an empty string.
`ENOTDIR`	A component of the path prefix is not a directory.

The `ftok()` function may fail if:

`ENAMETOOLONG`	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `{PATH_MAX}` .

Usage For maximum portability, *id* should be a single-byte character.

Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each

other's operation. It is still possible to interfere intentionally. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

Notes Since the `ftok()` function returns a value based on the *id* given and the file serial number of the file named by *path* in a type that is no longer large enough to hold all file serial numbers, it may return the same key for paths naming different files on large filesystems.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `msgget(2)`, `semget(2)`, `shmget(2)`, `stat(2)`, `attributes(5)`, `standards(5)`

Name ftw, nftw – walk a file tree

Synopsis

```
#include <ftw.h>

int ftw(const char *path, int (*fn) (const char *,
     const struct stat *, int), int depth);

int nftw(const char *path, int (*fn) (const char *,
     const struct stat *, int, struct FTW *), int depth,
     int flags);
```

Description The `ftw()` function recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, `ftw()` calls the user-defined function *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a `stat` structure (see `stat(2)`) containing information about the object, and an integer. Possible values of the integer, defined in the <`ftw.h`> header, are:

`FTW_F` The object is a file.

`FTW_D` The object is a directory.

`FTW_DNR` The object is a directory that cannot be read. Descendants of the directory are not processed.

`FTW_NS` The `stat()` function failed on the object because of lack of appropriate permission or the object is a symbolic link that points to a non-existent file. The `stat` buffer passed to *fn* is undefined.

The `ftw()` function visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a non-zero value, or some error is detected within `ftw()` (such as an I/O error). If the tree is exhausted, `ftw()` returns `0`. If *fn* returns a non-zero value, `ftw()` stops its tree traversal and returns whatever value was returned by *fn*.

The `nftw()` function is similar to `ftw()` except that it takes the additional argument *flags*, which is a bitwise-inclusive OR of zero or more of the following flags:

`FTW_CHDIR` If set, `nftw()` changes the current working directory to each directory as it reports files in that directory. If clear, `nftw()` does not change the current working directory.

`FTW_DEPTH` If set, `nftw()` reports all files in a directory before reporting the directory itself. If clear, `nftw()` reports any directory before reporting the files in that directory.

`FTW_MOUNT` If set, `nftw()` reports only files in the same file system as path. If clear, `nftw()` reports all files encountered during the walk.

`FTW_PHYS` If set, `nftw()` performs a physical walk and does not follow symbolic links.

If FTW_PHYS is clear and FTW_DEPTH is set, nftw() follows links instead of reporting them, but does not report any directory that would be a descendant of itself. If FTW_PHYS is clear and FTW_DEPTH is clear, nftw() follows links instead of reporting them, but does not report the contents of any directory that would be a descendant of itself.

At each file it encounters, nftw() calls the user-supplied function *fn* with four arguments:

- The first argument is the pathname of the object.
- The second argument is a pointer to the stat buffer containing information on the object.
- The third argument is an integer giving additional information. Its value is one of the following:

  FTW_F — The object is a file.

  FTW_D — The object is a directory.

  FTW_DP — The object is a directory and subdirectories have been visited. (This condition only occurs if the FTW_DEPTH flag is included in flags.)

  FTW_SL — The object is a symbolic link. (This condition only occurs if the FTW_PHYS flag is included in flags.)

  FTW_SLN — The object is a symbolic link that points to a non-existent file. (This condition only occurs if the FTW_PHYS flag is not included in flags.)

  FTW_DNR — The object is a directory that cannot be read. The user-defined function *fn* will not be called for any of its descendants.

  FTW_NS — The stat() function failed on the object because of lack of appropriate permission. The stat buffer passed to *fn* is undefined. Failure of stat() for any other reason is considered an error and nftw() returns −1.

- The fourth argument is a pointer to an FTW structure that contains the following members:

```
int   base;
int   level;
```

  The base member is the offset of the object's filename in the pathname passed as the first argument to *fn*(). The value of level indicates the depth relative to the root of the walk, where the root level is 0.

  The results are unspecified if the application-supplied *fn*() function does not preserve the current working directory.

Both ftw() and nftw() use one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors used. If *depth* is zero or negative, the effect is the same as if it were 1. It must not be greater than the number of file descriptors currently available for use. The ftw() function runs faster if *depth* is at least as large as the number of levels in the tree. Both ftw() and nftw() are able to descend to arbitrary depths in a file hierarchy and do not

fail due to path length limitations unless either the length of the path name pointed to by the *path* argument exceeds {PATH_MAX} requirements, or for ftw(), the specified depth is less than 2, or for nftw(), the specified depth is less than 2 and FTW_CHDIR is not set. When ftw() and nftw() return, they close any file descriptors they have opened; they do not close any file descriptors that might have been opened by *fn*.

Return Values If the tree is exhausted, ftw() and nftw() return 0. If the function pointed to by *fn* returns a non-zero value, ftw() and nftw() stop their tree traversal and return whatever value was returned by the function pointed to by *fn*. If ftw() and nftw() detect an error, they return −1 and set errno to indicate the error.

If ftw() and nftw() encounter an error other than EACCES (see FTW_DNR and FTW_NS above), they return −1 and set errno to indicate the error. The external variable errno can contain any error value that is possible when a directory is opened or when one of the stat functions is executed on a directory or file.

Errors The ftw() and nftw() functions will fail if:

ELOOP — A loop exists in symbolic links encountered during resolution of the *path* argument

ENAMETOOLONG — The length of the path name pointed to by the *path* argument exceeds {PATH_MAX}, or a path name component is longer than {NAME_MAX}.

ENOENT — A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR — A component of *path* is not a directory.

EOVERFLOW — A field in the stat structure cannot be represented correctly in the current programming environment for one or more files found in the file hierarchy.

The ftw() function will fail if:

EACCES — Search permission is denied for any component of *path* or read permission is denied for *path*.

ENAMETOOLONG — The ftw() function has descended to a path that exceeds {PATH_MAX} and the depth argument specified by the application is less than 2 and FTW_CHDIR is not set.

The nftw() function will fail if:

EACCES — Search permission is denied for any component of *path* or read permission is denied for *path*, or *fn*() returns −1 and does not reset errno.

The nftw() and ftw() functions may fail if:

ELOOP	Too many symbolic links were encountered during resolution of the *path* argument.
ENAMETOOLONG	Pathname resolution of a symbolic link in the path name pointed to by the *path* argument produced an intermediate result whose length exceeds {PATH_MAX}.

The ftw() function may fail if:

EINVAL	The value of the *depth* argument is invalid.

The nftw() function may fail if:

EMFILE	There are {OPEN_MAX} file descriptors currently open in the calling process.
ENFILE	Too many files are currently open in the system.

If the function pointed to by *fn* encounters system errors, errno may be set accordingly.

Examples

EXAMPLE 1 Walk a directory structure using ftw().

The following example walks the current directory structure, calling the *fn*() function for every directory entry, using at most 10 file descriptors:

```
#include <ftw.h>
...
if (ftw(".", fn, 10) != 0) {
       perror("ftw"); exit(2);
}
```

EXAMPLE 2 Walk a directory structure using nftw().

The following example walks the /tmp directory and its subdirectories, calling the nftw() function for every directory entry, to a maximum of 5 levels deep.

```
#include <ftw.h>
...
int nftwfunc(const char *, const struct stat *, int, struct FTW *);
int nftwfunc(const char *filename, const struct stat *statptr,
      int fileflags, struct FTW *pfwt)
{
      return 0;
}
...
char *startpath = "/tmp";
int depth = 5;
int flags = FTW_CHDIR | FTW_DEPTH | FTW_MOUNT;
int ret;
ret = nftw(startpath, nftwfunc, depth, flags);
```

Usage Because `ftw()` and `nftw()` are recursive, they can terminate with a memory fault when applied by a thread with a small stack to very deep file structures.

The `ftw()` and `nftw()` functions allocate resources (memory, file descriptors) during their operation. If `ftw()` they are forcibly terminated, such as by `longjmp(3C)` being executed by *fn* or an interrupt routine, they will not have a chance to free those resources, so they remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred and arrange to have *fn* return a non-zero value at its next invocation.

The `ftw()` and `nftw()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

The `ftw()` function is safe in multithreaded applications. The `nftw()` function is safe in multithreaded applications when the `FTW_CHDIR` flag is not set.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `stat(2)`, `longjmp(3C)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Name fwide – set stream orientation

Synopsis

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

int fwide(FILE *stream, int mode);
```

Description The `fwide()` function determines the orientation of the stream pointed to by *stream*. If *mode* is greater than 0, the function first attempts to make the stream wide-orientated. If *mode* is less than 0, the function first attempts to make the stream byte-orientated. Otherwise, *mode* is 0 and the function does not alter the orientation of the stream.

If the orientation of the stream has already been determined, `fwide()` does not change it.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to 0, then call `fwide()`, then check `errno` and if it is non-zero, assume an error has occurred.

Return Values The `fwide()` function returns a value greater than 0 if, after the call, the stream has wide-orientation, a value less than 0 if the stream has byte-orientation, or 0 if the stream has no orientation.

Errors The `fwide()` function may fail if:

`EBADF` The *stream* argument is not a valid stream.

Usage A call to `fwide()` with *mode* set to 0 can be used to determine the current orientation of a stream.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `attributes(5)`, `standards(5)`

Name fwprintf, wprintf, swprintf – print formatted wide-character output

Synopsis

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

int fwprintf(FILE *restrict stream, const wchar_t *restrict format,
     ...);

int wprintf(const wchar_t *restrict format, ...);

int swprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict format,
     ...);
```

Description The `fwprintf()` function places output on the named output *stream*. The `wprintf()` function places output on the standard output stream `stdout`. The `swprintf()` function places output followed by the null wide-character in consecutive wide-characters starting at **s*; no more than *n* wide-characters are written, including a terminating null wide-character, which is always added (unless *n* is zero).

Each of these functions converts, formats and prints its arguments under control of the *format* wide-character string. The *format* is composed of zero or more directives: *ordinary wide-characters*, which are simply copied to the output stream and *conversion specifications*, each of which results in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion wide-character `%` (see below) is replaced by the sequence `%`*n*`$`, where *n* is a decimal integer in the range [1, `NL_ARGMAX`], giving the position of the argument in the argument list. This feature provides for the definition of format wide-character strings that select arguments in an order appropriate to specific languages (see the `EXAMPLES` section).

In format wide-character strings containing the `%`*n*`$` form of conversion specifications, numbered arguments in the argument list can be referenced from the format wide-character string as many times as required.

In format wide-character strings containing the `%` form of conversion specifications, each argument in the argument list is used exactly once.

All forms of the `fwprintf()` functions allow for the insertion of a language-dependent radix character in the output string, output as a wide-character value. The radix character is defined in the program's locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

Each conversion specification is introduced by the `%` wide-character or by the wide-character sequence `%`*n*`$`, after which the following appear in sequence:

- Zero or more *flags* (in any order), which modify the meaning of the conversion specification.

- An optional minimum *field width*. If the converted value has fewer wide-characters than the field width, it will be padded with spaces by default on the left; it will be padded on the right, if the left-adjustment flag (–), described below, is given to the field width. The field width takes the form of an asterisk (*), described below, or a decimal integer.
- An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions; the number of digits to appear after the radix character for the `a`, `A`, `e`, `E`, `f`, and `F` conversions; the maximum number of significant digits for the `g` and `G` conversions; or the maximum number of wide-characters to be printed from a string in `s` conversions. The precision takes the form of a period (.) followed by either an asterisk (*), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion wide-character, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A *conversion specifier* wide character that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (*). In this case an argument of type `int` supplies the field width or precision. Arguments specifying field width, or precision, or both must appear in that order before the argument, if any, to be converted. A negative field width is taken as a – flag followed by a positive field width. A negative precision is taken as if the precision were omitted. In format wide-character strings containing the `%`*n*`$` form of a conversion specification, a field width or precision may be indicated by the sequence **m*`$`, where *m* is a decimal integer in the range [1, `NL_ARGMAX`] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
wprintf(L"%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The *format* can contain either numbered argument specifications (that is, `%`*n*`$` and **m*`$)`, or unnumbered argument specifications (that is, `%` and ***), but normally not both. The only exception to this is that `%%` can be mixed with the `%`*n*`$` form. The results of mixing numbered and unnumbered argument specifications in a *format* wide-character string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N–1*)th, are specified in the format wide-character string.

The flag wide-characters and their meanings are:

`'` The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%F`, `%g`, or `%G`) will be formatted with thousands' grouping wide-characters. For other conversions the behavior is undefined. The non-monetary grouping wide-character is used.

– The result of the conversion will be left-justified within the field. The conversion will be right-justified if this flag is not specified.

`+` The result of a signed conversion will always begin with a sign (+ or –). The conversion will begin with a sign only when a negative value is converted if this flag is not specified.

space If the first wide-character of a signed conversion is not a sign or if a signed conversion results in no wide-characters, a space will be prefixed to the result. This means that if the space and + flags both appear, the space flag will be ignored.

`#` This flag specifies that the value is to be converted to an alternative form. For `o` conversion, it increases the precision (if necessary) to force the first digit of the result to be 0. For `x` or `X` conversions, a non-zero result will have 0x (or 0X) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversions, the result will always contain a radix character, even if no digits follow it. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For `g` and `G` conversions, trailing zeros will *not* be removed from the result as they normally are. For other conversions, the behavior is undefined.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `0` and – flags both appear, the `0` flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag will be ignored. If the `0` and `'` flags both appear, the grouping wide-characters are inserted before zero padding. For other conversions, the behavior is undefined.

The length modifiers and their meanings:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `signed char` or `unsigned char` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `signed char` or `unsigned char` before printing); or that a following `n` conversion specifier applies to a pointer to a `signed char` argument.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `short` or `unsigned short` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `short` or `unsigned short` before printing); or that a following `n` conversion specifier applies to a pointer to a `short` argument.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long` or `unsigned long` argument; that a following n conversion specifier applies to a pointer to a `long` argument; that a following `c` conversion specifier applies to a `wint_t` argument; that a following `s` conversion specifier applies to a pointer to a `wchar_t` argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

`ll` (ell-ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long long` or unsigned `long long` argument; or that a following `n` conversion specifier applies to a pointer to a `long long` argument.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an `intmax_t` or `uintmax_t` argument; or that a following `n` conversion specifier applies to a pointer to an `intmax_t` argument.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `size_t` or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to `size_t` argument.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned type argument; or that a following `n` conversion specifier applies to a pointer to a `ptrdiff_t` argument.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a long double argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion wide-characters and their meanings are:

`d`, `i` The `int` argument is converted to a signed decimal in the style `[−]`*dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

`o` The `unsigned int` argument is converted to unsigned octal format in the style *dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

`u` The `unsigned int` argument is converted to unsigned decimal format in the style *dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

`x` The `unsigned int` argument is converted to unsigned hexadecimal format in the style *dddd*; the letters abcdef are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

`X` Behaves the same as the `x` conversion wide-character except that letters "`ABCDEF`" are used instead of "`abcdef`".

`f`, `F` The `double` argument is converted to decimal notation in the style [−]*ddd*.*ddd*, where the number of digits after the radix character (see `setlocale(3C)`) is equal to the precision specification. If the precision is missing it is taken as 6; if the precision is explicitly 0 and the # flag is not specified, no radix character appears. If a radix character appears, at least 1 digit appears before it. The converted value is rounded to fit the specified output format according to the prevailing floating point rounding direction mode. If the conversion is not exact, an inexact exception is raised.

For the `f` specifier, a double argument representing an infinity or NaN is converted in the style of the `e` conversion specifier, except that for an infinite argument, "infinity" or "Infinity" is printed when the precision is at least 8 and "inf" or "Inf" is printed otherwise.

For the F specifier, a double argument representing an infinity or NaN is converted in the SUSv3 style of the E conversion specifier, except that for an infinite argument, "INFINITY" is printed when the precision is at least 8 and or "INF" is printed otherwise.

`e`, `E` The `double` argument is converted in the style `[−]`*d.ddd*`e` ± dd, where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is 0 and no # flag is present, no radix character appears. The converted value is rounded to fit the specified output format according to the prevailing floating point rounding direction mode. If the conversion is not exact, an inexact exception is raised. The `E` conversion wide-character will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. If the value is 0, the exponent is 0.

Infinity and NaN values are handled in one of the following ways:

SUSv3 For the `e` specifier, a `double` argument representing an infinity is printed as "[−]`infinity`", when the precision for the conversion is at least 7 and as "[−]`inf`" otherwise. A `double` argument representing a NaN is printed as "[−]`nan`". For the `E` specifier, "`INF`", "`INFINITY`", and "`NAN`" are printed instead of "`inf`", "`infinity`", and "`nan`", respectively. Printing of the sign follows the rules described above.

Default A `double` argument representing an infinity is printed as "[−]`Infinity`", when the precision for the conversion is at least 7 and as "[−]`Inf`" otherwise. A double argument representing a NaN is printed as "[−]`NaN`". Printing of the sign follows the rules described above.

`g`, `G` The `double` argument is converted in the style `f` or `e` (or in the style `E` in the case of a `G` conversion wide-character), with the precision specifying the number of significant

digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted; style `e` (or `E`) will be used only if the exponent resulting from such a conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a radix character appears only if it is followed by a digit.

A `double` argument representing an infinity or NaN is converted in the style of the `e` or `E` conversion specifier, except that for an infinite argument, "infinity", "INFINITY", or "Infinity" is printed when the precision is at least 8 and "inf", "INF", or "Inf" is printed otherwise.

`a`, `A` A double argument representing a floating-point number is converted in the style "[-]`0xh.hhhhp±d`", where the single hexadecimal digit preceding the radix point is 0 if the value converted is zero and 1 otherwise and the number of hexadecimal digits after it are equal to the precision; if the precision is missing, the number of digits printed after the radix point is 13 for the conversion of a double value, 16 for the conversion of a long double value on x86, and 28 for the conversion of a long double value on SPARC; if the precision is zero and the '#' flag is not specified, no decimal-point wide character appears. The letters "abcdef" are used for `a` conversion and the letters "ABCDEF" for `A` conversion. The `A` conversion specifier produces a number with 'X' and 'P' instead of 'x' and 'p'. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

The converted valueis rounded to fit the specified output format according to the prevailing floating point rounding direction mode. If the conversion is not exact, an inexact exception is raised.

A `double` argument representing an infinity or NaN is converted in the SUSv3 style of an `e` or `E` conversion specifier.

`c` If no `l` (ell) qualifier is present, the `int` argument is converted to a wide-character as if by calling the `btowc(3C)` function and the resulting wide-character is written. Otherwise the `wint_t` argument is converted to `wchar_t`, and written.

`s` If no `l` (ell) qualifier is present, the argument must be a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the `mbrtowc(3C)` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted, and written up to (but not including) the terminating null wide-character. If the precision is specified, no more than that many wide-characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null wide-character.

If an `l` (ell) qualifier is present, the argument must be a pointer to an array of type `wchar_t`. Wide characters from the array are written up to (but not including) a

terminating null wide-character. If no precision is specified or is greater than the size of the array, the array must contain a null wide-character. If a precision is specified, no more than that many wide-characters are written.

`p` The argument must be a pointer to `void`. The value of the pointer is converted to a sequence of printable wide-characters.

`n` The argument must be a pointer to an integer into which is written the number of wide-characters written to the output so far by this call to one of the `fwprintf()` functions. No argument is converted.

`C` Same as `lc`.

`S` Same as `ls`.

`%` Output a % wide-character; no argument is converted. The entire conversion specification must be `%%`.

If a conversion specification does not match one of the above forms, the behavior is undefined.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by `fwprintf()` and `wprintf()` are printed as if `fputwc(3C)` had been called.

The `st_ctime` and `st_mtime` fields of the file will be marked for update between the call to a successful execution of `fwprintf()` or `wprintf()` and the next successful completion of a call to `fflush(3C)` or `fclose(3C)` on the same stream or a call to `exit(3C)` or `abort(3C)`.

Return Values Upon successful completion, these functions return the number of wide-characters transmitted excluding the terminating null wide-character in the case of `swprintf()` or a negative value if an output error was encountered.

If *n* or more wide characters were requested to be written, `swprintf()` returns a negative value.

Errors For the conditions under which `fwprintf()` and `wprintf()` will fail and may fail, refer to `fputwc(3C)`.

In addition, all forms of `fwprintf()` may fail if:

`EILSEQ` A wide-character code that does not correspond to a valid character has been detected.

`EINVAL` There are insufficient arguments.

In addition, `wprintf()` and `fwprintf()` may fail if:

`ENOMEM` Insufficient storage space is available.

Examples

EXAMPLE 1 Print Language-dependent Date and Time Format.

To print the language-independent date and time format, the following statement could be used:

```
wprintf(format, weekday, month, day, hour, min);
```

For American usage, *format* could be a pointer to the wide-character string:

```
L"%s, %s %d, %d:%.2d\n"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, *format* could be a pointer to the wide-character string:

```
L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the message:

```
Sonntag, 3. Juli, 10:02
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also

`btowc(3C)`, `fputwc(3C)`, `fwscanf(3C)`, `mbrtowc(3C)`, `setlocale(3C)`, `attributes(5)`, `standards(5)`

Notes

The `fwprintf()`, `wprintf()`, and `swprintf()` functions can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

If the `j` length modifier is used, 32-bit applications that were compiled using `c89` on releases prior to Solaris 10 will experience undefined behavior.

Name fwrite – binary output

Synopsis

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
```

Description The `fwrite()` function writes, from the array pointed to by *ptr*, up to *nitems* elements whose size is specified by *size*, to the stream pointed to by *stream*. For each object, *size* calls are made to the `fputc(3C)` function, taking the values (in order) from an array of `unsigned char` exactly overlaying the object. The file-position indicator for the stream (if defined) is advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is unspecified.

The `st_ctime` and `st_mtime` fields of the file will be marked for update between the successful execution of `fwrite()` and the next successful completion of a call to `fflush(3C)` or `fclose(3C)` on the same stream or a call to `exit(2)` or `abort(3C)`.

Return Values The `fwrite()` function returns the number of elements successfully written, which might be less than *nitems* if a write error is encountered. If *size* or *nitems* is 0, `fwrite()` returns 0 and the state of the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for the stream is set and `errno` is set to indicate the error.

Errors Refer to `fputc(3C)`.

Usage Because of possible differences in element length and byte ordering, files written using `fwrite()` are application-dependent, and possibly cannot be read using `fread(3C)` by a different application or by the same application on a different processor.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `write(2)`, `fclose(3C)`, `ferror(3C)`, `fopen(3C)`, `fread(3C)`, `getc(3C)`, `gets(3C)`, `printf(3C)`, `putc(3C)`, `puts(3C)`, `attributes(5)`, `standards(5)`

Name fwscanf, wscanf, swscanf, vfwscanf, vwscanf, vswscanf – convert formatted wide-character input

Synopsis

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

int fwscanf(FILE *restrict stream, const wchar_t *restrict format, ...);

int wscanf(const wchar_t *restrict format, ...);

int swscanf(const wchar_t *restrict s, const wchar_t *restrict format,
     ...);

#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwscanf(FILE *restrict stream, const wchar_t *restrict format,
     va_list arg);

int vswcanf(const wchar_t *restrict ws, const wchar_t *restrict format,
     va_list arg);

int vwscanf(const wchar_t *restrict format, va_list arg);
```

Description The `fwscanf()` function reads from the named input *stream*.

The `wscanf()` function reads from the standard input stream `stdin`.

The `swscanf()` function reads from the wide-character string *s*.

The `vfwscanf()`, `vswscanf()`, and `vwscanf()` functions are equivalent to the `fwscanf()`, `swscanf()`, and `wscanf()` functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by the <`stdarg.h`> header . These functions do not invoke the `va_end()` macro. Applications using these functions should call `va_end`(*ap*) afterwards to clean up.

Each function reads wide-characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control wide-character string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion wide-character `%` (see below) is replaced by the sequence `%`*n*`$`, where *n* is a decimal integer in the range [1, `NL_ARGMAX`]. This feature provides for the definition of format wide-character strings that select arguments in an order appropriate to specific languages. In format wide-character strings containing the `%`*n*`$` form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format wide-character string more than once.

The *format* can contain either form of a conversion specification, that is, `%` or `%`*n*`$`, but the two forms cannot normally be mixed within a single *format* wide-character string. The only exception to this is that `%%` or `%*` can be mixed with the `%`*n*`$` form.

The `fwscanf()` function in all its forms allows for detection of a language-dependent radix character in the input string, encoded as a wide-character value. The radix character is defined in the program's locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

The format is a wide-character string composed of zero or more directives. Each directive is composed of one of the following: one or more white-space wide-characters (space, tab, newline, vertical-tab or form-feed characters); an ordinary wide-character (neither `%` nor a white-space character); or a conversion specification. Each conversion specification is introduced by a `%` or the sequence `%`*n*`$` after which the following appear in sequence:

- An optional assignment-suppressing character *.
- An optional non-zero decimal integer that specifies the maximum field width.
- An option length modifier that specifies the size of the receiving object.
- A conversion specifier wide-character that specifies the type of conversion to be applied. The valid conversion wide-characters are described below.

The `fwscanf()` functions execute each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space wide-characters is executed by reading input until no more valid input can be read, or up to the first wide-character which is not a white-space wide-character, which remains unread.

A directive that is an ordinary wide-character is executed as follows. The next wide-character is read from the input and compared with the wide-character that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent wide-characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion wide-character. A conversion specification is executed in the following steps:

Input white-space wide-characters (as specified by `iswspace(3C)`) are skipped, unless the conversion specification includes a `[`, `c`, or `n` conversion character.

An item is read from the input unless the conversion specification includes an `n` conversion wide-character. The length of the item read is limited to any specified maximum field width. In Solaris default mode, the input item is defined as the longest sequence of input wide-characters that forms a matching sequence. In some cases, `fwscanf()` might need to read several extra wide-characters beyond the end of the input item to find the end of a

matching sequence. In C99/SUSv3 mode, the input item is defined as the longest sequence of input wide-characters that is, or is a prefix of, a matching sequence. With this definition, `fwscanf()` need only read at most one wide-character beyond the end of the input item. Therefore, in C99/SUSv3 mode, some sequences that are acceptable to `wcstod(3C)`, `wcstol(3C)`, and similar functions are unacceptable to `fwscanf()`. In either mode, `fwscanf()` attempts to push back any excess bytes read using `ungetc(3C)`. Assuming all such attempts succeed, the first wide-character, if any, after the input item remains unread. If the length of the input item is 0, the conversion fails. This condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` conversion wide-character, the input item (or, in the case of a `%`*n* conversion specification, the count of input wide-characters) is converted to a type appropriate to the conversion wide-character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result if the conversion specification is introduced by `%`, or in the *n*th argument if introduced by the wide-character sequence `%`*n*`$`. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `signed char` or `unsigned char`.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `short` or `unsigned short`.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `long` or `unsigned long`; that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to `double`; or that a following `c`, `s`, or `[` conversion specifier applies to an argument with type pointer to `wchar_t`.

`ll` (ell-ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `long long` or `unsigned long long`.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `intmax_t` or `uintmax_t`.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `size_t` or the corresponding signed integer type.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `ptrdiff_t` or the corresponding `unsigned` type.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to `long double`.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The following conversion wide-characters are valid:

`d` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `wcstol(3C)` with the value 10 for the *base* argument. In the absence of a size modifier, the corresponding argument must be a pointer to `int`.

`i` Matches an optionally signed integer, whose format is the same as expected for the subject sequence of `wcstol(3C)` with 0 for the *base* argument. In the absence of a size modifier, the corresponding argument must be a pointer to `int`.

`o` Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of `wcstoul(3C)` with the value 8 for the *base* argument. In the absence of a size modifier, the corresponding argument must be a pointer to `unsigned int`.

`u` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `wcstoul(3C)` with the value 10 for the *base* argument. In the absence of a size modifier, the corresponding argument must be a pointer to `unsigned int`.

`x` Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of `wcstoul(3C)` with the value 16 for the *base* argument. In the absence of a size modifier, the corresponding argument must be a pointer to `unsigned int`.

`a,e,f,g` Matches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of `wcstod(3C)`. In the absence of a size modifier, the corresponding argument must be a pointer to `float`. The `e`, `f`, and `g` specifiers match hexadecimal floating point values only in C99/SUSv3 (see `standards(5)`) mode, but the `a` specifier always matches hexadecimal floating point values.

These conversion specifiers match any subject sequence accepted by `strtod(3C)`, including the INF, INFINITY, NAN, and NAN(*n-char-sequence*) forms. The result of the conversion is the same as that of calling `strtod()` (or `strtof()` or `strtold()`) with the matching sequence, including the raising of floating point exceptions and the setting of `errno` to `ERANGE`, if applicable.

`s` Matches a sequence of non white-space wide-characters. If no `l` (ell) qualifier is present, characters from the input field are converted as if by repeated calls to the `wcrtomb(3C)` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide-character is converted. The corresponding

argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically.

Otherwise, the corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

`[` Matches a non-empty sequence of wide-characters from a set of expected wide-characters (the *scanset*). If no `l` (ell) qualifier is present, wide-characters from the input field are converted as if by repeated calls to the `wcrtomb()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically.

If an `l` (ell) qualifier is present, the corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

The conversion specification includes all subsequent widw characters in the *format* string up to and including the matching right square bracket (`]`). The wide-characters between the square brackets (the *scanlist*) comprise the scanset, unless the wide-character after the left square bracket is a circumflex (`^`), in which case the scanset contains all wide-characters that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with `[ ]` or `[^]`, the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise the first right square bracket is the one that ends the conversion specification. If a minus-sign (–) is in the scanlist and is not the first wide-character, nor the second where the first wide-character is a `^`, nor the last wide-character, it indicates a range of characters to be matched.

`c` Matches a sequence of wide-characters of the number specified by the field width (1 if no field width is present in the conversion specification). If no `l` (ell) qualifier is present, wide-characters from the input field are converted as if by repeated calls to the `wcrtomb()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence. No null character is added.

Otherwise, the corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the sequence. No null wide-character is added.

`p` Matches the set of sequences that is the same as the set of sequences that is produced by the `%p` conversion of the corresponding `fwprintf(3C)` functions. The corresponding argument must be a pointer to a pointer to `void`. If the input item is

a value converted earlier during the same program execution, the pointer that results will compare equal to that value; otherwise the behavior of the `%p` conversion is undefined.

`n` No input is consumed. The corresponding argument must be a pointer to the integer into which is to be written the number of wide-characters read from the input so far by this call to the `fwscanf()` functions. Execution of a `%n` conversion specification does not increment the assignment count returned at the completion of execution of the function.

`C` Same as `lc`.

`S` Same as `ls`.

`%` Matches a single `%`; no conversion or assignment occurs. The complete conversion specification must be `%%`.

If a conversion specification is invalid, the behavior is undefined.

The conversion characters `A`, `E`, `F`, `G`, and `X` are also valid and behave the same as, respectively, `a`, `e`, `f`, `g`, and `x`.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide-characters matching the current conversion specification (except for `%n`) have been read (other than leading white-space, where permitted), execution of the current conversion specification terminates with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) is terminated with an input failure.

Reaching the end of the string in `swscanf()` is equivalent to encountering end-of-file for `fwscanf()`.

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white space (including newline) is left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the %n conversion specification.

The `fwscanf()` and `wscanf()` functions may mark the `st_atime` field of the file associated with *stream* for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc(3C)`, `fgetwc(3C)`, `fgets(3C)`, `fgetws(3C)`, `fread(3C)`, `getc(3C)`, `getwc(3C)`, `getchar(3C)`, `getwchar(3C)`, `gets(3C)`, `fscanf(3C)` or `fwscanf()` using *stream* that returns data not supplied by a prior call to `ungetc(3C)`.

Return Values Upon successful completion, these functions return the number of successfully matched and assigned input items; this number can be 0 in the event of an early matching failure. If the

input ends before the first matching failure or conversion, EOF is returned. If a read error occurs the error indicator for the stream is set, EOF is returned, and `errno` is set to indicate the error.

Errors For the conditions under which the `fwscanf()` functions will fail and may fail, refer to `fgetwc(3C)`.

In addition, `fwscanf()` may fail if:

`EILSEQ` Input byte sequence does not form a valid character.

`EINVAL` There are insufficient arguments.

Usage In format strings containing the `%` form of conversion specifications, each argument in the argument list is used exactly once.

Examples **EXAMPLE 1** `wscanf()` example

The call:

```
int i, n; float x; char name[50];
n = wscanf(L"%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E–1 Hamster
```

will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain the string Hamster.

The call:

```
int i; float x; char name[50];
(void) wscanf(L"%2d%f%*d %[0123456789], &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip 0123, and place the string 56\0 in *name*. The next call to `getchar(3C)` will return the character `a`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also fgetc(3C), fgets(3C), fgetwc(3C), fgetws(3C), fread(3C), fscanf(3C), fwprintf(3C), getc(3C), getchar(3C), gets(3C), getwc(3C), getwchar(3C), setlocale(3C), strtod(3C), wcrtomb(3C), wcstod(3C), wcstol(3C), wcstoul(3C), attributes(5), standards(5)

Notes The behavior of the conversion specifier "%%" has changed for all of the functions described on this manual page. Previously the "%%" specifier accepted a "%" character from input only if there were no preceding whitespace characters. The new behavior accepts "%" even if there are preceding whitespace characters. This new behavior now aligns with the description on this manual page and in various standards. If the old behavior is desired, the conversion specification "%*[%]" can be used.

Name getcpuid, gethomelgroup – obtain information on scheduling decisions

Synopsis

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

processorid_t getcpuid(void);

ushort_t gethomelgroup(void);
```

Description The `getcpuid()` function returns the processor ID on which the calling thread is currently executing.

The `gethomelgroup()` function returns the home locality group ID of the calling thread.

Return Values See `DESCRIPTION`.

Errors No errors are defined.

Usage Both the current CPU and the home locality group can change at any time.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	MT-Safe

The `getcpuid()` function is Committed. The `gethomelgroup()` function is Obsolete.

See Also `psradm(1M)`, `psrinfo(1M)`, `psrset(1M)`, `p_online(2)`, `processor_bind(2)`, `processor_info(2)`, `pset_assign(2)`, `pset_bind(2)`, `pset_info(2)`, `meminfo(2)`, `lgrp_home(3LGRP)`, `sysconf(3C)`, `attributes(5)`

Notes The `gethomelgroup()` function is obsolete and might be removed in a future release. It has been replaced by `lgrp_home(3LGRP)`.

Name getcwd – get pathname of current working directory

Synopsis

```
#include <unistd.h>

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

Description The `getcwd()` function places an absolute pathname of the current working directory in the array pointed to by *buf*, and returns *buf*. The pathname copied to the array contains no components that are symbolic links. The *size* argument is the size in bytes of the character array pointed to by *buf* and must be at least one greater than the length of the pathname to be returned.

If *buf* is not a null pointer, the pathname is stored in the space pointed to by *buf*.

If *buf* is a null pointer, `getcwd()` obtains *size* bytes of space using `malloc(3C)`. The pointer returned by `getcwd()` can be used as the argument in a subsequent call to `free()`.

Return Values Upon successful completion, `getcwd()` returns the *buf* argument. If *buf* is an invalid destination buffer address, `NULL` is returned and `errno` is set to `EFAULT`. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

Errors The `getcwd()` function will fail if:

`EFAULT` The *buf* argument is an invalid destination buffer address.

`EINVAL` The *size* argument is equal to 0.

`ERANGE` The *size* argument is greater than 0 and less than the length of the pathname plus 1.

The `getcwd()` function may fail if:

`EACCES` A parent directory cannot be read to get its name.

`ENOMEM` Insufficient storage space is available.

Examples **EXAMPLE 1** Determine the absolute pathname of the current working directory.

The following example returns a pointer to an array that holds the absolute pathname of the current working directory. The pointer is returned in the *ptr* variable, which points to the *buf* array where the pathname is stored.

```
#include <stdlib.h>
#include <unistd.h>
...
long size;
char *buf;
char *ptr;
size = pathconf(".", _PC_PATH_MAX);
if ((buf = (char *)malloc((size_t)size)) != NULL)
       ptr = getcwd(buf, (size_t)size);
```

EXAMPLE 1 Determine the absolute pathname of the current working directory. *(Continued)*

```
...
```

EXAMPLE 2 Print the current working directory.

The following example prints the current working directory.

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

main( )
{
    char *cwd;
    if ((cwd = getcwd(NULL, 64)) == NULL) {
        perror("pwd");
        exit(2);
    }
    (void)printf("%s\n", cwd);
    free(cwd); /* free memory allocated by getcwd() */
    return(0);
}
```

Usage

Applications should exercise care when using chdir(2) in conjunction with `getcwd()`. The current working directory is global to all threads within a process. If more than one thread calls `chdir()` to change the working directory, a subsequent call to `getcwd()` could produce unexpected results.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also

chdir(2), malloc(3C), attributes(5), standards(5)

Name getdate – convert user format date and time

Synopsis

```
#include <time.h>

struct tm *getdate(const char *string);
extern int getdate_err;
```

Description The `getdate()` function converts user-definable date and/or time specifications pointed to by *string* to a `tm` structure. The `tm` structure is defined in the `<time.h>` header.

User-supplied templates are used to parse and interpret the input string. The templates are text files created by the user and identified via the environment variable `DATEMSK`. Each line in the template represents an acceptable date and/or time specification using conversion specifications similar to those used by `strftime(3C)` and `strptime(3C)`. Dates before 1902 and after 2037 are illegal. The first line in the template that matches the input specification is used for interpretation and conversion into the internal time format.

Conversion Specifications

The following conversion specifications are supported:

`%%` Same as `%`.

`%a` Locale's abbreviated weekday name.

`%A` Locale's full weekday name.

`%b` Locale's abbreviated month name.

`%B` Locale's full month name.

`%c` Locale's appropriate date and time representation.

`%C` Century number (the year divided by 100 and truncated to an integer as a decimal number [1,99]); single digits are preceded by 0; see `standards(5)`. If used without the `%y` specifier, this format specifier will assume the current year offset in whichever century is specified. The only valid years are between 1902-2037.

`%d` day of month [01,31]; leading zero is permitted but not required.

`%D` Date as `%m/%d/%y`.

`%e` Same as `%d`.

`%h` Locale's abbreviated month name.

`%H` Hour (24-hour clock) [0,23]; leading zero is permitted but not required.

`%I` Hour (12-hour clock) [1,12]; leading zero is permitted but not required.

`%j` Day number of the year [1,366]; leading zeros are permitted but not required.

`%m` Month number [1,12]; leading zero is permitted but not required.

`%M` Minute [0,59]; leading zero is permitted but not required.

`%n` Any white space.

`%p` Locale's equivalent of either a.m. or p.m.

`%r` Appropriate time representation in the 12-hour clock format with `%p`.

`%R` Time as `%H:%M`.

SUSv3

`%S` Seconds [0,60]; leading zero is permitted but not required. The range of values is [00,60] rather than [00,59] to allow for the occasional leap second.

Default and other standards

`%S` Seconds [0,61]; leading zero is permitted but not required. The range of values is [00,61] rather than [00,59] to allow for the occasional leap second and even more occasional double leap second.

`%t` Any white space.

`%T` Time as `%H:%M:%S`.

`%U` Week number of the year as a decimal number [0,53], with Sunday as the first day of the week; leading zero is permitted but not required.

`%w` Weekday as a decimal number [0,6], with 0 representing Sunday.

`%W` Week number of the year as a decimal number [0,53], with Monday as the first day of the week; leading zero is permitted but not required.

`%x` Locale's appropriate date representation.

`%X` Locale's appropriate time representation.

`%y` Year within century. When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969 to 1999 inclusive); values in the range 00-68 refer to years in the twenty-first century (2000 to 2068 inclusive).

`%Y` Year, including the century (for example, 1993).

`%Z` Time zone name or no characters if no time zone exists.

Modified Conversion Specifications

Some conversion specifications can be modified by the `E` and `O` modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified specification. If the alternative format or specification does not exist in the current locale, the behavior be as if the unmodified conversion specification were used.

`%Ec` Locale's alternative appropriate date and time representation.

`%EC` Name of the base year (period) in the locale's alternative representation.

`%Ex` Locale's alternative date representation.

`%EX` Locale's alternative time representation.

`%Ey` Offset from `%EC` (year only) in the locale's alternative representation.

`%EY` Full alternative year representation.

`%Od` Day of the month using the locale's alternative numeric symbols; leading zeros are permitted but not required.

`%Oe` Same as `%Od`.

`%OH` Hour (24-hour clock) using the locale's alternative numeric symbols.

`%OI` Hour (12-hour clock) using the locale's alternative numeric symbols.

`%Om` Month using the locale's alternative numeric symbols.

`%OM` Minutes using the locale's alternative numeric symbols.

`%OS` Seconds using the locale's alternative numeric symbols.

`%OU` Week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.

`%Ow` Number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

`%OW` Week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

`%Oy` Year (offset from `%C`) in the locale's alternative representation and using the locale's alternative numeric symbols.

Internal Format Conversion

The following rules are applied for converting the input specification into the internal format:

- If only the weekday is given, today is assumed if the given day is equal to the current day and next week if it is less.
- If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given. (The first day of month is assumed if no day is given.)
- If only the year is given, the values of the `tm_mon`, `tm_mday`, `tm_yday`, `tm_wday`, and `tm_isdst` members of the returned `tm` structure are not specified.
- If the century is given, but the year within the century is not given, the current year within the century is assumed.
- If no hour, minute, and second are given, the current hour, minute, and second are assumed.
- If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

General Specifications

A conversion specification that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the conversion specification, the specification fails, and the differing and subsequent characters remain unscanned.

A series of conversion specifications composed of `%n`, `%t`, white space characters, or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

Any other conversion specification is executed by scanning characters until a character matching the next conversion specification is scanned, or until no more characters can be scanned. These characters, except the one matching the next conversion specification, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate *tm* structure members are set to values corresponding to the locale information. If no match is found, `getdate()` fails and no more characters are scanned.

The month names, weekday names, era names, and alternative numeric symbols can consist of any combination of upper and lower case letters. The user can request that the input date or time specification be in a specific language by setting the `LC_TIME` category using `setlocale(3C)`.

Return Values

If successful, `getdate()` returns a pointer to a `tm` structure; otherwise, it returns `NULL` and sets the global variable `getdate_err` to indicate the error. Subsequent calls to `getdate()` alter the contents of `getdate_err`.

The following is a complete list of the `getdate_err` settings and their meanings:

1 The `DATEMSK` environment variable is null or undefined.

2 The template file cannot be opened for reading.

3 Failed to get file status information.

4 The template file is not a regular file.

5 An error is encountered while reading the template file.

6 The `malloc()` function failed (not enough memory is available).

7 There is no line in the template that matches the input.

8 The input specification is invalid (for example, `February 31`).

Usage

The `getdate()` function makes explicit use of macros described on the `ctype(3C)` manual page.

Examples

EXAMPLE 1 Examples of the `getdate()` function.

The following example shows the possible contents of a template:

EXAMPLE 1 Examples of the `getdate()` function. *(Continued)*

```
%m
%A %B %d %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p,%B %dnd
%A den %d. %B %Y %H.%M Uhr
```

The following are examples of valid input specifications for the above template:

```
getdate("10/1/87 4 PM")
getdate("Friday")
getdate("Friday September 19 1987, 10:30:30")
getdate("24,9,1986 10:30")
getdate("at monday the 1st of december in 1986")
getdate("run job at 3 PM, december 2nd")
```

If the `LANG` environment variable is set to `de` (German), the following is valid:

```
getdate("freitag den 10. oktober 1986 10.30 Uhr")
```

Local time and date specification are also supported. The following examples show how local date and time specification can be defined in the template.

Invocation	Line in Template
getdate("11/27/86")	%m/%d/%y
getdate("27.11.86")	%d.%m.%y
getdate("86-11-27")	%y-%m-%d
getdate("Friday 12:00:00")	%A %H:%M:%S

The following examples illustrate the Internal Format Conversion rules. Assume that the current date is Mon Sep 22 12:19:47 EDT 1986 and the `LANG` environment variable is not set.

Input	Template Line	Date
`Mon`	`%a`	Mon Sep 22 12:19:48 EDT 1986
`Sun`	`%a`	Sun Sep 28 12:19:49 EDT 1986
`Fri`	`%a`	Fri Sep 26 12:19:49 EDT 1986

`September`	`%B`	Mon Sep 1 12:19:49 EDT 1986
`January`	`%B`	Thu Jan 1 12:19:49 EST 1987
`December`	`%B`	Mon Dec 1 12:19:49 EDT 1986
`Sep Mon`	`%b %a`	Mon Sep 1 12:19:50 EDT 1986
`Jan Fri`	`%b %a`	Fri Jan 2 12:19:50 EST 1987
`Dec Mon`	`%b %a`	Mon Dec 1 12:19:50 EST 1986
`Jan Wed 1989`	`%b %a %Y`	Wed Jan 4 12:19:51 EST 1989
`Fri 9`	`%a %H`	Fri Sep 26 09:00:00 EDT 1986
`Feb 10:30`	`%b %H:%S`	Sun Feb 1 10:00:30 EST 1987
`10:30`	`%H:%M`	Tue Sep 23 10:30:00 EDT 1986
`13:30`	`%H:%M`	Mon Sep 22 13:30:00 EDT 1986

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also

`ctype(3C)`, `mktime(3C)`, `setlocale(3C)`, `strftime(3C)`, `strptime(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Name getdtablesize – get the file descriptor table size

Synopsis

```
#include <unistd.h>

int getdtablesize(void);
```

Description The `getdtablesize()` function is equivalent to `getrlimit(2)` with the `RLIMIT_NOFILE` option.

Return Values The `getdtablesize()` function returns the current soft limit as if obtained from a call to `getrlimit()` with the `RLIMIT_NOFILE` option.

Errors No errors are defined.

Usage There is no direct relationship between the value returned by `getdtablesize()` and `OPEN_MAX` defined in <`limits.h`>.

Each process has a file descriptor table which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The `getdtablesize()` function returns the current maximum size of this table by calling the `getrlimit()` function.

See Also `close(2)`, `getrlimit(2)`, `open(2)`, `setrlimit(2)`, `select(3C)`

Name getenv – return value for environment name

Synopsis

```
#include <stdlib.h>

char *getenv(const char *name);
```

Description The `getenv()` function searches the environment list (see `environ(5)`) for a string of the form *name*=*value* and, if the string is present, returns a pointer to the *value* in the current environment.

Return Values If successful, `getenv()` returns a pointer to the *value* in the current environment; otherwise, it returns a null pointer.

Usage The `getenv()` function can be safely called from a multithreaded application. Care must be exercised when using both `getenv()` and `putenv(3C)` in a multithreaded application. These functions examine and modify the environment list, which is shared by all threads in an application. The system prevents the list from being accessed simultaneously by two different threads. It does not, however, prevent two threads from successively accessing the environment list using `getenv()` or `putenv(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `exec(2)`, `putenv(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Name getexecname – return pathname of executable

Synopsis

```
#include <stdlib.h>

const char *getexecname(void);
```

Description The `getexecname()` function returns the pathname (the first argument of one of the `exec` family of functions; see `exec(2)`) of the executable that started the process.

Normally this is an absolute pathname, as the majority of commands are executed by the shells that append the command name to the user's `PATH` components. If this is not an absolute path, the output of `getcwd(3C)` can be prepended to it to create an absolute path, unless the process or one of its ancestors has changed its root directory or current working directory since the last successful call to one of the `exec` family of functions.

Return Values If successful, `getexecname()` returns a pointer to the executables pathname; otherwise, it returns `0`.

Usage The `getexecname()` function obtains the executable pathname from the `AT_SUN_EXECNAME` aux vector. These vectors are made available to dynamically linked processes only.

A successful call to one of the `exec` family of functions will always have `AT_SUN_EXECNAME` in the aux vector. The associated pathname is guaranteed to be less than or equal to `PATH_MAX`, not counting the trailing null byte that is always present.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `exec(2)`, `getcwd(3C)`, `attributes(5)`

Name getgrnam, getgrnam_r, getgrent, getgrent_r, getgrgid, getgrgid_r, setgrent, endgrent, fgetgrent, fgetgrent_r – group database entry functions

Synopsis

```
#include <grp.h>

struct group *getgrnam(const char *name);

struct group *getgrnam_r(const char *name, struct group *grp,
     char *buffer, int bufsize);

struct group *getgrent(void);

struct group *getgrent_r(struct group *grp, char *buffer, int bufsize);

struct group *getgrgid(gid_t gid);

struct group *getgrgid_r(gid_t gid, struct group *grp, char *buffer,
     int bufsize);

void setgrent(void);

void endgrent(void);

struct group *fgetgrent(FILE *f);

struct group *fgetgrent_r(FILE *f, struct group *grp, char *buffer,
     int bufsize);
```

Standard conforming

```
cc [ flag... ] file... -D_POSIX_PTHREAD_SEMANTICS [ library... ]

int getgrnam_r(const char *name, struct group *grp, char *buffer,
     size_t bufsize, struct group **result);

int getgrgid_r(gid_t gid, struct group *grp, char *buffer,
     size_t bufsize, struct group **result);
```

Description These functions are used to obtain entries describing user groups. Entries can come from any of the sources for `group` specified in the `/etc/nsswitch.conf` file (see `nsswitch.conf(4)`).

The `getgrnam()` function searches the group database for an entry with the group name specified by the character string parameter *name*.

The `getgrgid()` function searches the group database for an entry with the (numeric) group id specified by *gid*.

The `setgrent()`, `getgrent()`, and `endgrent()` functions are used to enumerate group entries from the database.

The `setgrent()` function effectively rewinds the group database to allow repeated searches. It sets (or resets) the enumeration to the beginning of the set of group entries. This function should be called before the first call to `getgrent()`.

The `getgrent()` function returns a pointer to a structure containing the broken-out fields of an entry in the group database. When first called, `getgrent()` returns a pointer to a `group` structure containing the next group structure in the group database. Successive calls can be used to search the entire database.

The `endgrent()` function can be called to close the group database and deallocate resources when processing is complete. It is permissible, though possibly less efficient, for the process to call more group functions after calling `endgrent()`.

The `fgetgrent()` function, unlike the other functions above, does not use `nsswitch.conf`. It reads and parses the next line from the stream *f*, which is assumed to have the format of the `group` file (see `group(4)`).

Reentrant Interfaces

The `getgrnam()`, `getgrgid()`, `getgrent()`, and `fgetgrent()` functions use thread-specific storage that is reused in each call to one of these functions by the same thread, making them safe to use but not recommended for multithreaded applications.

The parallel functions `getgrnam_r()`, `getgrgid_r()`, `getgrent_r()`, and `fgetgrent_r()` provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the `_r` suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results instead of using thread-specific data that can be overwritten by each call. They are safe for use in both single-threaded and multithreaded applications.

Each reentrant interface takes the same arguments as its non-reentrant counterpart, as well as the following additional parameters. The *grp* argument must be a pointer to a `struct group` structure allocated by the caller. On successful completion, the function returns the group entry in this structure. Storage referenced by the group structure is allocated from the memory provided with the *buffer* argument that is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the `_SC_GETGR_R_SIZE_MAX` `sysconf(3C)` parameter. The standard-conforming versions place a pointer to the modified *grp* structure in the *result* parameter, instead of returning a pointer to this structure. A null pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. The `setgrent()` function can be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to `getgrent_r()`, the threads will enumerate disjoint subsets of the group database. Like their non-reentrant counterparts, `getgrnam_r()` and `getgrgid_r()` leave the enumeration position in an indeterminate state.

group Structure

Group entries are represented by the `struct group` structure defined in <grp.h>:

```
struct group {
    char *gr_name;          /* the name of the group */
    char *gr_passwd;        /* the encrypted group password */
```

```
    gid_t gr_gid;           /* the numerical group ID */
    char **gr_mem;          /* vector of pointers to member
                               names */
};
```

Return Values The `getgrnam()`, `getgrnam_r()`, `getgrgid()`, and `getgrgid_r()` functions each return a pointer to a `struct group` if they successfully locate the requested entry. They return a null pointer if either the requested entry was not found or an error occurred. On error, `errno` is set to indicate the error. The standard-conforming functions `getgrnam_r()` and `getgrgid_r()` return `0` upon success or an error number in case of failure.

The `getgrent()`, `getgrent_r()`, `fgetgrent()`, and `fgetgrent_r()` functions each return a pointer to a `struct group` if they successfully enumerate an entry; otherwise they return a null pointer on end-of-file or error. On error, `errno` is set to indicate the error.

The `getgrnam()`, `getgrgid()`, `getgrent()`, and `fgetgrent()` functions use thread–specific data storage, so returned data must be copied before a subsequent call to any of these functions if the data are to be saved.

When the pointer returned by the reentrant functions `getgrnam_r()`, `getgrgid_r()`, `getgrent_r()`, and `fgetgrent_r()` is non-null, it is always equal to the *grp* pointer that was supplied by the caller.

Applications wishing to check for error situations should set `errno` to 0 before calling `getgrnam()`, `getgrnam_r()`, `getgrent()`, `getgrent_r()getgrgid()`, `getgrgid_r()`, `fgetgrent()`, and `fgetgrent_r()`. If these functions return a null pointer and `errno` is non-zero, an error occurred.

Errors The `getgrent_r()`, `fgetgrent()`, and `fgetgrent_r()` functions will fail if:

`EIO` An I/O error has occurred.

`ERANGE` Insufficient storage was supplied by *buffer* and *bufsize* to contain the data to be referenced by the resulting `group` structure.

The `getgrent_r()` function will fail if:

`EMFILE` There are {`OPEN_MAX`} file descriptors currently open in the calling process.

`ENFILE` The maximum allowable number of files is currently open in the system.

The `getgrnam()`, `getgrnam_r()`, `getgrgid()`, `getgrgid_r()`, and `getgrent()` functions may fail if:

`EINTR` A signal was caught during the operation.

`EIO` An I/O error has occurred.

`EMFILE` There are {`OPEN_MAX`} file descriptors currently open in the calling process.

ENFILE The maximum allowable number of files is currently open in the system.

The `getgrnam_r()` and `getgrgid_r()` functions may fail if:

ERANGE Insufficient storage was supplied by *buffer* and *bufsize* to contain the data to be referenced by the resulting `group` structure.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	See `Reentrant Interfaces` in `DESCRIPTION`.

The `endgrent()`, `getgrent()`, `getgrgid()`, `getgrgid_r()`, `getgrnam()`, `getgrnam_r()`, and `setgrent()` functions are Standard.

See Also `Intro(3)`, `getpwnam(3C)`, `group(4)`, `nsswitch.conf(4)`, `passwd(4)`, `attributes(5)`, `standards(5)`

Notes When compiling multithreaded programs, see `Intro(3)`.

Use of the enumeration interfaces `getgrent()` and `getgrent_r()` is discouraged; enumeration is supported for the group file and NIS, but in general is not efficient and might not be supported for all database sources. The semantics of enumeration are discussed further in `nsswitch.conf(4)`.

Previous releases allowed the use of "+" and "-" entries in `/etc/group` to selectively include and exclude entries from NIS. The primary usage of these entries is superseded by the name service switch, so the "+/-" form might not be supported in future releases.

If required, the "+/-" functionality can still be obtained for NIS by specifying `compat` as the source for `group`.

Solaris 2.4 and earlier releases provided definitions of the `getgrnam_r()` and `getgrgid_r()` functions as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface for these functions. Support for the Draft 6 interface is provided for compatibility only and might not be supported in future releases. New applications and libraries should use the standard-conforming interface.

For POSIX.1c-conforming applications, the `_POSIX_PTHREAD_SEMANTICS` and `_REENTRANT` flags are automatically turned on by defining the `_POSIX_C_SOURCE` flag with a value ≥199506L.

Name gethostid – get an identifier for the current host

Synopsis

```
#include <unistd.h>

long gethostid(void);
```

Description The `gethostid()` function returns the 32-bit identifier for the current host. If the hardware capability exists, this identifier is taken from platform-dependent stable storage; otherwise it is a randomly generated number. It is not guaranteed to be unique.

If the calling thread's process is executing within a non-global zone that emulates a host identifier, then the zone's emulated 32-bit host identifier is returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `hostid(1)`, `sysinfo(2)`, `attributes(5)`, `standards(5)`, `zones(5)`

Name gethostname, sethostname – get or set name of current host

Synopsis

```
#include <unistd.h>

int gethostname(char *name, int namelen);

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

Description The `gethostname()` function returns the standard host name for the current processor, as previously set by `sethostname()`. The *namelen* argument specifies the size of the array pointed to by *name*. The returned name is null-terminated unless insufficient space is provided.

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

Host names are limited to `MAXHOSTNAMELEN` characters, currently 256, defined in the <`netdb.h`> header.

Return Values Upon successful completion, `gethostname()` and `sethostname()` return 0. Otherwise, they return −1 and set `errno` to indicate the error.

Errors The `gethostname()` and `sethostname()` functions will fail if:

`EFAULT` The *name* or *namelen* argument gave an invalid address.

The `sethostname()` function will fail if:

`EPERM` The {`PRIV_SYS_ADMIN`} privilege was not asserted in the effective set of the calling process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `sysinfo(2)`, `uname(2)`, `gethostid(3C)`, `attributes(5)`, `standards(5)`

Name gethrtime, gethrvtime – get high resolution time

Synopsis

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

hrtime_t gethrtime(void);

hrtime_t gethrvtime(void);
```

Description The `gethrtime()` function returns the current high-resolution real time. Time is expressed as nanoseconds since some arbitrary time in the past; it is not correlated in any way to the time of day, and thus is not subject to resetting or drifting by way of `adjtime(2)` or `settimeofday(3C)`. The hi-res timer is ideally suited to performance measurement tasks, where cheap, accurate interval timing is required.

The `gethrvtime()` function returns the current high-resolution LWP virtual time, expressed as total nanoseconds of execution time.

The `gethrtime()` and `gethrvtime()` functions both return an `hrtime_t`, which is a 64-bit (`long long`) signed integer.

Examples The following code fragment measures the average cost of `getpid(2)`:

```
hrtime_t start, end;
int i, iters = 100;

start = gethrtime();
for (i = 0; i < iters; i++)
        getpid();
end = gethrtime();

printf("Avg getpid() time = %lld nsec\n", (end – start) / iters);
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `proc(1)`, `adjtime(2)`, `gettimeofday(3C)`, `settimeofday(3C)`, `attributes(5)`

Notes Although the units of hi-res time are always the same (nanoseconds), the actual resolution is hardware dependent. Hi-res time is guaranteed to be monotonic (it won't go backward, it won't periodically wrap) and linear (it won't occasionally speed up or slow down for adjustment, like the time of day can), but not necessarily unique: two sufficiently proximate calls may return the same value.

Name getline, getdelim – delimited string input

Synopsis

```
#include <stdio.h>

ssize_t getline(char **restrict lineptr, size_t *restrict n,
FILE *restrict stream);

ssize_t getdelim(char **restrict lineptr, size_t *restrict n,
int delimiter, FILE *restrict stream);
```

Description The `getline()` function reads an entire line from *stream*, storing the address of the buffer containing the line in **lineptr*. The buffer is null-terminated and includes the NEWLINE character if one was found.

If **lineptr* is a null pointer, `getline()` allocates a buffer for storing the line. Alternatively, before the call to `getline()`, **lineptr* can contain a pointer to a buffer allocated by `malloc(3C)` whose size is **n* bytes. If the buffer is not large enough to store the line, `getline()` resizes the buffer with `realloc(3C)`. In either case, a successful call to `getline()` updates **lineptr* and **n* to reflect the buffer address and size, respectively. The buffer should be freed with a call to `free(3C)`.

The `getdelim()` function is identical to `getline()`, except a line delimiter other than NEWLINE can be specified as the *delimiter* argument. As with `getline()`, a delimiter character is not added if one was not present in *stream* before end-of-file was reached.

Return Values Upon successful completion, the `getline()` and `getdelim()` functions return the number of characters written into the buffer, including the delimiter character but excluding the terminating null character. Upon failure to read a line (including end of file condition), these function return −1 and set *errno* to indicate the error.

Errors The `getline()` and `getdelim()` functions will fail if:

`EINVAL` Either *lineptr* or *n* is a null pointer.

`ENOMEM` Insufficient memory is available.

The `getline()` and `getdelim()` functions may fail if:

`EOVERFLOW` More than {`SSIZE_MAX`} characters were read without encountering the delimiter character.

See `fgetc(3C)` for other conditions under which these functions will and may fail.

Examples **EXAMPLE 1** Retrieve a line length.

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

int main(void)
{
```

EXAMPLE 1 Retrieve a line length. *(Continued)*

```
    FILE *fp;
    char *line = NULL;
    size_t len = 0;
    ssize_t read;
    fp = fopen("/etc/motd", "r");
    if (fp == NULL)
        exit(1);
    while ((read = getline(&line, &len, fp)) != -1) {
        printf("Retrieved line of length %zu :\
", read);
        printf("%s", line);
    }
    if (ferror(fp)) {
        /* handle error */
    }
    free(line);
    fclose(fp);
    return 0;
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

fgetc(3C), fgets(3C), free(3C), malloc(3C), realloc(3C), attributes(5)

Name getloadavg – get system load averages

Synopsis

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

int getloadavg(double loadavg[ ], int nelem);
```

Description The `getloadavg()` function returns the number of processes in the system run queue averaged over various periods of time. Up to *nelem* samples are retrieved and assigned to successive elements of *loadavg*[ ]. The system imposes a maximum of 3 samples, representing averages over the last 1, 5, and 15 minutes, respectively. The `LOADAVG_1MIN`, `LOADAVG_5MIN`, and `LOADAVG_15MIN` indices, defined in `<sys/loadavg.h>`, can be used to extract the data from the appropriate element of the *loadavg*`[ ]` array.

Return Values Upon successful completion, the number of samples actually retrieved is returned. If the load average was unobtainable, −1 is returned and `errno` is set to indicate the error.

Errors The `getloadavg()` function will fail if:

`EINVAL` The number of elements specified is less than 0.

Usage If the caller is in a non-global zone and the pools facility is active, the behavior of `getloadavg()` is equivalent to that of `pset_getloadavg(3C)` called with *psetid* set to `PS_MYID`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also `uptime(1)`, `w(1)`, `pooladm(1M)`, `Kstat(3PERL)`, `pset_getloadavg(3C)`, `standards(5)`

Name getlogin, getlogin_r – get login name

Synopsis

```
#include <unistd.h>

char *getlogin(void);

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

Standard conforming

```
cc [ flag ... ] file... -D_POSIX_PTHREAD_SEMANTICS [ library ... ]

int getlogin_r(char *name, size_t namesize);
```

Description The `getlogin()` function returns a pointer to the login name as found in `/var/adm/utmpx`. It can be used in conjunction with `getpwnam(3C)` to locate the correct password file entry when the same user ID is shared by several login names.

If `getlogin()` is called within a process that is not attached to a terminal, it returns a null pointer. The correct procedure for determining the login name is to call `cuserid(3C)`, or to call `getlogin()` and if it fails to call `getpwuid(3C)`.

The `getlogin_r()` function has the same functionality as `getlogin()` except that the caller must supply a buffer *name* with length *namelen* to store the result. The *name* buffer must be at least `_POSIX_LOGIN_NAME_MAX` bytes in size (defined in <`limits.h`>). The POSIX version (see `standards(5)`) of `getlogin_r()` takes a *namesize* parameter of type `size_t`.

Return Values Upon successful completion, `getlogin()` returns a pointer to the login name or a null pointer if the user's login name cannot be found. Otherwise it returns a null pointer and sets `errno` to indicate the error.

The standard-conforming `getlogin_r()` returns `0` if successful, or the error number upon failure.

Errors The `getlogin_r()` function will fail if:

`ERANGE` The size of the buffer is smaller than the result to be returned.

`EINVAL` And entry for the current user was not found in the `/var/adm/utmpx` file.

The `getlogin()` and `getlogin_r()` functions may fail if:

`EMFILE` There are {`OPEN_MAX`} file descriptors currently open in the calling process.

`ENFILE` The maximum allowable number of files is currently open in the system.

`ENXIO` The calling process has no controlling terminal.

The `getlogin_r()` function may fail if:

`ERANGE` The size of the buffer is smaller than the result to be returned.

Usage The return value of `getlogin()` points to thread-specific data whose content is overwritten on each call by the same thread.

Three names associated with the current process can be determined: `getpwuid(geteuid())` returns the name associated with the effective user ID of the process; `getlogin()` returns the name associated with the current login activity; and `getpwuid(getuid())` returns the name associated with the real user ID of the process.

Files `/var/adm/utmpx` user access and administration information

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See below.
Standard	See `standards(5)`.

See Also `geteuid(2)`, `getuid(2)`, `cuserid(3C)`, `getgrnam(3C)`, `getpwnam(3C)`, `getpwuid(3C)`, `utmpx(4)`, `attributes(5)`, `standards(5)`

Notes When compiling multithreaded programs, see `Intro(3)`.

The `getlogin()` function is safe to use in multithreaded applications, but is discouraged. The `getlogin_r()` function should be used instead.

Solaris 2.4 and earlier releases provided a `getlogin_r()` as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface as described above. Support for the Draft 6 interface is provided for compatibility only and may not be supported in future releases. New applications and libraries should use the standard-conforming interface.

Name getmntent, getmntany, getextmntent, hasmntopt, putmntent, resetmnttab – get mounted device information

Synopsis

```
#include <stdio.h>
#include <sys/mnttab.h>

int getmntent(FILE *fp, struct mnttab *mp);

int getmntany(FILE *fp, struct mnttab *mp, struct mnttab *mpref);

int getextmntent(FILE *fp, struct extmnttab *mp, int len);

char *hasmntopt(struct mnttab *mnt, char *opt);

int putmntent(FILE *iop, struct mnttab *mp);

void resetmnttab(FILE *fp);
```

Description

getmntent() and getmntany()

The `getmntent()` and `getmntany()` functions each fill in the structure pointed to by *mp* with the broken-out fields of a line in the `mnttab` file. Each line read from the file contains a `mnttab` structure, which is defined in the <`sys/mnttab.h`> header. The structure contains the following members, which correspond to the broken-out fields from a line in `/etc/mnttab` (see `mnttab(4)`).

```
char   *mnt_special;     /* name of mounted resource */
char   *mnt_mountp;      /* mount point */
char   *mnt_fstype;      /* type of file system mounted */
char   *mnt_mntopts;     /* options for this mount */
char   *mnt_time;        /* time file system mounted */
```

Fields with no actual content in `/etc/mnttab` are represented in the file as "-". To clearly distinguish empty fields, `getmntent()` set the corresponding field in *mp* to `NULL`.

Each `getmntent()` call causes a new line to be read from the `mnttab` file. Successive calls can be used to search the entire list. The `getmntany()` function searches the file referenced by *fp* until a match is found between a line in the file and *mpref*. A match occurs if all non-null entries in *mpref* match the corresponding fields in the file. These functions do not open, close, or rewind the file.

getextmntent()

The `getextmntent()` function is an extended version of the `getmntent()` function that returns, in addition to the information that `getmntent()` returns, the major and minor number of the mounted resource to which the line in `mnttab` corresponds. The `getextmntent()` function also fills in the `extmntent` structure defined in the <`sys/mnttab.h`> header. For `getextmntent()` to function properly, it must be notified when the `mnttab` file has been reopened or rewound since a previous `getextmntent()` call. This notification is accomplished by calling `resetmnttab()`. Otherwise, it behaves exactly as `getmntent()` described above

The data pointed to by the `mnttab` structure members are stored in a static area and must be copied to be saved between successive calls.

`hasmntopt()` The `hasmntopt()` function scans the `mnt_mntopts` member of the `mnttab` structure *mnt* for a substring that matches *opt*. It returns the address of the substring if a match is found; otherwise it returns `0`. Substrings are delimited by commas and the end of the `mnt_mntopts` string.

`putmntent()` The `putmntent()` function is obsolete and no longer has any effect. Entries appear in `mnttab` as a side effect of a `mount(2)` call. The function name is still defined for transition purposes.

`resetmnttab()` The `resetmnttab()` function notifies `getextmntent()` to reload from the kernel the device information that corresponds to the new snapshot of the `mnttab` information (see `mnttab(4)`). Subsequent `getextmntent()` calls then return correct `extmnttab` information. This function should be called whenever the `mnttab` file is either rewound or closed and reopened before any calls are made to `getextmntent()`.

Return Values

`getmntent()` and `getmntany()` If the next entry is successfully read by `getmntent()` or a match is found with `getmntany()`, `0` is returned. If an EOF is encountered on reading, these functions return `−1`. If an error is encountered, a value greater than 0 is returned. The following error values are defined in `<sys/mnttab.h>`:

`MNT_TOOLONG` A line in the file exceeded the internal buffer size of `MNT_LINE_MAX`.

`MNT_TOOMANY` A line in the file contains too many fields.

`MNT_TOOFEW` A line in the file contains too few fields.

`hasmntopt()` Upon successful completion, `hasmntopt()` returns the address of the substring if a match is found. Otherwise, it returns `0`.

`putmntent()` The `putmntent()` is obsolete and always returns `−1`.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also

`mount(2)`, `mnttab(4)`, `attributes(5)`

Name getnetgrent, getnetgrent_r, setnetgrent, endnetgrent, innetgr – get network group entry

Synopsis

```
#include <netdb.h>

int getnetgrent(char **machinep, char **userp, char **domainp);

int getnetgrent_r(char **machinep, char **userp, char **domainp,
     char *buffer, intbuflen);

int setnetgrent(const char *netgroup);

int endnetgrent(void);

int innetgr(const char *netgroup, const char *machine,
     const char *user, const char *domain);
```

Description These functions are used to test membership in and enumerate members of "netgroup" network groups defined in a system database. Netgroups are sets of (machine,user,domain) triples (see `netgroup(4)`).

These functions consult the source specified for `netgroup` in the `/etc/nsswitch.conf` file (see `nsswitch.conf(4)`).

The function `innetgr()` returns `1` if there is a netgroup *netgroup* that contains the specified *machine, user, domain* triple as a member; otherwise it returns `0`. Any of the supplied pointers *machine*, *user*, and *domain* may be `NULL`, signifying a "wild card" that matches all values in that position of the triple.

The `innetgr()` function is safe for use in single-threaded and multithreaded applications.

The functions `setnetgrent()`, `getnetgrent()`, and `endnetgrent()` are used to enumerate the members of a given network group.

The function `setnetgrent()` establishes the network group specified in the parameter *netgroup* as the current group whose members are to be enumerated.

Successive calls to the function `getnetgrent()` will enumerate the members of the group established by calling `setnetgrent()`; each call returns `1` if it succeeds in obtaining another member of the network group, or `0` if there are no further members of the group.

When calling either `getnetgrent()` or `getnetgrent_r()`, addresses of the three character pointers are used as arguments, for example:

```
char *mp, *up, *dp;
getnetgrent(&mp, &up, &dp);
```

Upon successful return from `getnetgrent()`, the pointer *mp* points to a string containing the name of the machine part of the member triple, *up* points to a string containing the user name and *dp* points to a string containing the domain name. If the pointer returned for *mp*, *up*, or *dp* is `NULL`, it signifies that the element of the netgroup contains wild card specifier in that position of the triple.

The pointers returned by `getnetgrent()` point into a buffer allocated by `setnetgrent()` that is reused by each call. This space is released when an `endnetgrent()` call is made, and should not be released by the caller. This implementation is not safe for use in multi-threaded applications.

The function `getnetgrent_r()` is similar to `getnetgrent()` function, but it uses a buffer supplied by the caller for the space needed to store the results. The parameter *buffer* should be a pointer to a buffer allocated by the caller and the length of this buffer should be specified by the parameter *buflen*. The buffer must be large enough to hold the data associated with the triple. The `getnetgrent_r()` function is safe for use both in single-threaded and multi-threaded applications.

The function `endnetgrent()` frees the space allocated by the previous `setnetgrent()` call. The equivalent of an `endnetgrent()` implicitly performed whenever a `setnetgrent()` call is made to a new network group.

Note that while `setnetgrent()` and `endnetgrent()` are safe for use in multi-threaded applications, the effect of each is process-wide. Calling `setnetgrent()` resets the enumeration position for all threads. If multiple threads interleave calls to `getnetgrent_r()` each will enumerate a disjoint subset of the netgroup. Thus the effective use of these functions in multi-threaded applications may require coordination by the caller.

Errors The function `getnetgrent_r()` will return `0` and set `errno` to `ERANGE` if the length of the buffer supplied by caller is not large enough to store the result. See `Intro(2)` for the proper usage and interpretation of `errno` in multi-threaded applications.

The functions `setnetgrent()` and `endnetgrent()` return `0` upon success.

Files `/etc/nsswitch.conf`

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	See `DESCRIPTION` section.

See Also `Intro(2)`, `Intro(3)`, `netgroup(4)`, `nsswitch.conf(4)`, `attributes(5)`

Warnings The function `getnetgrent_r()` is included in this release on an uncommitted basis only, and is subject to change or removal in future minor releases.

Notes Only the Network Information Services, NIS and NIS+, are supported as sources for the `netgroup` database.

When compiling multi-threaded applications, see `Intro(3)`, *Notes On Multithread Applications*, for information about the use of the `_REENTRANT` flag.

Name get_nprocs, get_nprocs_conf – get number of processors

Synopsis

```
#include <unistd.h>

int get_nprocs(void);

int get_nprocs_conf(void);
```

Description The `get_nprocs()` and `get_nprocs_conf()` functions are, respectively, equivalent to:

```
sysconf(_SC_NPROCESSORS_ONLN);

sysconf(_SC_NPROCESSORS_CONF);
```

Return Values The `get_nprocs()` function returns the number of available processors. The `get_nprocs_conf()` function returns the number of processors the operating system configured.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `sysconf(3C)`, `attributes(5)`

Notes The `get_nprocs()` and `get_nprocs_conf()` functions are provided only as GNU/Linux compatibility interfaces.

Name getopt – command option parsing

Synopsis

SVID3, XPG3

```
#include <stdio.h>

int getopt(int argc, char * const argv[], const char *optstring);

extern char *optarg;

extern int optind, opterr, optopt;
```

POSIX.2, XPG4, SUS, SUSv2, SUSv3

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *optstring);

extern char *optarg;

extern int optind, opterr, optopt;
```

Description The `getopt()` function is a command line parser that can be used by applications that follow Basic Utility Syntax Guidelines 3, 4, 5, 6, 7, 9, and 10 which parallel those defined by application portability standards (see intro(1)). It can also be used by applications which additionally follow the Command Line Interface Paradigm (CLIP) syntax extension guidelines 15, 16, and 17. It partially enforces guideline 18 by requiring that every option has a short-name, but it allows multiple long-names to be associated with an option. The remaining guidelines are not addressed by `getopt()` and are the responsibility of the application.

The *argc* and *argv* arguments are the argument count and argument array as passed to main (see exec(2)). The *optstring* argument specifies the acceptable options. For utilities wanting to conform to the Basic Utility Syntax Guidelines, *optstring* is a string of recognized option characters. All option characters allowed by Utility Syntax Guideline 3 are allowed in *optstring*. If a character is followed by a colon (:), the option is expected to have an option-argument, which can be separated from it by white space. Utilities wanting to conform to the extended CLIP guidelines can specify long-option equivalents to short options by following the short-option character (and optional colon) with a sequence of strings, each enclosed in parentheses, that specify the long-option aliases.

The `getopt()` function returns the short-option character in *optstring* that corresponds to the next option found in *argv*.

The `getopt()` function places in *optind* the *argv* index of the next argument to be processed. The *optind* variable is external and is initialized to 1 before the first call to `getopt()`. The `getopt()` function sets the variable *optarg* to point to the start of the option-argument as follows:

- If the option is a short option and that character is the last character in the argument, then *optarg* contains the next element of *argv*, and *optind* is incremented by 2.

- If the option is a short option and that character is not the last character in the argument, then *optarg* points to the string following the option character in that argument, and *optind* is incremented by 1.
- If the option is a long option and the character equals is not found in the argument, then *optarg* contains the next element of *argv*, and *optind* is incremented by 2.
- If the option is a long option and the character equals is found in the argument, then *optarg* points to the string following the equals character in that argument and *optind* is incremented by 1.

In all cases, if the resulting value of *optind* is not less than *argc*, this indicates a missing option-argument and `getopt()` returns an error indication.

When all options have been processed (that is, up to the first operand), `getopt()` returns -1. The special option "--"(two hyphens) can be used to delimit the end of the options; when it is encountered, -1 is returned and "--" is skipped. This is useful in delimiting non-option arguments that begin with "-" (hyphen).

If `getopt()` encounters a short-option character or a long-option string not described in the *opstring* argument, it returns the question-mark (?) character. If it detects a missing option-argument, it also returns the question-mark (?) character, unless the first character of the *optstring* argument was a colon (:), in which case `getopt()` returns the colon (:) character. For short options, `getopt()` sets the variable *optopt* to the option character that caused the error. For long options, *optopt* is set to the hyphen (-) character and the failing long option can be identified through *argv*[*optind*-1]. If the application has not set the variable *opterr* to 0 and the first character of *optstring* is not a colon (:), `getopt()` also prints a diagnostic message to `stderr`.

Return Values The `getopt()` function returns the short-option character associated with the option recognized.

A colon (:) is returned if `getopt()` detects a missing argument and the first character of *optstring* was a colon (:).

A question mark (?) is returned if `getopt()` encounters an option not specified in *optstring* or detects a missing argument and the first character of *optstring* was not a colon (:).

Otherwise, `getopt()` returns -1 when all command line options are parsed.

Errors No errors are defined.

Examples **EXAMPLE 1** Parsing Command Line Options

The following code fragment shows how you might process the arguments for a utility that can take the mutually-exclusive options `a` and `b` and the options `f` and `o`, both of which require arguments:

EXAMPLE 1 Parsing Command Line Options *(Continued)*

```
#include <unistd.h>

int
main(int argc, char *argv[ ])
{
    int c;
    int bflg, aflg, errflg;
    char *ifile;
    char *ofile;
    extern char *optarg;
    extern int optind, optopt;
    . . .
    while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
        switch(c) {
        case 'a':
            if (bflg)
                errflg++;
            else
                aflg++;
            break;
        case 'b':
            if (aflg)
                errflg++;
            else {
                bflg++;
                bproc();
            }
            break;
        case 'f':
            ifile = optarg;
            break;
        case 'o':
            ofile = optarg;
            break;
        case ':':   /* -f or -o without operand */
            fprintf(stderr,
                    "Option -%c requires an operand\n", optopt);
            errflg++;
            break;
        case '?':
            fprintf(stderr,
                    "Unrecognized option: -%c\n", optopt);
            errflg++;
        }
    }
    if (errflg) {
```

EXAMPLE 1 Parsing Command Line Options *(Continued)*

```
        fprintf(stderr, "usage: . . . ");
        exit(2);
    }
    for ( ; optind < argc; optind++) {
        if (access(argv[optind], R_OK)) {
    . . .
}
```

This code accepts any of the following as equivalent:

```
cmd -ao arg path path
cmd -a -o arg path path
cmd -o arg -a path path
cmd -a -o arg -- path path
cmd -a -oarg path path
cmd -aoarg path path
```

EXAMPLE 2 Check Options and Arguments.

The following example parses a set of command line options and prints messages to standard output for each option and argument that it encounters.

```
#include <unistd.h>
#include <stdio.h>
...
int c;
char *filename;
extern char *optarg;
extern int optind, optopt, opterr;
...
while ((c = getopt(argc, argv, ":abf:")) != -1) {
    switch(c) {
    case 'a':
         printf("a is set\n");
         break;
    case 'b':
         printf("b is set\n");
         break;
    case 'f':
         filename = optarg;
         printf("filename is %s\n", filename);
         break;
    case ':':
         printf("-%c without filename\n", optopt);
         break;
    case '?':
         printf("unknown arg %c\n", optopt);
```

EXAMPLE 2 Check Options and Arguments. *(Continued)*

```
        break;
    }
}
```

This example can be expanded to be CLIP-compliant by substituting the following string for the *optstring* argument:

```
:a(ascii)b(binary)f:(in-file)o:(out-file)V(version)?(help)
```

and by replacing the '?' case processing with:

```
case 'V':
    fprintf(stdout, "cmd 1.1\n");
    exit(0);
case '?':
    if (optopt == '?') {
        print_help();
        exit(0);
    }
    if (optopt == '-')
        fprintf(stderr,
            "unrecognized option: %s\n", argv[optind-1]);
    else
        fprintf(stderr,
            "unrecognized option: -%c\n", optopt);
    errflg++;
    break;
```

and by replacing the ':' case processing with:

```
case ':':   /* -f or -o without operand */
    if (optopt == '-')
        fprintf(stderr,
            "Option %s requires an operand\n", argv[optind-1]);
    else
        fprintf(stderr,
            "Option -%c requires an operand\n", optopt);
    errflg++;
    break;
```

While not encouraged by the CLIP specification, multiple long-option aliases can also be assigned as shown in the following example:

```
:a(ascii)b(binary):(in-file)(input)o:(outfile)(output)V(version)?(help)
```

Environment Variables

See `environ(5)` for descriptions of the following environment variables that affect the execution of `getopt()`: `LANG`, `LC_ALL`, and `LC_MESSAGES`.

`LC_CTYPE` Determine the locale for the interpretation of sequences of bytes as characters in *optstring*.

Usage The `getopt()` function does not fully check for mandatory arguments because there is no unambiguous algorithm to do so. Given an option string `a:b` and the input `-a -b`, `getopt()` assumes that `-b` is the mandatory argument to the `-a` option and not that `-a` is missing a mandatory argument. Indeed, the only time a missing option-argument can be reliably detected is when the option is the final option on the command line and is not followed by any command arguments.

It is a violation of the Basic Utility Command syntax standard (see `Intro(1)`) for options with arguments to be grouped with other options, as in `cmd -abo` *filename* , where `a` and `b` are options, `o` is an option that requires an argument, and *filename* is the argument to `o`. Although this syntax is permitted in the current implementation, it should not be used because it may not be supported in future releases. The correct syntax to use is:

```
cmd –ab –o filename
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	See below.

For the Basic Utility Command syntax is Standard, see `standards(5)`.

See Also `Intro(1)`, `getopt(1)`, `getopts(1)`, `getsubopt(3C)`, `gettext(3C)`, `setlocale(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Name getopt_long, getopt_long_only, getopt_clip – parse long command options

Synopsis

```
#include <getopt.h>

int getopt_long(int argc, char * const *argv, const char *shortopts,
     const struct option *longopts, int *indexptr);

int getopt_long_only(int argc, char * const *argv, const char *shortopts,
     const struct option *longopts, int *indexptr);

int getopt_clip(int argc, char * const *argv, const char *shortopts,
     const struct option *longopts, int *indexptrextern char *optarg;
extern int optind, opterr, optopt;
```

Description These functions are provided as a porting aid for GNU/Freeware/OpenBSD utilities. The `getopt_long()` function is intended to be as closely compatible with the GNU and OpenBSD implementations as possible, but since these public implementations differ in some corner cases, it is not possible to be fully compatible with both. The differences are enumerated in the NOTES section.

The `getopt_long()` function is an aid for implementing the GNU command line argument conventions. See the GNU documentation for the details of these conventions (glibc 2.2.3). Note that the GNU conventions are not POSIX-conforming. Most notably, the GNU conventions allow for optional option-arguments and do not enforce that operands must follow options on the command line.

The `getopt_clip()` function provides an interface similar to `getopt_long()` except that it implements the Sun CLIP convention, which is slightly more restrictive than the GNU/Freeware conventions. CLIP is modeled after the GNU/Freeware conventions but removes POSIX violations and syntactic ambiguities (see `Intro(1)`). Specifically, `getopt_clip()` is a command line parser that can be used by applications that follow the Command Line Interface Paradigm or CLIP syntax guidelines 3, 4, 5, 6, 7, 9, 10, 15, and 16. The remaining guidelines are not addressed by `getopt_clip()` and are the responsibility of the application.

The `getopt_long()` function is similar to `getopt(3C)` except that it accepts options in two forms: words and characters, also referred to as long options and short options.

The `getopt_long()` function can be used in two ways. In the first way, every long option understood by the program is mapped to a single character that is usually a corresponding short option. The `option` structure is used only to translate from long options to short options. In the second way, a long option sets a `flag` specified in the `option` structure, or stores a pointer to the command line argument in the address passed to it for options that take arguments. These two methods apply individually to each long option. Both methods can be used in the same application.

The `getopt_long()` function accepts command lines that interleave options and operands. The `getopt_long()` function reorders the elements of the *argv* argument such that when all command line arguments have been processed, all operands follow options (and their

option-arguments) in the *argv* array and *optind* points to the first operand. The order of options relative to other options and operands relative to other operands is maintained. The argument "--" is accepted as a delimiter indicating the end of options. No argument reorder occurs past this delimeter. Argument reordering can not be unambiguously performed in all cases. The `getopt_long()` function depends on a number of internal heuristics to perform the reordering. The *argc* and *argv* arguments are the argument count and argument array as passed to `main()` (see `exec(2)`).

The *shortopts* argument contains the short-option characters recognized by the command using these functions. If a letter is followed by a colon (:), the option is expected to have an option-argument that should be separated from it by white space. If a character is followed by two colons (::), the option takes an optional option-argument. Any text after the option name it is returned in *optarg*; otherwise, *optarg* is set to 0. A whitespace character can never be used to separate an optional option-argument from its associated option. If *shortopts* contains the character "`W`" followed by a semicolon (;), then `-W foo` is treated as the long option `--foo`.

If the first character of the *shortopts* argument is the plus sign (+), `getopt_long()` enforces the POSIX requirement that operands follow options on the command line by returning -1 and stopping argument processing upon encountering the first operand (or "--"). This behavior can also be specified by setting the environment variable `POSIXLY_CORRECT`.

A hyphen (-) as the first character of the *shortopts* argument specifies that options and operands can be intermixed in *argv* but no argument reordering is performed. Operands are returned as arguments to option '\1', and option processing does not stop until "--" or the end of *argv* is found.

If the first character of the *shortopts* argument (after a potential plus or minus character) is a colon (:), a colon is returned by `getopt_long()` in response to a missing argument; otherwise, a question mark (?) is returned for this condition.

The *longopts* argument describes the long options to accept. It is an array of `struct option` structures, one for each long option. The array is terminated with an element containing all zeros.

The `struct option` structure contains the following members:

`const char *name`	Contains a pointer to the name of the option.
`int has_arg`	Specifies whether the option takes an argument. The possible values, defined in <`getopt.h`>, are `no_argument`, `optional_argument`, and `required_argument`.
`int *flag`	Contains the address of an `int` variable that is the flag for this option. The value contained in val is stored in this location to indicate that the option was seen. If `flag` is a null pointer, then the value contained in `val` is returned when this option is encountered, otherwise zero is returned.

`int val`	Contains the value to be stored at the variable pointed to by `flag` or returned by `getopt_long()` if `flag` is a null pointer.

For any long option, `getopt_long()` returns the index in the array *longopts* of the options definition by storing it in *indexptr*. The name of the option can be retrieved with *longopts*[(***indexptr*)].`name`. Long options can be distinguished either by the values in their `val` members or by their indices. The *indexptr* variable can also distinguish long options that set flags. The value of *indexptr* after encountering a short option is undefined.

If an option has an argument, the *optarg* global variable is set to point to the start of the option argument on return from `getopt_long()`; otherwise it is set to null. A long option can take an argument in one of two forms: --option=arg or --option arg. If the long option argument is optional, only the "`--option=arg`" form can be used to specify the option argument. No argument is specified by the simple form "`--option`". The form "`--option=`" specifies an empty string as the option argument.

Long-option names can be abbreviated if the abbreviation is unique or an exact match for some defined option. An exact match takes precedence over an abbreviated match. Thus, if `foo` and `foobar` are acceptable long-option names, then specifying `--foo` on the command line always matches the former. Specifying `--f` or `--fo` would not be accepted as a match for either.

The `getopt_long()` function places in *optind* the *argv* index of the next argument to be processed. The *optind* global variable is external and is initialized to 1 before the first call to `getopt_long()`. When all options have been processed (that is, up to the first non-option argument), `getopt_long()` returns -1. The special option "--" (two hyphens) can be used to delimit the end of the options; when it is encountered, -1 is returned and "—" is skipped. This ooption is useful in delimiting non-option arguments that begin with "-" (hyphen).

If `getopt_long()` encounters a short option character *shortopts* string or a long option not described in the *longopts* array, it returns the question mark (?) character. It also returns a question mark (?) character in response to a missing option argument unless the first character of *shortopts* is a colon (:) (or the second character, if the first character is either a plus (+) or a minus (-)), in which case it returns a colon (:). In either case, if the application has not set *opterr* to 0 and the first character of *shortopts* is not a colon (:), `getopt_long()` prints a diagnostic message to `stderr`.

The `getopt_long_only()` function is equivalent to the `getopt_long()` function except that it allows the user of the application to pass long options with only a single hyphen (-) instead of "--". The "--" prefix is still recognized. However, when a single hyphen (-) is encountered, `getopt_long_only()` attempts to match this argument to a long option, including abreviations of the long option. If a long option starts with the same character as a short option, a single hyphen followed by that character (and no other characters) will be recognized as a short option. Use of `getopt_long_only()` is strongly discouraged by Sun and GNU for new applications.

The behavior of `getopt_clip()` differs from that of `getopt_long()` in the following ways:

- The `getopt_clip()` function does not perform argument reordering. The `getopt_clip()` function always enforces the POSIX behavior that all options should precede operands on the command line. Specifically, `getopt_clip()` does not reorder arguments but returns -1 and stops processing upon encountering the first operand argument.
- The environment variable `POSIXLY_CORRECT` is ignored (the `getopt_clip()` function behaves as though it were set.)
- The plus and minus characters do not have a special meaning as the first character of the *shortopts* argument. They are treated as any other character (other than the colon) would be treated.
- Optional option-arguments are not allowed. The behavior of `getopt_clip()` when `optional_argument` is specified as the value of `has_arg` in the *longopts* argument or double colons are included in the *shortopts* argument is unspecified.
- Long-option abbreviations are not recognized.
- Short options are required to have at least one long-option equivalent. That is, each character in *shortopts* must appear as the `val` member in one or more `option` structures. Similarly, each long option must have a short option equivalent, meaning that the `val` member of each `option` structure must appear in the *shortopts* string. If these requirements are not met, `getopt_clip()` returns -1 and sets `errno` to `EINVAL`.

Return Values For short options (other than `-W` when `W;` is in *shortopts*), these functions return the next option character specified on the command line. For long options, the value returned by these functions depends upon the value of the `flag` structure element for the identified option. If `flag` is `NULL`, the value contained in the `val` structure element for the long option encountered on the command line is returned. Otherwise, these functions return 0 (and the value specified in the `val` member for the long option is stored into the location pointed to by `flag`). When `W;` is in *shortopts* and `-W` is encountered in the command line and the option argument to `-W` matches a long-option name, the return state from these functions is as if the long option had been encountered. However, if no argument is specified to the long option, *optarg* is set to the option argument of `-W` (the long-option name or unique prefix). If the option argument of `-W` does not match a long option (or unique prefix), the return state is as for any other short option.

A colon (:) is returned if `getopt_long()` detects a missing argument and the first character of *shortopts* (other than a possible initial "+" or "-") was a colon (':').

A question mark (?) is returned if `getopt_long()` encounters an option letter not included in *shortopts* or detects a missing argument and the first character of *shortopts* (other than a possible initial “+” or “-”) was not a colon (:).

The `getopt_clip()` function expects all short options to have one or more long-option equivalent and all long options to have one short option equivalent (see NOTES for details). If proper equivalents are not found, `getopt_clip()` returns -1 and sets `errno` to `EINVAL`.

Errors The `getopt_clip()` function will fail if:

`EINVAL` A short option does not have at least one long-option equivalent, or a long option does not have at least one short-option equivalent.

Examples **EXAMPLE 1** Exmple using `getopt()`.

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

/* Flag set by ‘--verbose’. */
static int verbose_flag;

int
main (int argc, char **argv)
{
  int c;

  while (1) {
    static struct option long_options[] = {
      /* These options set a flag. */
      {"verbose", no_argument, &verbose_flag, 1},
      {"brief",   no_argument, &verbose_flag, 0},
      /* The following options don’t set a flag. */
      {"add",     no_argument, NULL, ’a’},
      {"append",  no_argument, NULL, ’b’},
      {"delete",  required_argument, NULL, ’d’},
      {"create",  required_argument, NULL, ’c’},
      {"file",    required_argument, NULL, ’f’},
      {0, 0, 0, 0}
    };
    /* getopt_long stores the option index here. */
    int option_index = 0;

    c = getopt_long (argc, argv, "abc:d:f:",
                     long_options, &option_index);

    /* Detect the end of the options. */
    if (c == -1)
      break;

    switch (c) {
      case 0:
        /* (In this example) only options which set */
```

EXAMPLE 1 Exmple using `getopt()`. *(Continued)*

```
        /* a flag return zero, so do nothing. */
        break;

      case 'a':
        puts ("option --add (-a)\n");
        break;

      case 'b':
        puts ("option --append (-b)\n");
        break;

      case 'c':
        printf ("option --create (-c) with value `%s'\n", optarg);
        break;

      case 'd':
        printf ("option --delete (-d) with value `%s'\n", optarg);
        break;

      case 'f':
        printf ("option --file (-f) with value `%s'\n", optarg);
        break;

      case '?':
        /* getopt_long already printed an error message. */
        break;

      default:
        abort ();
    }
  }

  /* Instead of reporting `--verbose'
     and `--brief' as they are encountered,
     we report the final status resulting from them. */
  if (verbose_flag)
    puts ("verbose flag is set");

  /* Print any remaining command line arguments (not options). */
  if (optind < argc) {
      printf ("non-option ARGV-elements: ");
      while (optind < argc)
        printf ("%s ", argv[optind++]);
      putchar ('\n');
    }
```

EXAMPLE 1 Exmple using `getopt()`. *(Continued)*

```
  exit (0);
}
```

Environment Variables

See environ(5) for descriptions of the following environment variables that affect the execution of `getopt_long()`: `LANG`, `LC_ALL`, and `LC_MESSAGES`.

`POSIXLY_CORRECT` When set (and the first character of the *shortopts* argument is neither a plus or minus sign), the POSIX rule that all operands must follow all options is enforced. Option processing terminates when the first operand is encountered. The `getopt_clip()` function ignores the setting of `POSIXLY_CORRECT` and always behaves as if it were set.

`LC_CTYPE` Determine the locale for the interpretation of sequences of bytes as characters in *shortopts* and the *longopts*[].`name` structure members.

Usage

The `getopt_long()` function does not fully check for mandatory arguments because there is no unambiguous algorithm to do so. Given an option string `a:b` and the input `-a -b`, `getopt_long()` assumes that `-b` is the mandatory argument to the `-a` option and not that `-a` is missing a mandatory argument. Indeed, the only time a missing option argument can be reliably detected is when the option is the final option on the command line and is not followed by any command arguments.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also

`Intro(1)`, `getopts(1)`, `getopt(3C)`, `getsubopt(3C)`, `gettext(3C)`, `setlocale(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Notes

Use of `getopt_long()` is discouraged for applications targeted strictly for Solaris. It should be used only for applications targeted at Solaris and platforms that adhere to the GNU command line conventions. The `getopt_long_only()` function is provided by Solaris and GNU for legacy applications and its use is discouraged by both current conventions.

The differences between the Solaris/GNU and OpenBSD versions of these functions are as follows:

- The handling of the hyphen (-) as the first character of the option string in presence of the environment variable `POSIXLY_CORRECT`:

Solaris/GNU — Operands are returned as arguments to option '\\1', and option processing does not stop until "--" or the end of *argv* is found.

OpenBSD — obeys `POSIXLY_CORRECT` and stops at the first non-option.

- The handling of the hyphen (-) within the *shortopts* parameter string when not the first character.

  Solaris/GNU — treats a single hyphen (-) on the command line as an operand.

  OpenBSD — treats a single hyphen (-) on the command line as an option. BSD recognizes this behavior as incorrect, but maintains it for compatibility.

- The return value in the event of a missing argument if the first character after "+" or "-" in the option string is not a colon (:)

  Solaris/GNU — returns "?".

  OpenBSD — returns ":" (since OpenBSD's getopt does).

- The setting *optopt* for long options with `flag` != `NULL`:

  Solaris/GNU — sets *optopt* to `val`.

  OpenBSD — sets *optopt* to 0 (since `val` would never be returned).

- The setting of *optarg* for long options without an argument that are invoked with `-W` (`W;` in option string):

  Solaris/GNU — sets *optarg* to the option name (the argument of `-W`).

  OpenBSD — sets *optarg* to `NULL` (the argument of the long option).

- The handling of `-W` with an argument that is not (a prefix to) a known long option (`W;` in option string):

  Solaris/GNU — returns 'W' with *optarg* set to the unknown option.

  OpenBSD — treats as an error (unknown option) and returns "?" with *optopt* set to 0 and *optarg* set to `NULL`.

- The error messages are different (all).

- The implementations do not permute the argument vector at the same points in the calling sequence. The aspects normally used by the caller (ordering after -1 is returned, value of *optind* relative to current positions) are the same. Applications should not depend upon the ordering of the argument vector before -1 is returned.

Name getpagesize – get system page size

Synopsis

```
#include <unistd.h>

int getpagesize(void);
```

Description The `getpagesize()` function returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a system page size and need not be the same as the underlying hardware page size.

The `getpagesize()` function is equivalent to `sysconf(_SC_PAGE_SIZE)` and `sysconf(_SC_PAGESIZE)`. See `sysconf(3C)`.

Return Values The `getpagesize()` function returns the current page size.

Errors No errors are defined.

Usage The value returned by `getpagesize()` need not be the minimum value that `malloc(3C)` can allocate. Moreover, the application cannot assume that an object of this size can be allocated with `malloc()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `pagesize(1)`, `brk(2)`, `getrlimit(2)`, `mmap(2)`, `mprotect(2)`, `munmap(2)`, `malloc(3C)`, `msync(3C)`, `sysconf(3C)`, `attributes(5)`

Name getpagesizes – get system supported page sizes

Synopsis

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

int getpagesizes(size_t pagesize[ ], int nelem);
```

Description The `getpagesizes()` function returns either the number of different page sizes supported by the system or the actual sizes themselves. When called with *nelem* as 0 and pagesize as `NULL`, `getpagesizes()` returns the number of supported page sizes. Otherwise, up to *nelem* page sizes are retrieved and assigned to successive elements of *pagesize*[ ]. The return value is the number of page sizes retrieved and set in *pagesize*[ ].

Return Values Upon successful completion, the number of pagesizes supported or actually retrieved is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `getpagesizes()` function will fail if:

`EINVAL` The *nelem* argument is less than 0 or *pagesize* is `NULL` but *nelem* is non-zero.

Usage The `getpagesizes()` function returns all the page sizes for which the hardware and system software provide support for the `memcntl(2)` command `MC_HAT_ADVISE`. Not all processors support all page sizes or combinations of page sizes with equal efficiency. Applications programmers should take this into consideration when using `getpagesizes()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `memcntl(2)`, `mmap(2)`, `getpagesize(3C)`, `attributes(5)`

Name getpass, getpassphrase – read a string of characters without echo

Synopsis

```
#include <stdlib.h>

char *getpass(const char *prompt);

char *getpassphrase(const char *prompt);
```

XPG4, SUS, SUSv2

```
#include <unistd.h>

char *getpass(const char *prompt);
```

Description The `getpass()` function opens the process's controlling terminal, writes to that device the null-terminated string *prompt*, disables echoing, reads a string of characters up to the next newline character or EOF, restores the terminal state and closes the terminal.

The `getpassphrase()` function is identical to `getpass()`, except that it reads and returns a string of up to 257 characters in length.

Return Values Upon successful completion, `getpass()` returns a pointer to a null-terminated string of at most 9 bytes that were read from the terminal device. If an error is encountered, the terminal state is restored and a null pointer is returned.

Errors The `getpass()` and `getpassphrase()` functions may fail if:

`EINTR` The function was interrupted by a signal.

`EIO` The process is a member of a background process attempting to read from its controlling terminal, the process is ignoring or blocking the `SIGTTIN` signal or the process group is orphaned.

`EMFILE` `OPEN_MAX` file descriptors are currently open in the calling process.

`ENFILE` The maximum allowable number of files is currently open in the system.

`ENXIO` The process does not have a controlling terminal.

Usage The return value points to static data whose content may be overwritten by each call.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	`getpass()` is Standard.
MT-Level	Unsafe

See Also `attributes(5)`, `standards(5)`

Name getpeerucred – get connected socket or stream peer's credentials

Synopsis

```
#include <ucred.h>

int getpeerucred(int fd, ucred_t **ucred);
```

Description The `getpeerucred()` function returns the credentials of the peer endpoint of a connection-oriented socket (`SOCK_STREAM`) or stream *fd* at the time the endpoint was created or the connection was established. A process that initiates a connection retrieves the credentials of its peer at the time the peer's endpoint was created. A process that listens for connections retrieves the credentials of the peer at the time the peer initiated the connection.

When successful, `getpeerucred()` stores the pointer to a freshly allocated `ucred_t` in the memory location pointed to by the *ucred* argument if that memory location contains the null pointer. If the memory location is non-null, it will reuse the existing `ucred_t`.

When *ucred* is no longer needed, a credential allocated by `getpeerucred()` should be freed with `ucred_free(3C)`.

It is possible that all fields of the `ucred_t` are not available to all peer endpoints and all callers.

Return Values Upon successful completion, `getpeerucred()` returns 0. Otherwise, it returns −1 and `errno` is set to indicate the error.

Errors The `getpeerucred()` function will fail if:

`EAGAIN` There is not enough memory available to allocate sufficient memory to hold the user credential. The application can try again later.

`EBADF` The *fd* argument is not a valid file descriptor.

`EFAULT` The pointer location pointed to by the `ucred_t **` argument points to an invalid, non-null address.

`EINVAL` The socket is connected but the peer credentials are unknown.

`ENOMEM` The physical limits of the system are exceeded by the memory allocation needed to hold the user credential.

`ENOTCONN` The socket or stream is not connected or the stream's peer is unknown.

`ENOTSUP` This operation is not supported on this file descriptor.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also door_ucred(3C), ucred_get(3C), attributes(5), connld(7M)

Notes The system currently supports both sides of connection endpoints for local AF_UNIX, AF_INET, and AF_INET6 sockets, /dev/tcp, /dev/ticots, and /dev/ticotsord XTI/TLI connections, and pipe file descriptors sent using I_SENDFD as a result of the open of a named pipe with the "connld" module pushed.

Name getpriority, setpriority – get and set the nice value

Synopsis

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

int getpriority(int which, id_t who);

int setpriority(int which, id_t who, int value);
```

Description The `getpriority()` function obtains the nice value of a process, thread, or set of processes. The `setpriority()` function sets the nice value of a process, thread, or set of processes to *value*+`NZERO`, where `NZERO` is defined to be 20.

Target entities are specified by the values of the *which* and *who* arguments. The *which* argument can be one of the following values: `PRIO_PROCESS`, `PRIO_PGRP`, `PRIO_USER`, `PRIO_GROUP`, `PRIO_SESSION`, `PRIO_LWP`, `PRIO_TASK`, `PRIO_PROJECT`, `PRIO_ZONE`, or `PRIO_CONTRACT`, indicating that the *who* argument is to be interpreted as a process ID, a process group ID, an effective user ID, an effective group ID, a session ID, a thread (lwp) ID, a task ID, a project ID, a zone ID, or a process contract ID, respectively. A 0 value for the *who* argument specifies the current process, process group, or user. A 0 value for the *who* argument is treated as valid group ID, session ID, thread (lwp) ID, task ID, project ID, zone ID, or process contract ID. A `P_MYID` value for the *who* argument can be used to specify the current group, session, thread, task, project, zone, or process contract, respectively.

If a specified process is multi-threaded, the nice value set with `setpriority()` affects all threads in the process.

If more than one process is specified, `getpriority()` returns `NZERO` less than the lowest nice value pertaining to any of the specified entities, and `setpriority()` sets the nice values of all of the specified processes to *value*+`NZERO`.

The default nice value is `NZERO`. Lower nice values cause more favorable scheduling. The range of valid nice values is 0 to `NZERO`*2-1. If *value*+`NZERO` is less than the system's lowest supported nice value, `setpriority()` sets the nice value to the lowest supported value. If *value*+`NZERO` is greater than the system's highest supported nice value, `setpriority()` sets the nice value to the highest supported value.

Only a process with appropriate privileges can lower the nice value.

Any process or thread using `SCHED_FIFO` or `SCHED_RR` is unaffected by a call to `setpriority()`. This is not considered an error. A process or thread that subsequently reverts to `SCHED_OTHER` will not have its priority affected by such a `setpriority()` call.

The effect of changing the nice value varies depending on the scheduling policy in effect.

Since `getpriority()` can return the value -1 on successful completion, it is necessary to set `errno` to 0 prior to a call to `getpriority()`. If `getpriority()` returns the value -1, then `errno` can be checked to see if an error occurred or if the value is a legitimate nice value.

Return Values Upon successful completion, `getpriority()` returns an integer in the range from `-NZERO` to `NZERO-1`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Upon successful completion, `setpriority()` returns `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `getpriority()` and `setpriority()` functions will fail if:

`ESRCH` No process or thread could be located using the *which* and *who* argument values specified.

`EINVAL` The value of the *which* argument was not recognized, or the value of the *who* argument is not a valid process ID, process group ID, user ID, group ID, session ID, thread (lwp) ID, task ID, project ID, or zone ID.

In addition, `setpriority()` may fail if:

`EPERM` A process was located, but neither the real nor effective user ID of the executing process match the effective user ID of the process whose nice value is being changed.

`EACCES` A request was made to change the nice value to a lower numeric value and the current process does not have appropriate privileges.

Examples **EXAMPLE 1** Example using `getpriority()`

The following example returns the current scheduling priority for the process ID returned by the call to `getpid(2)`.

```
#include <sys/resource.h>
...
int which = PRIO_PROCESS;
id_t pid;
int ret;

pid = getpid();
ret = getpriority(which, pid);
```

EXAMPLE 2 Example using `setpriority()`

The following example sets the nice value for the current process to 0.

```
#include <sys/resource.h>
...
int which = PRIO_PROCESS;
id_t pid;
int value = -20;
int ret;

pid = getpid();
```

EXAMPLE 2 Example using `setpriority()` *(Continued)*

```
ret = setpriority(which, pid, value);
```

Usage

The `getpriority()` and `setpriority()` functions work with an offset nice value (*value*-`NZERO`). The nice value is in the range 0 to 2*`NZERO`-1, while the return value for `getpriority()` and the third parameter for `setpriority()` are in the range -`NZERO` to `NZERO`-1.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Standard	See `standards(5)`.

See Also

`nice(1)`, `renice(1)`, `sched_get_priority_max(3C)`, `sched_setscheduler(3C)`, `attributes(5)`, `standards(5)`

Name getprogname, setprogname – get or set the program name

Synopsis

```
#include <stdlib.h>

const char *getprogname(void);

void setprogname(const char *progname);
```

Description The `getprogname()` function returns the name of the program. If the name has not yet been set, it returns `NULL`.

The `setprogname()` function sets the name of the program to be the last component of the *progname* argument. Since a pointer to the given string is kept as the program name, it should not be modified for the duration of the program.

These functions are used by error-reporting routines to produce consistent output.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `err(3C)`, `attributes(5)`

Name getpw – get passwd entry from UID

Synopsis

```
#include <stdlib.h>

int getpw(uid_t uid, char *buf);
```

Description The `getpw()` function searches the user data base for a user id number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. `getpw()` returns non-zero if *uid* cannot be found.

Usage This function is included only for compatibility with prior systems and should not be used; the functions described on the `getpwnam(3C)` manual page should be used instead.

If the `/etc/passwd` and the `/etc/group` files have a plus sign (+) for the NIS entry, then `getpwent()` and `getgrent()` will not return `NULL` when the end of file is reached. See `getpwnam(3C)`.

Return Values The `getpw()` function returns non-zero on error.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `getpwnam(3C)`, `passwd(4)`, `attributes(5)`

Name getpwnam, getpwnam_r, getpwent, getpwent_r, getpwuid, getpwuid_r, setpwent, endpwent, fgetpwent, fgetpwent_r – get password entry

Synopsis

```
#include <pwd.h>

struct passwd *getpwnam(const char *name);

struct passwd *getpwnam_r(const char *name, struct passwd *pwd,
     char *buffer, int buflen);

struct passwd *getpwent(void);

struct passwd *getpwent_r(struct passwd *pwd, char *buffer,
     int buflen);

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwuid_r(uid_t uid, struct passwd *pwd,
    char *buffer, int  buflen);

void setpwent(void);

void endpwent(void);

struct passwd *fgetpwent(FILE *f);

struct passwd *fgetpwent_r(FILE *f, struct passwd *pwd,
     char *buffer, int buflen);
```

Standard conforming

```
cc [ flag... ] file... -D_POSIX_PTHREAD_SEMANTICS [ library... ]

int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,
     size_t bufsize, struct passwd **result);

int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
     size_t bufsize, struct passwd **result);
```

Description These functions are used to obtain password entries. Entries can come from any of the sources for `passwd` specified in the `/etc/nsswitch.conf` file (see `nsswitch.conf(4)`).

The `getpwnam()` function searches for a password entry with the login name specified by the character string parameter *name*.

The `getpwuid()` function searches for a password entry with the (numeric) user ID specified by the *uid* parameter.

The `setpwent()`, `getpwent()`, and `endpwent()` functions are used to enumerate password entries from the database. The `setpwent()` function sets (or resets) the enumeration to the beginning of the set of password entries. This function should be called before the first call to `getpwent()`. Calls to `getpwnam()` and `getpwuid()` leave the enumeration position in an indeterminate state. Successive calls to `getpwent()` return either successive entries or a null pointer, indicating the end of the enumeration.

The `endpwent()` function may be called to indicate that the caller expects to do no further password retrieval operations; the system may then close the password file, deallocate resources it was using, and so forth. It is still allowed, but possibly less efficient, for the process to call more password functions after calling `endpwent()`.

The `fgetpwent()` function, unlike the other functions above, does not use `nsswitch.conf` but reads and parses the next line from the stream *f*, which is assumed to have the format of the `passwd` file. See `passwd(4)`.

Reentrant Interfaces

The `getpwnam()`, `getpwuid()`, `getpwent()`, and `fgetpwent()` functions use thread–specific data storage that is reused in each call to one of these functions by the same thread, making them safe to use but not recommended for multithreaded applications.

The parallel functions `getpwnam_r()`, `getpwuid_r()`, `getpwent_r()`, and `fgetpwent_r()` provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the " _r " suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results instead of using thread-specific data that can be overwritten by each call. They are safe for use in both single-threaded and multithreaded applications.

Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The *pwd* parameter must be a pointer to a `struct passwd` structure allocated by the caller. On successful completion, the function returns the password entry in this structure. The parameter *buffer* is a pointer to a buffer supplied by the caller, used as storage space for the password data. All pointers within the returned `struct passwd` *pwd* point to data stored within this buffer; see `passwd Structure` below. The buffer must be large enough to hold all the data associated with the password entry. The parameter *buflen* (or *bufsize* for the standard-conforming versions; see `standards(5)`) should give the size in bytes of *buffer*. The maximum size needed for this buffer can be determined with the {`_SC_GETPW_R_SIZE_MAX`} `sysconf(3C)` parameter. The standard-conforming versions place a pointer to the modified *pwd* structure in the *result* parameter, instead of returning a pointer to this structure. A null pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. The `setpwent()` function can be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to `getpwent_r()`, the threads will enumerate disjoint subsets of the password database.

Like their non-reentrant counterparts, `getpwnam_r()` and `getpwuid_r()` leave the enumeration position in an indeterminate state.

passwd Structure

Password entries are represented by the `struct passwd` structure defined in <pwd.h>:

```
struct passwd {
    char *pw_name;      /* user's login name */
    char *pw_passwd;    /* no longer used */
    uid_t pw_uid;       /* user's uid */
    gid_t pw_gid;       /* user's gid */
    char *pw_age;       /* not used */
    char *pw_comment;   /* not used */
    char *pw_gecos;     /* typically user's full name */
    char *pw_dir;       /* user's home dir */
    char *pw_shell;     /* user's login shell */
};
```

The `pw_passwd` member should not be used as the encrypted password for the user; use `getspnam()` or `getspnam_r()` instead. See `getspnam(3C)`.

Return Values

The `getpwnam()`, `getpwnam_r()`, `getpwuid()`, and `getpwuid_r()` functions each return a pointer to a `struct passwd` if they successfully locate the requested entry. A null pointer is returned if the requested entry is not found, or an error occurs. On error, `errno` is set to indicate the error.

Applications wishing to check for error situations should set `errno` to 0 before calling `getpwnam()`, `getpwnam_r()`, `getpwuid()`, `getpwuid_r()`, `getpwent()`, `getpwent_r()`, `fgetpwent()`, and `fgetpwent_r()`. If these non-reentrant functions return a null pointer and `errno` is non-zero, an error occurred.

The standard-conforming functions `getpwnam_r()` and `getpwuid_r()` can return `0` even on an error, particularly in the case where the requested entry is not found. The application needs to check the return value and that the *pwd* pointer is non-null. Otherwise, an error value is returned to indicate the error.

The `getpwent()`, `getpwent_r()`, `fgetpwent()`, and `fgetpwent_r()` functions each return a pointer to a `struct passwd` if they successfully enumerate an entry; otherwise they return a null pointer on end-of-file or error. On error, `errno` is set to indicate the error.

See `Intro(2)` for the proper usage and interpretation of `errno` in multithreaded applications.

The `getpwnam()`, `getpwuid()`, `getpwent()`, and `fgetpwent()` functions use thread–specific data storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved.

When the pointer returned by the reentrant functions `getpwnam_r()`, `getpwuid_r()`, `getpwent_r()`, and `fgetpwent_r()` is non-null, it is always equal to the *pwd* pointer that was supplied by the caller.

Errors The `getpwent_r()`, `fgetpwent()`, and `fgetpwent_r()` functions will fail if:

`EIO` An I/O error has occurred.

`ERANGE` Insufficient storage was supplied by *buffer* and *bufsize* to contain the data to be referenced by the resulting `passwd` structure.

The `getpwent_r()` function will fail if:

`EMFILE` There are {`OPEN_MAX`} file descriptors currently open in the calling process.

`ENFILE` The maximum allowable number of files is currently open in the system.

The `getpwnam()`, `getpwnam_r()`, `getpwuid()`, `getpwuid_r()`, `getpwent()`, `setpwent()`, and `endpwent()` functions may fail if:

`EIO` An I/O error has occurred.

The `getpwnam()`, `getpwnam_r()`, `getpwuid()`, `getpwuid_r()`, `getpwent()`, and `setpwent()` functions may fail if:

`EMFILE` There are {`OPEN_MAX`} file descriptors currently open in the calling process.

`ENFILE` The maximum allowable number of files is currently open in the system.

The `getpwnam()`, `getpwnam_r()`, `getpwuid()`, and `getpwuid_r()` functions may fail if:

`EINTR` A signal was caught during the execution of the function call.

The `getpwnam_r()` and `getpwuid_r()` functions may fail if:

`ERANGE` Insufficient storage was supplied by *buffer* and *bufsize* to contain the data to be referenced by the resulting `passwd` structure.

Usage Three names associated with the current process can be determined: `getpwuid(geteuid())` returns the name associated with the effective user ID of the process; `getlogin()` returns the name associated with the current login activity; and `getpwuid(getuid())` returns the name associated with the real user ID of the process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See `Reentrant Interfaces` in `DESCRIPTION`.
Standard	See below.

For `endpwent()`, `getpwent()`, `getpwnam()`, `getpwnam_r()`, `getpwuid()`, `getpwuid_r()`, and `setpwent()`, see `standards(5)`.

See Also `passwd(1)`, `yppasswd(1)`, `Intro(2)`, `Intro(3)`, `cuserid(3C)`, `getgrnam(3C)`, `getlogin(3C)`, `getspnam(3C)`, `nsswitch.conf(4)`, `passwd(4)`, `shadow(4)`, `attributes(5)`, `standards(5)`

Notes When compiling multithreaded programs, see `Intro(3)`.

Use of the enumeration interfaces `getpwent()` and `getpwent_r()` is discouraged; enumeration is supported for the `passwd` file and NIS, but in general is not efficient and might not be supported for all database sources. The semantics of enumeration are discussed further in `nsswitch.conf(4)`.

Previous releases allowed the use of '+' and '-' entries in `/etc/passwd` to selectively include and exclude NIS entries. The primary usage of these '+/-' entries is superseded by the name service switch, so the '+/-' form might not be supported in future releases.

If required, the '+/-' functionality can still be obtained for NIS by specifying `compat` as the source for `passwd`.

If the '+/-' is used, both `/etc/shadow` and `/etc/passwd` should have the same '+' and '-' entries to ensure consistency between the password and shadow databases.

If a password entry from any of the sources contains an empty *uid* or *gid* field, that entry will be ignored by the files and NIS name service switch backends, causing the user to appear unknown to the system.

If a password entry contains an empty *gecos*, *home directory*, or *shell* field, `getpwnam()` and `getpwnam_r()` return a pointer to a null string in the respective field of the `passwd` structure.

If the shell field is empty, `login(1)` automatically assigns the default shell. See `login(1)`.

Solaris 2.4 and earlier releases provided definitions of the `getpwnam_r()` and `getpwuid_r()` functions as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface for these functions. Support for the Draft 6 interface is provided for compatibility only and might not be supported in future releases. New applications and libraries should use the standard-conforming interface.

For POSIX.1c-conforming applications, the `_POSIX_PTHREAD_SEMANTICS` and `_REENTRANT` flags are automatically turned on by defining the `_POSIX_C_SOURCE` flag with a value ≥199506L.

Name getrusage – get information about resource utilization

Synopsis

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

int getrusage(int who, struct rusage *r_usage);
```

Description The `getrusage()` function provides measures of the resources used by the current process, its terminated and waited-for child processes, or the current light weight process (LWP). If the value of the *who* argument is `RUSAGE_SELF`, information is returned about resources used by the current process. If the value of the *who* argument is `RUSAGE_CHILDREN`, information is returned about resources used by the terminated and waited-for children of the current process. If the child is never waited for (for instance, if the parent has `SA_NOCLDWAIT` set or sets `SIGCHLD` to `SIG_IGN`), the resource information for the child process is discarded and not included in the resource information provided by `getrusage()`. If the value of the *who* argument is `RUSAGE_LWP`, information is returned about resources used by the current LWP.

The *r_usage* argument is a pointer to an object of type `struct rusage` in which the returned information is stored. The members of `rusage` are as follows:

```
struct timeval  ru_utime;     /* user time used */
struct timeval  ru_stime;     /* system time used */
long            ru_maxrss;    /* maximum resident set size */
long            ru_idrss;     /* integral resident set size */
long            ru_minflt;    /* page faults not requiring physical
                                 I/O */
long            ru_majflt;    /* page faults requiring physical I/O */
long            ru_nswap;     /* swaps */
long            ru_inblock;   /* block input operations */
long            ru_oublock;   /* block output operations */
long            ru_msgsnd;    /* messages sent */
long            ru_msgrcv;    /* messages received */
long            ru_nsignals;  /* signals received */
long            ru_nvcsw;     /* voluntary context switches */
long            ru_nivcsw;    /* involuntary context switches */
```

The structure members are interpreted as follows:

`ru_utime` The total amount of time spent executing in user mode. Time is given in seconds and microseconds.

`ru_stime` The total amount of time spent executing in system mode. Time is given in seconds and microseconds.

`ru_maxrss` The maximum resident set size. Size is given in pages (the size of a page, in bytes, is given by the `getpagesize(3C)` function). See the `NOTES` section of this page.

`ru_idrss` An “integral” value indicating the amount of memory in use by a process while the process is running. This value is the sum of the resident set sizes of

the process running when a clock tick occurs. The value is given in pages times clock ticks. It does not take sharing into account. See the NOTES section of this page.

`ru_minflt` The number of page faults serviced which did not require any physical I/O activity. See the NOTES section of this page.

`ru_majflt` The number of page faults serviced which required physical I/O activity. This could include page ahead operations by the kernel. See the NOTES section of this page.

`ru_nswap` The number of times a process was swapped out of main memory.

`ru_inblock` The number of times the file system had to perform input in servicing a `read(2)` request.

`ru_oublock` The number of times the file system had to perform output in servicing a `write(2)` request.

`ru_msgsnd` The number of messages sent over sockets.

`ru_msgrcv` The number of messages received from sockets.

`ru_nsignals` The number of signals delivered.

`ru_nvcsw` The number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).

`ru_nivcsw` The number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

Return Values Upon successful completion, `getrusage()` returns `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `getrusage()` function will fail if:

`EFAULT` The address specified by the *r_usage* argument is not in a valid portion of the process' address space.

`EINVAL` The `who` parameter is not a valid value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Standard	See `standards(5)`.

See Also sar(1M), read(2), times(2), write(2), getpagesize(3C), gettimeofday(3C), wait(3C), attributes(5), standards(5)

Notes The ru_maxrss, ru_ixrss, ru_idrss, and ru_isrss members of the rusage structure are set to 0 in this implementation.

The numbers ru_inblock and ru_oublock account only for real I/O, and are approximate measures at best. Data supplied by the cache mechanism is charged only to the first process to read and the last process to write the data.

The way resident set size is calculated is an approximation, and could misrepresent the true resident set size.

Page faults can be generated from a variety of sources and for a variety of reasons. The customary cause for a page fault is a direct reference by the program to a page which is not in memory. Now, however, the kernel can generate page faults on behalf of the user, for example, servicing read(2) and write(2) functions. Also, a page fault can be caused by an absent hardware translation to a page, even though the page is in physical memory.

In addition to hardware detected page faults, the kernel may cause pseudo page faults in order to perform some housekeeping. For example, the kernel may generate page faults, even if the pages exist in physical memory, in order to lock down pages involved in a raw I/O request.

By definition, major page faults require physical I/O, while minor page faults do not require physical I/O. For example, reclaiming the page from the free list would avoid I/O and generate a minor page fault. More commonly, minor page faults occur during process startup as references to pages which are already in memory. For example, if an address space faults on some “hot” executable or shared library, this results in a minor page fault for the address space. Also, any one doing a read(2) or write(2) to something that is in the page cache will get a minor page fault(s) as well.

There is no way to obtain information about a child process which has not yet terminated.

Name gets, fgets – get a string from a stream

Synopsis

```
#include <stdio.h>

char *gets(char *s);

char *fgets(char *s, int n, FILE *stream);
```

Description The `gets()` function reads bytes from the standard input stream (see `Intro(3)`), `stdin`, into the array pointed to by *s*, until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null byte.

If the length of an input line exceeds the size of *s*, indeterminate behavior may result. For this reason, it is strongly recommended that `gets()` be avoided in favor of `fgets()`.

The `fgets()` function reads bytes from the *stream* into the array pointed to by *s*, until *n*−1 bytes are read, or a newline character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null byte.

The `fgets()` and `gets()` functions may mark the `st_atime` field of the file associated with *stream* for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc(3C)`, `fgets()`, `fread(3C)`, `fscanf(3C)`, `getc(3C)`, `getchar(3C)`, `getdelim(3C)`, `getline(3C)`, `gets()`, or `scanf(3C)` using *stream* that returns data not supplied by a prior call to `ungetc(3C)` or `ungetwc(3C)`.

Return Values If end-of-file is encountered and no bytes have been read, no bytes are transferred to *s* and a null pointer is returned. For standard-conforming (see `standards(5)`) applications, if the end-of-file indicator for the stream is set, no bytes are transferred to *s* and a null pointer is returned whether or not the stream is at end-of-file. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a null pointer is returned and the error indicator for the stream is set. If end-of-file is encountered, the `EOF` indicator for the stream is set. Otherwise *s* is returned.

Errors Refer to `fgetc(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `lseek(2)`, `read(2)`, `ferror(3C)`, `fgetc(3C)`, `fgetwc(3C)`, `fopen(3C)`, `fread(3C)`, `getchar(3C)`, `getdelim(3C)`, `getline(3C)`, `scanf(3C)`, `stdio(3C)`, `ungetc(3C)`, `ungetwc(3C)`, `attributes(5)`, `standards(5)`

Name getspnam, getspnam_r, getspent, getspent_r, setspent, endspent, fgetspent, fgetspent_r – get password entry

Synopsis

```
#include <shadow.h>

struct spwd *getspnam(const char *name);

struct spwd *getspnam_r(const char *name, struct spwd *result,
     char *buffer, int buflen);

struct spwd *getspent(void);

struct spwd *getspent_r(struct spwd *result, char *buffer,
     int buflen);

void setspent(void);

void endspent(void);

struct spwd *fgetspent(FILE *fp);

struct spwd *fgetspent_r(FILE *fp, struct spwd *result,
     char *buffer, int buflen);
```

Description These functions are used to obtain shadow password entries. An entry may come from any of the sources for `shadow` specified in the `/etc/nsswitch.conf` file (see `nsswitch.conf(4)`).

The `getspnam()` function searches for a shadow password entry with the login name specified by the character string argument *name*.

The `setspent()`, `getspent()`, and `endspent()` functions are used to enumerate shadow password entries from the database.

The `setspent()` function sets (or resets) the enumeration to the beginning of the set of shadow password entries. This function should be called before the first call to `getspent()`. Calls to `getspnam()` leave the enumeration position in an indeterminate state.

Successive calls to `getspent()` return either successive entries or `NULL`, indicating the end of the enumeration.

The `endspent()` function may be called to indicate that the caller expects to do no further shadow password retrieval operations; the system may then close the shadow password file, deallocate resources it was using, and so forth. It is still allowed, but possibly less efficient, for the process to call more shadow password functions after calling `endspent()`.

The `fgetspent()` function, unlike the other functions above, does not use `nsswitch.conf`; it reads and parses the next line from the stream *fp*, which is assumed to have the format of the `shadow` file (see `shadow(4)`).

Reentrant Interfaces

The `getspnam()`, `getspent()`, and `fgetspent()` functions use thread-specific data storage that is reused in each call to one of these functions by the same thread, making them safe to use but not recommended for multithreaded applications.

The `getspnam_r()`, `getspent_r()`, and `fgetspent_r()` functions provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the `_r` suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multithreaded applications.

Each reentrant interface takes the same argument as its non-reentrant counterpart, as well as the following additional arguments. The *result* argument must be a pointer to a `struct spwd` structure allocated by the caller. On successful completion, the function returns the shadow password entry in this structure. The *buffer* argument must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the shadow password data. All of the pointers within the returned `struct spwd` *result* point to data stored within this buffer (see `RETURN VALUES`). The buffer must be large enough to hold all of the data associated with the shadow password entry. The *buflen* argument should give the size in bytes of the buffer indicated by *buffer*.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. The `setspent()` function may be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to `getspent_r()`, the threads will enumerate disjoint subsets of the shadow password database.

Like its non-reentrant counterpart, `getspnam_r()` leaves the enumeration position in an indeterminate state.

Return Values

Password entries are represented by the `struct spwd` structure defined in `<shadow.h>`:

```
struct spwd{
    char           *sp_namp;     /* login name */
    char           *sp_pwdp;     /* encrypted passwd */
    int            sp_lstchg;    /* date of last change */
    int            sp_min;       /* min days to passwd change */
    int            sp_max;       /* max days to passwd change*/
    int            sp_warn;      /* warning period */
    int            sp_inact;     /* max days inactive */
    int            sp_expire;    /* account expiry date */
    unsigned int   sp_flag;      /* not used */
};
```

See shadow(4) for more information on the interpretation of this data.

The `getspnam()`and `getspnam_r()` functions each return a pointer to a `struct spwd` if they successfully locate the requested entry; otherwise they return `NULL`.

The `getspent()`, `getspent_r()`, `fgetspent()`, and `fgetspent()` functions each return a pointer to a `struct spwd` if they successfully enumerate an entry; otherwise they return `NULL`, indicating the end of the enumeration.

The `getspnam()`, `getspent()`, and `fgetspent()` functions use thread–specific data storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved.

When the pointer returned by the reentrant functions `getspnam_r()`, `getspent_r()`, and `fgetspent_r()` is non-null, it is always equal to the *result* pointer that was supplied by the caller.

Errors The reentrant functions `getspnam_r()`, `getspent_r()`, and `fgetspent_r()` will return NULL and set `errno` to `ERANGE` if the length of the buffer supplied by caller is not large enough to store the result. See `Intro(2)` for the proper usage and interpretation of `errno` in multithreaded applications.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	See “Reentrant Interfaces” in `DESCRIPTION`.

See Also `passwd(1)`, `yppasswd(1)`, `Intro(3)`, `getlogin(3C)`, `getpwnam(3C)`, `nsswitch.conf(4)`, `passwd(4)`, `shadow(4)`, `attributes(5)`

Warnings The reentrant interfaces `getspnam_r()`, `getspent_r()`, and `fgetspent_r()` are included in this release on an uncommitted basis only, and are subject to change or removal in future minor releases.

Notes When compiling multithreaded applications, see `Intro(3)`, *Notes On Multithreaded Applications*, for information about the use of the `_REENTRANT` flag.

Use of the enumeration interfaces `getspent()` and `getspent_r()` is not recommended; enumeration is supported for the shadow file and NIS, but in general is not efficient and may not be supported for all database sources. The semantics of enumeration are discussed further in `nsswitch.conf(4)`.

Access to shadow password information may be restricted in a manner depending on the database source being used. Access to the `/etc/shadow` file is generally restricted to processes running with the effective uid of the file owner or the {`PRIV_FILE_DAC_READ`} privilege. Other database sources may impose stronger or less stringent restrictions.

Empty fields in the database source return -1 values for all fields except `sp_pwdp` and `sp_flag`, where the value returned is 0.

When NIS is used as the database source, the information for the shadow password entries is obtained from the “passwd.byname" map. This map stores only the information for the `sp_namp` and `sp_pwdp` fields of the `struct spwd` structure. Shadow password entries obtained from NIS will contain the value `-1` in the remainder of the fields.

Name getsubopt – parse suboption arguments from a string

Synopsis

```
#include <stdlib.h>

int getsubopt(char **optionp, char * const *keylistp, char **valuep);
```

Description The `getsubopt()` function parses suboption arguments in a flag argument. Such options often result from the use of `getopt(3C)`.

The `getsubopt()` argument *optionp* is a pointer to a pointer to the option argument string. The suboption arguments are separated by commas and each can consist of either a single token or a token-value pair separated by an equal sign.

The *keylistp* argument is a pointer to a vector of strings. The end of the vector is identified by a null pointer. Each entry in the vector is one of the possible tokens that might be found in **optionp*. Since commas delimit suboption arguments in *optionp*, they should not appear in any of the strings pointed to by *keylistp*. Similarly, because an equal sign separates a token from its value, the application should not include an equal sign in any of the strings pointed to by *keylistp*.

The *valuep* argument is the address of a value string pointer.

If a comma appears in *optionp*, it is interpreted as a suboption separator. After commas have been processed, if there are one or more equal signs in a suboption string, the first equal sign in any suboption string is interpreted as a separator between a token and a value. Subsequent equal signs in a suboption string are interpreted as part of the value.

If the string at **optionp* contains only one suboption argument (equivalently, no commas), `getsubopt()` updates **optionp* to point to the null character at the end of the string. Otherwise, it isolates the suboption argument by replacing the comma separator with a null character and updates **optionp* to point to the start of the next suboption argument. If the suboption argument has an associated value (equivalently, contains an equal sign), `getsubopt()` updates **valuep* to point to the value's first character. Otherwise, it sets **valuep* to a null pointer. The calling application can use this information to determine whether the presence or absence of a value for the suboption is an error.

Additionally, when `getsubopt()` fails to match the suboption with a token in the *keylistp* array, the calling application should decide if this is an error or if the unrecognized option should be processed in another way.

Return Values The `getsubopt()` function returns the index of the matched token string or -1 if no token strings were matched.

Errors No errors are defined.

Examples

EXAMPLE 1 Use `getsubopt()` to process options.

The following example demonstrates the processing of options to the `mount(1M)` utility using `getsubopt()`.

```
#include <stdlib.h>

char *myopts[] = {
#define READONLY     0
            "ro",
#define READWRITE    1
            "rw",
#define WRITESIZE    2
            "wsize",
#define READSIZE     3
            "rsize",
            NULL};

main(argc, argv)
    int  argc;
    char **argv;
{
    int sc, c, errflag;
    char *options, *value;
    extern char *optarg;
    extern int optind;
    .
    .
    .
    while((c = getopt(argc, argv, "abf:o:")) != -1) {
        switch (c) {
        case 'a': /* process a option */
            break;
        case 'b': /* process b option */
            break;
        case 'f':
            ofile = optarg;
            break;
        case '?':
            errflag++;
            break;
        case 'o':
            options = optarg;
            while (*options != '\0') {
                switch(getsubopt(&options,myopts,&value)){
                case READONLY : /* process ro option */
                    break;
                case READWRITE : /* process rw option */
                    break;
```

EXAMPLE 1 Use `getsubopt()` to process options. *(Continued)*

```
                case WRITESIZE : /* process wsize option */
                    if (value == NULL) {
                        error_no_arg();
                        errflag++;
                    } else
                        write_size = atoi(value);
                    break;
                case READSIZE : /* process rsize option */
                    if (value == NULL) {
                        error_no_arg();
                        errflag++;
                    } else
                        read_size = atoi(value);
                    break;
                default :
                    /* process unknown token */
                    error_bad_token(value);
                    errflag++;
                    break;
                  }
            }
              break;
        }
    }
    if (errflag) {
        /* print usage instructions etc. */
    }
    for (; optind<argc; optind++) {
        /* process remaining arguments */
    }
    .
    .
    .
}
```

EXAMPLE 2 Parse suboptions.

The following example uses the `getsubopt()` function to parse a value argument in the *optarg* external variable returned by a call to `getopt(3C)`.

```
#include <stdlib.h>
...
char *tokens[] = {"HOME", "PATH", "LOGNAME", (char *) NULL };
char *value;
int opt, index;
while ((opt = getopt(argc, argv, "e:")) != -1) {
    switch(opt) {
```

EXAMPLE 2 Parse suboptions. *(Continued)*

```
    case 'e' :
        while ((index = getsubopt(&optarg, tokens, &value)) != -1) {
            switch(index) {
...
        }
        break;
...
    }
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also

`mount(1M)`, `getopt(3C)`, `attributes(5)`, `standards(5)`

Name

gettext, dgettext, dcgettext, ngettext, dngettext, dcngettext, textdomain, bindtextdomain, bind_textdomain_codeset – message handling functions

Synopsis

Solaris and GNU-compatible

```
#include <libintl.h>

char *gettext(const char *msgid);

char *dgettext(const char *domainname, const char *msgid);

char *textdomain(const char *domainname);

char *bindtextdomain(const char *domainname, const char *dirname);

#include <libintl.h>
#include <locale.h>

char *dcgettext(const char *domainname, const char *msgid,
     int category);
```

GNU-compatible

```
#include <libintl.h>

char *ngettext(const char *msgid1, const char *msgid2,
     unsigned long int n);

char *dngettext(const char *domainname, const char *msgid1,
     const char *msgid2, unsigned long int n);

char *bind_textdomain_codeset(const char *domainname,
     const char *codeset);

extern int _nl_msg_cat_cntr;
extern int *_nl_domain_bindings;

#include <libintl.h>
#include <locale.h>

char *dcngettext(const char *domainname, const char *msgid1,
     const char *msgid2, unsigned long int n, int category);
```

Description

The `gettext()`, `dgettext()`, and `dcgettext()` functions attempt to retrieve a target string based on the specified *msgid* argument within the context of a specific domain and the current locale. The length of strings returned by `gettext()`, `dgettext()`, and `dcgettext()` is undetermined until the function is called. The *msgid* argument is a null-terminated string.

The `ngettext()`, `dngettext()`, and `dcngettext()` functions are equivalent to `gettext()`, `dgettext()`, and `dcgettext()`, respectively, except for the handling of plural forms. These functions work only with GNU-compatible message catalogues. The `ngettext()`, `dngettext()`, and `dcngettext()` functions search for the message string using the *msgid1* argument as the key and the *n* argument to determine the plural form. If no message catalogues are found, *msgid1* is returned if $n == 1$, otherwise *msgid2* is returned.

The NLSPATH environment variable (see environ(5)) is searched first for the location of the LC_MESSAGES catalogue. The setting of the LC_MESSAGES category of the current locale determines the locale used by gettext() and dgettext() for string retrieval. The *category* argument determines the locale used by dcgettext( ). If NLSPATH is not defined and the current locale is "C", gettext(), dgettext(), and dcgettext() simply return the message string that was passed. In a locale other than "C", if NLSPATH is not defined or if a message catalogue is not found in any of the components specified by NLSPATH, the routines search for the message catalogue using the scheme described in the following paragraph.

The LANGUAGE environment variable is examined to determine the GNU-compatible message catalogues to be used. The value of LANGUAGE is a list of locale names separated by a colon (':') character. If LANGUAGE is defined, each locale name is tried in the specified order and if a GNU-compatible message catalogue is found, the message is returned. If a GNU-compatible message catalogue is found but failed to find a corresponding *msgid*, the *msgid* string is return. If LANGUAGE is not defined or if a Solaris message catalogue is found or no GNU-compatible message catalogue is found in processing LANGUAGE, the pathname used to locate the message catalogue is *dirname*/*locale*/*category*/*domainname*.mo, where *dirname* is the directory specified by bindtextdomain(), *locale* is a locale name, and *category* is either LC_MESSAGES if gettext(), dgettext(), ngettext(), or dngettext() is called, or LC_XXX where the name is the same as the locale category name specified by the *category* argument to dcgettext() or dcngettext().

For gettext() and ngettext(), the domain used is set by the last valid call to textdomain(). If a valid call to textdomain() has not been made, the default domain (called messages) is used.

For dgettext(), dcgettext(), dngettext(), and dcngettext(), the domain used is specified by the *domainname* argument. The *domainname* argument is equivalent in syntax and meaning to the *domainname* argument to textdomain(), except that the selection of the domain is valid only for the duration of the dgettext(), dcgettext(), dngettext(), or dcngettext() function call.

The textdomain() function sets or queries the name of the current domain of the active LC_MESSAGES locale category. The *domainname* argument is a null-terminated string that can contain only the characters allowed in legal filenames.

The *domainname* argument is the unique name of a domain on the system. If there are multiple versions of the same domain on one system, namespace collisions can be avoided by using bindtextdomain(). If textdomain() is not called, a default domain is selected. The setting of domain made by the last valid call to textdomain() remains valid across subsequent calls to setlocale(3C), and gettext().

The *domainname* argument is applied to the currently active LC_MESSAGES locale.

The current setting of the domain can be queried without affecting the current state of the domain by calling `textdomain()` with *domainname* set to the null pointer. Calling `textdomain()` with a *domainname* argument of a null string sets the domain to the default domain (`messages`).

The `bindtextdomain()` function binds the path predicate for a message domain *domainname* to the value contained in *dirname*. If *domainname* is a non-empty string and has not been bound previously, `bindtextdomain()` binds *domainname* with *dirname*.

If *domainname* is a non-empty string and has been bound previously, `bindtextdomain()` replaces the old binding with *dirname*. The *dirname* argument can be an absolute or relative pathname being resolved when `gettext()`, `dgettext()`, or `dcgettext()` are called. If *domainname* is a null pointer or an empty string, `bindtextdomain()` returns `NULL`. User defined domain names cannot begin with the string `SYS_`. Domain names beginning with this string are reserved for system use.

The `bind_textdomain_codeset()` function can be used to specify the output codeset for message catalogues for domain *domainname*. The *codeset* argument must be a valid codeset name that can be used for the `iconv_open(3C)` function, or a null pointer. If the *codeset* argument is the null pointer, `bind_textdomain_codeset()` returns the currently selected codeset for the domain with the name *domainname*. It returns a null pointer if a codeset has not yet been selected. The `bind_textdomain_codeset()` function can be used multiple times. If used multiple times with the same *domainname* argument, the later call overrides the settings made by the earlier one. The `bind_textdomain_codeset()` function returns a pointer to a string containing the name of the selected codeset. The string is allocated internally in the function and must not be changed by the user.

The external variables `_nl_msg_cat_cntr` and `_nl_domain_bindings` are provided for the compatibility with the GNU `gettext()` implementation.

Return Values The `gettext()`, `dgettext()`, and `dcgettext()` functions return the message string if the search succeeds. Otherwise they return the *msgid* string.

The `ngettext()`, `dngettext()`, and `dcngettext()` functions return the message string if the search succeeds. If the search fails, *msgid1* is returned if *n* == 1. Otherwise *msgid2* is returned.

The individual bytes of the string returned by `gettext()`, `dgettext()`, `dcgettext()`, `ngettext()`, `dngettext()`, or `dcngettext()` can contain any value other than `NULL`. If *msgid* is a null pointer, the return value is undefined. The string returned must not be modified by the program and can be invalidated by a subsequent call to `bind_textdomain_codeset()` or `setlocale(3C)`. If the *domainname* argument to `dgettext()`,`dcgettext()`, `dngettext()`, or `dcngettext()` is a null pointer, the the domain currently bound by `textdomain()` is used.

The normal return value from `textdomain()` is a pointer to a string containing the current setting of the domain. If *domainname* is a null pointer, `textdomain()` returns a pointer to the string containing the current domain. If `textdomain()` was not previously called and

domainname is a null string, the name of the default domain is returned. The name of the default domain is `messages`. If `textdomain()` fails, a null pointer is returned.

The return value from `bindtextdomain()` is a null-terminated string containing *dirname* or the directory binding associated with *domainname* if *dirname* is `NULL`. If no binding is found, the default return value is `/usr/lib/locale`. If *domainname* is a null pointer or an empty string, `bindtextdomain()` takes no action and returns a null pointer. The string returned must not be modified by the caller. If `bindtextdomain()` fails, a null pointer is returned.

Usage These functions impose no limit on message length. However, a text *domainname* is limited to `TEXTDOMAINMAX` (256) bytes.

The `gettext()`, `dgettext()`, `dcgettext()`, `ngettext()`, `dngettext()`, `dcngettext()`, `textdomain()`, and `bindtextdomain()` functions can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

The `gettext()`, `dgettext()`, `dcgettext()`, `textdomain()`, and `bindtextdomain()` functions work with both Solaris message catalogues and GNU-compatible message catalogues. The `ngettext()`, `dngettext()`, `dcngettext()`, and `bind_textdomain_codeset()` functions work only with GNU-compatible message catalogues. See `msgfmt(1)` for information about Solaris message catalogues and GNU-compatible message catalogues.

Files `/usr/lib/locale`
default path predicate for message domain files

`/usr/lib/locale/`*locale*`/LC_MESSAGES/`*domainname*`.mo`
system default location for file containing messages for language *locale* and *domainname*

`/usr/lib/locale/`*locale*`/LC_XXX/`*domainname*`.mo`
system default location for file containing messages for language *locale* and *domainname* for `dcgettext()` calls where `LC_XXX` is `LC_CTYPE`, `LC_NUMERIC`, `LC_TIME`, `LC_COLLATE`, `LC_MONETARY`, or `LC_MESSAGES`

dirname`/`*locale*`/LC_MESSAGES/`*domainname*`.mo`
location for file containing messages for domain *domainname* and path predicate *dirname* after a successful call to `bindtextdomain()`

dirname`/`*locale*`/LC_XXX/`*domainname*`.mo`
location for files containing messages for domain *domainname,* language *locale,* and path predicate *dirname* after a successful call to `bindtextdomain()` for `dcgettext()` calls where `LC_XXX` is one of `LC_CTYPE`, `LC_NUMERIC`, `LC_TIME`, `LC_COLLATE`, `LC_MONETARY`, or `LC_MESSAGES`

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.

MT-Level	Safe with exceptions

The external variables `_nl_msg_cat_cntr` and `_nl_domain_bindings` are Uncommitted.

See Also `msgfmt`(1), `xgettext`(1), `iconv_open`(3C), `libintl.h`(3HEAD), `setlocale`(3C), `attributes`(5), `environ`(5)

Name gettimeofday, settimeofday – get or set the date and time

Synopsis

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

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

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

Description The `gettimeofday()` function gets and the `settimeofday()` function sets the system's notion of the current time. The current time is expressed in elapsed seconds and microseconds since 00:00 Universal Coordinated Time, January 1, 1970. The resolution of the system clock is hardware dependent; the time may be updated continuously or in clock ticks.

The *tp* argument points to a `timeval` structure, which includes the following members:

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

If *tp* is a null pointer, the current time information is not returned or set.

The `TZ` environment variable holds time zone information. See `TIMEZONE(4)`.

The *tzp* argument to `gettimeofday()` and `settimeofday()` is ignored.

Only privileged processes can set the time of day.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `settimeofday()` function will fail if:

`EINVAL` The structure pointed to by *tp* specifies an invalid time.

`EPERM` The {`PRIV_SYS_TIME`} privilege was not asserted in the effective set of the calling process.

The `gettimeofday()` function will fail for 32-bit interfaces if:

`EOVERFLOW` The system time has progressed beyond 2038, thus the size of the `tv_sec` member of the `timeval` structure pointed to by *tp* is insufficient to hold the current time in seconds.

Usage If the `tv_usec` member of *tp* is > 500000, `settimeofday()` rounds the seconds upward. If the time needs to be set with better than one second accuracy, call `settimeofday()` for the seconds and then `adjtime(2)` for finer accuracy.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	`gettimeofday()` is Standard.

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also adjtime(2), ctime(3C), gethrtime(3C), TIMEZONE(4), attributes(5), privileges(5), standards(5)

Name gettxt – retrieve a text string

Synopsis

```
#include <nl_types.h>

char *gettxt(const char *msgid, const char *dflt_str);
```

Description The `gettxt()` function retrieves a text string from a message file. The arguments to the function are a message identification *msgid* and a default string *dflt_str* to be used if the retrieval fails.

The text strings are in files created by the `mkmsgs` utility (see `mkmsgs(1)`) and installed in directories in `/usr/lib/locale/`*locale*`/LC_MESSAGES`.

The directory `locale` can be viewed as the language in which the text strings are written. The user can request that messages be displayed in a specific language by setting the environment variable `LC_MESSAGES`. If `LC_MESSAGES` is not set, the environment variable `LANG` will be used. If `LANG` is not set, the files containing the strings are in `/usr/lib/locale/C/LC_MESSAGES/*`.

The user can also change the language in which the messages are displayed by invoking the `setlocale(3C)` function with the appropriate arguments.

If `gettxt()` fails to retrieve a message in a specific language it will try to retrieve the same message in U.S. English. On failure, the processing depends on what the second argument *dflt_str* points to. A pointer to the second argument is returned if the second argument is not the null string. If *dflt_str* points to the null string, a pointer to the U.S. English text string "`Message not found!!\n`" is returned.

The following depicts the acceptable syntax of *msgid* for a call to `gettxt()`.

<*msgid*> = <*msgfilename*>:<*msgnumber*>

The first field is used to indicate the file that contains the text strings and must be limited to 14 characters. These characters must be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash) and : (colon). The names of message files must be the same as the names of files created by `mkmsgs` and installed in `/usr/lib/locale/`*locale*`/LC_MESSAGES/*`. The numeric field indicates the sequence number of the string in the file. The strings are numbered from 1 to *n* where *n* is the number of strings in the file.

Return Values Upon failure to pass either the correct *msgid* or a valid message number to `gettxt()`, a pointer to the text string "`Message not found!!\n`" is returned.

Usage It is recommended that `gettext(3C)` be used in place of this function.

Examples **EXAMPLE 1** Example of `gettxt()` function.

In the following example,

```
gettxt("UX:10", "hello world\n")
gettxt("UX:10", "")
```

EXAMPLE 1 Example of `gettxt()` function. *(Continued)*

`UX` is the name of the file that contains the messages and `10` is the message number.

Files

`/usr/lib/locale/C/LC_MESSAGES/*`	contains default message files created by `mkmsgs`
`/usr/lib/locale/`*locale*`/LC_MESSAGES/*`	contains message files for different languages created by `mkmsgs`

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe with exceptions

See Also `exstr(1)`, `mkmsgs(1)`, `srchtxt(1)`, `gettext(3C)`, `fmtmsg(3C)`, `setlocale(3C)`, `attributes(5)`, `environ(5)`

Name getusershell, setusershell, endusershell – get legal user shells

Synopsis

```
#include <unistd.h>

char *getusershell(void);

void setusershell(void);

void endusershell(void);
```

Description The `getusershell()` function returns a pointer to a legal user shell as defined by the system manager in the file `/etc/shells`. If `/etc/shells` does not exist, the following locations of the standard system shells are used in its place:

```
/bin/bash                   /bin/csh
/bin/jsh                    /bin/ksh
/bin/ksh93                  /bin/pfcsh
/bin/pfksh                  /bin/pfsh
/bin/sh                     /bin/tcsh
/bin/zsh                    /sbin/jsh
/sbin/pfsh                  /sbin/sh
/usr/bin/bash               /usr/bin/csh
/usr/bin/jsh                /usr/bin/ksh
/usr/bin/ksh93              /usr/bin/pfcsh
/usr/bin/pfksh              /usr/bin/pfsh
/usr/bin/sh                 /usr/bin/tcsh
/usr/bin/zsh                /usr/sfw/bin/zsh
/usr/xpg4/bin/sh
```

The `getusershell()` function opens the file `/etc/shells`, if it exists, and returns the next entry in the list of shells.

The `setusershell()` function rewinds the file or the list.

The `endusershell()` function closes the file, frees any memory used by `getusershell()` and `setusershell()`, and rewinds the file `/etc/shells`.

Return Values The `getusershell()` function returns a null pointer on EOF.

Bugs All information is contained in memory that may be freed with a call to `endusershell()`, so it must be copied if it is to be saved.

Notes Restricted shells should not be listed in `/etc/shells`.

Name getutent, getutid, getutline, pututline, setutent, endutent, utmpname – user accounting database functions

Synopsis

```
#include <utmp.h>

struct utmp *getutent(void);

struct utmp *getutid(const struct utmp *id);

struct utmp *getutline(const struct utmp *line);

struct utmp *pututline(const struct utmp *utmp);

void setutent(void);

void endutent(void);

int utmpname(const char *file);
```

Description These functions provide access to the user accounting database, `utmp`. Entries in the database are described by the definitions and data structures in `<utmp.h>`.

The `utmp` structure contains the following members:

```
char                ut_user[8];   /* user login name */
char                ut_id[4];     /* /sbin/inittab id */
                                  /* (usually line #) */
char                ut_line[12];  /* device name (console, lnxx) */
short               ut_pid;       /* process id */
short               ut_type;      /* type of entry */
struct exit_status  ut_exit;      /* exit status of a process */
                                  /* marked as DEAD_PROCESS */
time_t              ut_time;      /* time entry was made */
```

The structure `exit_status` includes the following members:

```
short   e_termination;   /* termination status */
short   e_exit;          /* exit status */
```

`getutent()` The `getutent()` function reads in the next entry from a `utmp` database. If the database is not already open, it opens it. If it reaches the end of the database, it fails.

`getutid()` The `getutid()` function searches forward from the current point in the `utmp` database until it finds an entry with a `ut_type` matching *id*->`ut_type` if the type specified is `RUN_LVL`, `BOOT_TIME`, `DOWN_TIME`, `OLD_TIME`, or `NEW_TIME`. If the type specified in *id* is `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS`, or `DEAD_PROCESS`, then `getutid()` will return a pointer to the first entry whose type is one of these four and whose `ut_id` member matches *id*->`ut_id`. If the end of database is reached without a match, it fails.

`getutline()` The `getutline()` function searches forward from the current point in the `utmp` database until it finds an entry of the type `LOGIN_PROCESS` or `ut_line` string matching the *line*->`ut_line` string. If the end of database is reached without a match, it fails.

`pututline()` The `pututline()` function writes the supplied `utmp` structure into the `utmp` database. It uses `getutid()` to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of `pututline()` will have searched for the proper entry using one of the these functions. If so, `pututline()` will not search. If `pututline()` does not find a matching slot for the new entry, it will add a new entry to the end of the database. It returns a pointer to the `utmp` structure. When called by a non-root user, `pututline()` invokes a `setuid()` root program to verify and write the entry, since the `utmp` database is normally writable only by root. In this event, the `ut_name` member must correspond to the actual user name associated with the process; the `ut_type` member must be either `USER_PROCESS` or `DEAD_PROCESS`; and the `ut_line` member must be a device special file and be writable by the user.

`setutent()` The `setutent()` function resets the input stream to the beginning. This reset should be done before each search for a new entry if it is desired that the entire database be examined.

`endutent()` The `endutent()` function closes the currently open database.

`utmpname()` The `utmpname()` function allows the user to change the name of the database file examined to another file. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. The `utmpname()` function does not open the file but closes the old file if it is currently open and saves the new file name.

Return Values

A null pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write. If the file name given is longer than 79 characters, `utmpname()` returns `0`. Otherwise, it returns `1`.

Usage

These functions use buffered standard I/O for input, but `pututline()` uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the `utmp` and `wtmp` databases.

Applications should not access the `utmp` and `wtmp` databases directly, but should use these functions to ensure that these databases are maintained consistently. Using these functions, however, may cause applications to fail if user accounting data cannot be represented properly in the `utmp` structure (for example, on a system where PIDs can exceed 32767). Use the functions described on the `getutxent(3C)` manual page instead.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `getutxent(3C)`, `ttyslot(3C)`, `utmpx(4)`, `attributes(5)`

Notes The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. On each call to either `getutid()` or `getutline()`, the function examines the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason, to use `getutline()` to search for multiple occurrences, it would be necessary to zero out the static area after each success, or `getutline()` would just return the same structure over and over again. There is one exception to the rule about emptying the structure before further reads are done. The implicit read done by `pututline()` (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the `getutent()`, `getutid()` or `getutline()` functions, if the user has just modified those contents and passed the pointer back to `pututline()`.

Name getutxent, getutxid, getutxline, pututxline, setutxent, endutxent, utmpxname, getutmp, getutmpx, updwtmp, updwtmpx – user accounting database functions

Synopsis

```
#include <utmpx.h>

struct utmpx *getutxent(void);

struct utmpx *getutxid(const struct utmpx *id);

struct utmpx *getutxline(const struct utmpx *line);

struct utmpx *pututxline(const struct utmpx *utmpx);

void setutxent(void);

void endutxent(void);

int utmpxname(const char *file);

void getutmp(struct utmpx *utmpx, struct utmp *utmp);

void getutmpx(struct utmp *utmp, struct utmpx *utmpx);

void updwtmp(char *wfile, struct utmp *utmp);

void updwtmpx(char *wfilex, struct utmpx *utmpx);
```

Description These functions provide access to the user accounting database, `utmpx` (see `utmpx(4)`). Entries in the database are described by the definitions and data structures in `<utmpx.h>`.

The `utmpx` structure contains the following members:

```
char                ut_user[32];   /* user login name */
char                ut_id[4];      /* /etc/inittab id */
                                   /* (usually line #) */
char                ut_line[32];   /* device name */
                                   /* (console, lnxx) */
pid_t               ut_pid;        /* process id */
short               ut_type;       /* type of entry */
struct exit_status  ut_exit;       /* exit status of a process */
                                   /* marked as DEAD_PROCESS */
struct timeval      ut_tv;         /* time entry was made */
int                 ut_session;    /* session ID, used for */
                                   /* windowing */
short               ut_syslen;     /* significant length of */
                                   /* ut_host */
                                   /* including terminating null */
char                ut_host[257];  /* host name, if remote */
```

The `exit_status` structure includes the following members:

```
short   e_termination;   /* termination status */
short   e_exit;          /* exit status */
```

`getutxent()` The `getutxent()` function reads in the next entry from a `utmpx` database. If the database is not already open, it opens it. If it reaches the end of the database, it fails.

`getutxid()` The `getutxid()` function searches forward from the current point in the `utmpx` database until it finds an entry with a `ut_type` matching *id*–>`ut_type`, if the type specified is `RUN_LVL`, `BOOT_TIME`, `DOWN_TIME`, `OLD_TIME`, or `NEW_TIME`. If the type specified in *id* is `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS`, or `DEAD_PROCESS`, then `getutxid()` will return a pointer to the first entry whose type is one of these four and whose `ut_id` member matches *id*–>`ut_id`. If the end of database is reached without a match, it fails.

`getutxline()` The `getutxline()` function searches forward from the current point in the `utmpx` database until it finds an entry of the type `LOGIN_PROCESS` or `USER_PROCESS` which also has a *ut_line* string matching the *line*->`ut_line` string. If the end of the database is reached without a match, it fails.

`pututxline()` The `pututxline()` function writes the supplied `utmpx` structure into the `utmpx` database. It uses `getutxid()` to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of `pututxline()` will have searched for the proper entry using one of the `getutx()` routines. If so, `pututxline()` will not search. If `pututxline()` does not find a matching slot for the new entry, it will add a new entry to the end of the database. It returns a pointer to the `utmpx` structure. When called by a non-root user, `pututxline()` invokes a `setuid()` root program to verify and write the entry, since the `utmpx` database is normally writable only by root. In this event, the `ut_name` member must correspond to the actual user name associated with the process; the `ut_type` member must be either `USER_PROCESS` or `DEAD_PROCESS`; and the `ut_line` member must be a device special file and be writable by the user.

`setutxent()` The `setutxent()` function resets the input stream to the beginning. This should be done before each search for a new entry if it is desired that the entire database be examined.

`endutxent()` The `endutxent()` function closes the currently open database.

`utmpxname()` The `utmpxname()` function allows the user to change the name of the database file examined from `/var/adm/utmpx` to any other file, most often `/var/adm/wtmpx`. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. The `utmpxname()` function does not open the file, but closes the old file if it is currently open and saves the new file name. The new file name must end with the "x" character to allow the name of the corresponding `utmp` file to be easily obtainable.; otherwise, an error value of `0` is returned. The function returns `1` on success.

`getutmp()` The `getutmp()` function copies the information stored in the members of the `utmpx` structure to the corresponding members of the `utmp` structure. If the information in any member of `utmpx` does not fit in the corresponding `utmp` member, the data is silently truncated. (See `getutent(3C)` for `utmp` structure)

`getutmpx()` The `getutmpx()` function copies the information stored in the members of the `utmp` structure to the corresponding members of the `utmpx` structure. (See `getutent(3C)` for `utmp` structure)

`updwtmp()` The `updwtmp()` function can be used in two ways.

If *wfile* is `/var/adm/wtmp`, the `utmp` format record supplied by the caller is converted to a `utmpx` format record and the `/var/adm/wtmpx` file is updated (because the `/var/adm/wtmp` file no longer exists, operations on `wtmp` are converted to operations on `wtmpx` by the library functions.

If *wfile* is a file other than `/var/adm/wtmp`, it is assumed to be an old file in `utmp` format and is updated directly with the `utmp` format record supplied by the caller.

`updwtmpx()` The `updwtmpx()` function writes the contents of the `utmpx` structure pointed to by *utmpx* to the database.

`utmpx` structure The values of the `e_termination` and `e_exit` members of the `ut_exit` structure are valid only for records of type `DEAD_PROCESS`. For `utmpx` entries created by `init(1M)`, these values are set according to the result of the `wait()` call that `init` performs on the process when the process exits. See the `wait(3C)`, manual page for the values `init` uses. Applications creating `utmpx` entries can set `ut_exit` values using the following code example:

```
u->ut_exit.e_termination = WTERMSIG(process->p_exit)
u->ut_exit.e_exit = WEXITSTATUS(process->p_exit)
```

See `wait.h(3HEAD)` for descriptions of the `WTERMSIG` and `WEXITSTATUS` macros.

The `ut_session` member is not acted upon by the operating system. It is used by applications interested in creating `utmpx` entries.

For records of type `USER_PROCESS`, the `nonuser()` and `nonuserx()` macros use the value of the `ut_exit.e_exit` member to mark `utmpx` entries as real logins (as opposed to multiple xterms started by the same user on a window system). This allows the system utilities that display users to obtain an accurate indication of the number of actual users, while still permitting each `pty` to have a `utmpx` record (as most applications expect.). The `NONROOT_USER` macro defines the value that `login` places in the `ut_exit.e_exit` member.

Return Values

Upon successful completion, `getutxent()`, `getutxid()`, and `getutxline()` each return a pointer to a `utmpx` structure containing a copy of the requested entry in the user accounting database. Otherwise a null pointer is returned.

The return value may point to a static area which is overwritten by a subsequent call to `getutxid ()` or `getutxline()`.

Upon successful completion, `pututxline()` returns a pointer to a `utmpx` structure containing a copy of the entry added to the user accounting database. Otherwise a null pointer is returned.

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

A null pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

Usage These functions use buffered standard I/O for input, but `pututxline()` uses an unbuffered write to avoid race conditions between processes trying to modify the `utmpx` and `wtmpx` files.

Applications should not access the `utmpx` and `wtmpx` databases directly, but should use these functions to ensure that these databases are maintained consistently.

Files

`/var/adm/utmpx`	user access and accounting information
`/var/adm/wtmpx`	history of user access and accounting information

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	Unsafe

The `endutxent()`, `getutxent()`, `getutxid()`, `getutxline()`, `pututxline()`, and `setutxent()` functions are Standard.

See Also `getutent(3C)`, `ttyslot(3C)`, `wait(3C)`, `wait.h(3HEAD)`, `utmpx(4)`, `attributes(5)`, `standards(5)`

Notes The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. On each call to either `getutxid()` or `getutxline()`, the routine examines the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason, to use `getutxline()` to search for multiple occurrences it would be necessary to zero out the static after each success, or `getutxline()` would just return the same structure over and over again. There is one exception to the rule about emptying the structure before further reads are done. The implicit read done by `pututxline()` (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the `getutxent()`, `getutxid()`, or `getutxline()` routines, if the user has just modified those contents and passed the pointer back to `pututxline()`.

Name getvfsent, getvfsfile, getvfsspec, getvfsany – get vfstab file entry

Synopsis

```
#include <stdio.h>
#include <sys/vfstab.h>

int getvfsent(FILE *fp, struct vfstab *vp);

int getvfsfile(FILE *fp, struct vfstab *vp, char *file);

int getvfsspec(FILE *, struct vfstab *vp, char *spec);

int getvfsany(FILE *, struct vfstab *vp, struct vfstab *vref);
```

Description The `getvfsent()`, `getvfsfile()`, `getvfsspec()`, and `getvfsany()` functions each fill in the structure pointed to by *vp* with the broken-out fields of a line in the `/etc/vfstab` file. Each line in the file contains a `vfstab` structure, declared in the <`sys/vfstab.h`> header, whose following members are described on the `vfstab(4)` manual page:

```
char    *vfs_special;
    char    *vfs_fsckdev;
    char    *vfs_mountp;
    char    *vfs_fstype;
    char    *vfs_fsckpass;
    char    *vfs_automnt;
    char    *vfs_mntopts;
```

The `getvfsent()` function returns a pointer to the next `vfstab` structure in the file; so successive calls can be used to search the entire file.

The `getvfsfile()` function searches the file referenced by *fp* until a mount point matching `file` is found and fills *vp* with the fields from the line in the file.

The `getvfsspec()` function searches the file referenced by *fp* until a special device matching *spec* is found and fills *vp* with the fields from the line in the file. The *spec* argument will try to match on device type (block or character special) and major and minor device numbers. If it cannot match in this manner, then it compares the strings.

The `getvfsany()` function searches the file referenced by *fp* until a match is found between a line in the file and *vref*. A match occurrs if all non-null entries in *vref* match the corresponding fields in the file.

Note that these functions do not open, close, or rewind the file.

Return Values If the next entry is successfully read by `getvfsent()` or a match is found with `getvfsfile()`, `getvfsspec()`, or `getvfsany()`, `0` is returned. If an end-of-file is encountered on reading, these functions return −`1`. If an error is encountered, a value greater than 0 is returned. The possible error values are:

`VFS_TOOLONG` A line in the file exceeded the internal buffer size of `VFS_LINE_MAX`.

`VFS_TOOMANY` A line in the file contains too many fields.

`VFS_TOOFEW` A line in the file contains too few fields.

Files `/etc/vfstab`

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `vfstab(4)`, `attributes(5)`

Notes The members of the `vfstab` structure point to information contained in a static area, so it must be copied if it is to be saved.

Name getwc – get wide character from a stream

Synopsis

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

wint_t getwc(FILE *stream);
```

Description The `getwc()` function is equivalent to `fgetwc(3C)`, except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

Return Values Refer to `fgetwc(3C)`.

Errors Refer to `fgetwc(3C)`.

Usage This interface is provided to align with some current implementations and with possible future ISO standards.

Because it may be implemented as a macro, `getwc()` may treat incorrectly a *stream* argument with side effects. In particular, `getwc(*f++)` may not work as expected. Therefore, use of this function is not recommended; `fgetwc(3C)` should be used instead.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `fgetwc(3C)`, `attributes(5)`, `standards(5)`

Name getwchar – get wide character from stdin stream

Synopsis

```
#include <wchar.h>

wint_t getwchar(void)
```

Description The `getwchar()` function is equivalent to `getwc(stdin)`.

Return Values Refer to `fgetwc`(3C).

Errors Refer to `fgetwc`(3C).

Usage If the `wint_t` value returned by `getwchar()` is stored into a variable of type `wchar_t` and then compared against the `wint_t` macro `WEOF`, the comparison may never succeed because `wchar_t` is defined as unsigned.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `fgetwc`(3C), `getwc`(3C), `attributes`(5), `standards`(5)

Name getwd – get current working directory pathname

Synopsis

```
#include <unistd.h>

char *getwd(char *path_name);
```

Description The `getwd()` function determines an absolute pathname of the current working directory of the calling process, and copies that pathname into the array pointed to by the *path_name* argument.

If the length of the pathname of the current working directory is greater than (`PATH_MAX` + 1) including the null byte, `getwd()` fails and returns a null pointer.

Return Values Upon successful completion, a pointer to the string containing the absolute pathname of the current working directory is returned. Otherwise, `getwd()` returns a null pointer and the contents of the array pointed to by *path_name* are undefined.

Errors No errors are defined.

Usage For portability to implementations conforming to versions of the X/Open Portability Guide prior to SUS, `getcwd(3C)` is preferred over this function.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `getcwd(3C)`, `attributes(5)`, `standards(5)`

Name getwidth – get codeset information

Synopsis

```
#include <euc.h>
#include <getwidth.h>

void getwidth(eucwidth_t *ptr);
```

Description The `getwidth()` function reads the character class table for the current locale to get information on the supplementary codesets. `getwidth()` sets this information into the struct `eucwidth_t`. This struct is defined in <euc.h> and has the following members:

```
short int   _eucw1,_eucw2,_eucw3;
short int   _scrw1,_scrw2,_scrw3;
short int   _pcw;
char        _multibyte;
```

Codeset width values for supplementary codesets 1, 2, and 3 are set in `_eucw1`, `_eucw2`, and `_eucw3`, respectively. Screen width values for supplementary codesets 1, 2, and 3 are set in `_scrw1`, `_scrw2`, and `_scrw3`, respectively.

The width of Extended Unix Code (EUC) Process Code is set in `_pcw`. The `_multibyte` entry is set to `1` if multibyte characters are used, and set to `0` if only single-byte characters are used.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe with exceptions

See Also `euclen(3C)`, `setlocale(3C)`, `attributes(5)`

Notes The `getwidth()` function can be used safely in a multithreaded application, as long as `setlocale(3C)` is not being called to change the locale.

The `getwidth()` function will only work with EUC locales.

Name getws, fgetws – get a wide-character string from a stream

Synopsis

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

wchar_t *getws(wchar_t *ws);

#include <stdio.h>
include <wchar.h>

wchar_t *fgetws(wchar_t *restrict ws, int n, FILE *restrict stream);
```

Description The `getws()` function reads a string of characters from the standard input stream, `stdin`, converts these characters to the corresponding wide-character codes, and writes them to the array pointed to by *ws*, until a newline character is read, converted and transferred to *ws* or an end-of-file condition is encountered. The wide-character string, *ws*, is then terminated with a null wide-character code.

The `fgetws()` function reads characters from the *stream*, converts them to the corresponding wide-character codes, and places them in the `wchar_t` array pointed to by *ws* until *n*−1 characters are read, or until a newline character is read, converted and transferred to *ws*, or an end-of-file condition is encountered. The wide-character string, *ws*, is then terminated with a null wide-character code.

If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

The `fgetws()` function may mark the `st_atime` field of the file associated with *stream* for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc`(3C), `fgets`(3C), `fgetwc`(3C), `fgetws()`, `fread`(3C), `fscanf`(3C), `getc`(3C), `getchar`(3C), `gets`(3C), or `scanf`(3C) using *stream* that returns data not supplied by a prior call to `ungetc`(3C) or `ungetwc`(3C).

Return Values Upon successful completion, `getws()` and `fgetws()` return *ws*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `fgetws()` returns a null pointer. For standard-conforming (see `standards`(5)) applications, if the end-of-file indicator for the stream is set, `fgetws()` returns a null pointer whether or not the stream is at end-of-file. If a read error occurs, the error indicator for the stream is set and `fgetws()` returns a null pointer and sets `errno` to indicate the error.

Errors See `fgetwc`(3C) for the conditions that will cause `fgetws()` to fail.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	`fgetws()` is Standard.

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also ferror(3C), fgetwc(3C), fread(3C), getwc(3C), putws(3C), scanf(3C), ungetc(3C), ungetwc(3C), attributes(5), standards(5)

Name getzoneid, getzoneidbyname, getzonenamebyid – map between zone id and name

Synopsis

```
#include <zone.h>

zoneid_t getzoneid(void);

zoneid_t getzoneidbyname(const char *name);

ssize_t getzonenamebyid(zoneid_t id, char *buf, size_t buflen);
```

Description The `getzoneid()` function returns the zone ID of the calling process.

The `getzoneidbyname()` function returns the zone ID corresponding to the named zone, if that zone is currently active. If *name* is `NULL`, the function returns the zone ID of the calling process.

The `getzonenamebyid()` function stores the name of the zone with ID specified by *id* in the location specified by *buf*. The *bufsize* argument specifies the size in bytes of the buffer. If the buffer is too small to hold the complete null-terminated name, the first *bufsize* bytes of the name are stored in the buffer. A buffer of size {`ZONENAME_MAX`} is sufficient to hold any zone name. If *buf* is `NULL` or *bufsize* is 0, the name is not copied into the buffer.

Return Values On successful completion, `getzoneid()` and `getzoneidbyname()` return a non-negative zone ID. Otherwise, `getzoneidbyname()` returns −1 and sets `errno` to indicate the error.

On successful completion, the `getzonenamebyid()` function returns the buffer size required to hold the full null-terminated name. Otherwise, it returns −1 and sets `errno` to indicate the error.

Errors The `getzoneidbyname()` function will fail if:

`EFAULT` The *name* argument is non-null and points to an illegal address.

`EINVAL` A zone with the indicated *name* is not active.

`ENAMETOOLONG` The length of the *name* argument exceeds {`ZONENAME_MAX`}.

The `getzonenamebyid()` function will fail if:

`EINVAL` A zone with the specified ID is not active.

`EFAULT` The *buf* argument points to an illegal address.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also Intro(2), chroot(2), malloc(3C), attributes(5), zones(5)

Name glob, globfree – generate path names matching a pattern

Synopsis

```
#include <glob.h>

int glob(const char *restrict pattern, int flags,
     int(*errfunc)(const char *epath, int eerrno),
     glob_t *restrict pglob);

void globfree(glob_t *pglob);
```

Description The `glob()` function is a path name generator.

The `globfree()` function frees any memory allocated by `glob()` associated with *pglob*.

pattern Argument The argument *pattern* is a pointer to a path name pattern to be expanded. The `glob()` function matches all accessible path names against this pattern and develops a list of all path names that match. In order to have access to a path name, `glob()` requires search permission on every component of a path except the last, and read permission on each directory of any filename component of *pattern* that contains any of the following special characters:

```
*       ?       [
```

pglob Argument The structure type `glob_t` is defined in the header `<glob.h>` and includes at least the following members:

```
size_t   gl_pathc;     /* count of paths matched by */
                       /* pattern */
char     **gl_pathv;   /* pointer to list of matched */
                       /* path names */
size_t   gl_offs;      /* slots to reserve at beginning */
                       /* of gl_pathv */
```

The `glob()` function stores the number of matched path names into *pglob*–>`gl_pathc` and a pointer to a list of pointers to path names into *pglob*–>`gl_pathv`. The path names are in sort order as defined by the current setting of the `LC_COLLATE` category. The first pointer after the last path name is a `NULL` pointer. If the pattern does not match any path names, the returned number of matched paths is set to 0, and the contents of *pglob*–>`gl_pathv` are implementation-dependent.

It is the caller's responsibility to create the structure pointed to by *pglob*. The `glob()` function allocates other space as needed, including the memory pointed to by `gl_pathv`. The `globfree()` function frees any space associated with *pglob* from a previous call to `glob()`.

flags Argument The *flags* argument is used to control the behavior of `glob()`. The value of *flags* is a bitwise inclusive OR of zero or more of the following constants, which are defined in the header `<glob.h>`:

`GLOB_APPEND` Append path names generated to the ones from a previous call to `glob()`.

GLOB_DOOFFS	Make use of *pglob*–>gl_offs. If this flag is set, *pglob*–>gl_offs is used to specify how many NULL pointers to add to the beginning of *pglob*–>gl_pathv. In other words, *pglob*–>gl_pathv will point to *pglob*–>gl_offs NULL pointers, followed by *pglob*–>gl_pathc path name pointers, followed by a NULL pointer.
GLOB_ERR	Causes glob() to return when it encounters a directory that it cannot open or read. Ordinarily, glob() continues to find matches.
GLOB_MARK	Each path name that is a directory that matches *pattern* has a slash appended.
GLOB_NOCHECK	If *pattern* does not match any path name, then glob() returns a list consisting of only *pattern*, and the number of matched path names is 1.
GLOB_NOESCAPE	Disable backslash escaping.
GLOB_NOSORT	Ordinarily, glob() sorts the matching path names according to the current setting of the LC_COLLATE category. When this flag is used the order of path names returned is unspecified.

The GLOB_APPEND flag can be used to append a new set of path names to those found in a previous call to glob(). The following rules apply when two or more calls to glob() are made with the same value of *pglob* and without intervening calls to globfree():

1. The first such call must not set GLOB_APPEND. All subsequent calls must set it.
2. All the calls must set GLOB_DOOFFS, or all must not set it.
3. After the second call, *pglob*–>gl_pathv points to a list containing the following:
   a. Zero or more NULL pointers, as specified by GLOB_DOOFFS and *pglob*–>gl_offs.
   b. Pointers to the path names that were in the *pglob*–>gl_pathv list before the call, in the same order as before.
   c. Pointers to the new path names generated by the second call, in the specified order.
4. The count returned in *pglob*–>gl_pathc will be the total number of path names from the two calls.
5. The application can change any of the fields after a call to glob(). If it does, it must reset them to the original value before a subsequent call, using the same *pglob* value, to globfree() or glob() with the GLOB_APPEND flag.

errfunc and *epath* Arguments

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not a NULL pointer, glob() calls (**errfunc*) with two arguments:

1. The *epath* argument is a pointer to the path that failed.
2. The *eerrno* argument is the value of *errno* from the failure, as set by the opendir(3C), readdir(3C) or stat(2) functions. (Other values may be used to report other errors not explicitly documented for those functions.)

The following constants are defined as error return values for `glob()`:

`GLOB_ABORTED` The scan was stopped because `GLOB_ERR` was set or `(*errfunc)` returned non-zero.

`GLOB_NOMATCH` The pattern does not match any existing path name, and `GLOB_NOCHECK` was not set in flags.

`GLOG_NOSPACE` An attempt to allocate memory failed.

If `(*errfunc)` is called and returns non-zero, or if the `GLOB_ERR` flag is set in *flags*, `glob()` stops the scan and returns `GLOB_ABORTED` after setting *gl_pathc* and *gl_pathv* in *pglob* to reflect the paths already scanned. If `GLOB_ERR` is not set and either *errfunc* is a `NULL` pointer or `(*errfunc)` returns 0, the error is ignored.

Return Values The following values are returned by `glob()`:

`0` Successful completion. The argument *pglob*–>`gl_pathc` returns the number of matched path names and the argument *pglob*–>`gl_pathv` contains a pointer to a null-terminated list of matched and sorted path names. However, if *pglob*–>`gl_pathc` is 0, the content of *pglob*–>`gl_pathv` is undefined.

`non-zero` An error has occurred. Non-zero constants are defined in <`glob.h`>. The arguments *pglob*–>`gl_pathc` and *pglob*–>`gl_pathv` are still set as defined above.

The `globfree()` function returns no value.

Usage This function is not provided for the purpose of enabling utilities to perform path name expansion on their arguments, as this operation is performed by the shell, and utilities are explicitly not expected to redo this. Instead, it is provided for applications that need to do path name expansion on strings obtained from other sources, such as a pattern typed by a user or read from a file.

If a utility needs to see if a path name matches a given pattern, it can use `fnmatch(3C)`.

Note that `gl_pathc` and `gl_pathv` have meaning even if `glob()` fails. This allows `glob()` to report partial results in the event of an error. However, if `gl_pathc` is 0, `gl_pathv` is unspecified even if `glob()` did not return an error.

The `GLOB_NOCHECK` option could be used when an application wants to expand a path name if wildcards are specified, but wants to treat the pattern as just a string otherwise.

The new path names generated by a subsequent call with `GLOB_APPEND` are not sorted together with the previous path names. This mirrors the way that the shell handles path name expansion when multiple expansions are done on a command line.

Applications that need tilde and parameter expansion should use the wordexp(3C) function.

Examples

EXAMPLE 1 Example of glob_doofs function.

One use of the GLOB_DOOFFS flag is by applications that build an argument list for use with the execv(), execve(), or execvp() functions (see exec(2)). Suppose, for example, that an application wants to do the equivalent of:

```
ls -l *.c
```

but for some reason:

```
system("ls -l *.c")
```

is not acceptable. The application could obtain approximately the same result using the sequence:

```
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] = "-l";
execvp ("ls", &globbuf.gl_pathv[0]);
```

Using the same example:

```
ls -l *.c *.h
```

could be approximately simulated using GLOB_APPEND as follows:

```
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob ("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
. . .
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also

execv(2), stat(2), fnmatch(3C), opendir(3C), readdir(3C), wordexp(3C), attributes(5), standards(5)

Name grantpt – grant access to the slave pseudo-terminal device

Synopsis

```
#include <stdlib.h>

int grantpt(int fildes);
```

Description The `grantpt()` function changes the mode and ownership of the slave pseudo-terminal device associated with its master pseudo-terminal counterpart. *fildes* is the file descriptor returned from a successful open of the master pseudo-terminal device. The user ID of the slave is set to the real UID of the calling process and the group ID is set to a reserved group. The permission mode of the slave pseudo-terminal is set to readable and writable by the owner and writable by the group.

Return Values Upon successful completion, `grantpt()` returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `grantpt()` function may fail if:

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

`EINVAL` The *fildes* argument is not associated with a master pseudo-terminal device.

`EACCES` The corresponding slave pseudo-terminal device could not be accessed.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `open(2)`, `ptsname(3C)`, `unlockpt(3C)`, `attributes(5)`, `standards(5)`

STREAMS Programming Guide

Name hsearch, hcreate, hdestroy – manage hash search tables

Synopsis

```
#include <search.h>

ENTRY *hsearch(ENTRY item, ACTION action);

int hcreate(size_t mekments);

void hdestroy(void);
```

Description The `hsearch()` function is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. The comparison function used by `hsearch()` is `strcmp()` (see `string(3C)`). The *item* argument is a structure of type `ENTRY` (defined in the `<search.h>` header) containing two pointers: `item.key` points to the comparison key, and `item.data` points to any other data to be associated with that key. (Pointers to types other than void should be cast to pointer-to-void.) The *action* argument is a member of an enumeration type `ACTION` (defined in `<search.h>`) indicating the disposition of the entry if it cannot be found in the table. `ENTER` indicates that the item should be inserted in the table at an appropriate point. Given a duplicate of an existing item, the new item is not entered and `hsearch()` returns a pointer to the existing item. `FIND` indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a null pointer.

The `hcreate()` function allocates sufficient space for the table, and must be called before `hsearch()` is used. The *nel* argument is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The `hdestroy()` function destroys the search table, and may be followed by another call to `hcreate()`.

Return Values The `hsearch()` function returns a null pointer if either the action is `FIND` and the item could not be found or the action is `ENTER` and the table is full.

The `hcreate()` function returns `0` if it cannot allocate sufficient space for the table.

Usage The `hsearch()` and `hcreate()` functions use `malloc(3C)` to allocate space.

Only one hash search table may be active at any given time.

Examples **EXAMPLE 1** Example to read in strings.

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it.

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

EXAMPLE 1 Example to read in strings. *(Continued)*

```
#include <stdlib.h>

struct info {              /* this is the info stored in table */
        int age, room;     /* other than the key */
};
#define NUM_EMPL    5000   /* # of elements in search table */
main( )
{
                        /* space to store strings */
        char string_space[NUM_EMPL*20];
                        /* space to store employee info */
        struct info info_space[NUM_EMPL];
                        /* next avail space in string_space */
        char *str_ptr = string_space;
                        /* next avail space in info_space */
        struct info *info_ptr = info_space;
        ENTRY item, *found_item;
                        /* name to look for in table */
        char name_to_find[30];
        int i = 0;

                        /* create table */
        (void) hcreate(NUM_EMPL);
        while (scanf("%s%d%d", str_ptr, &info_ptr->age,
               &info_ptr->room) != EOF && i++ < NUM_EMPL) {
                   /* put info in structure, and structure in item */
                item.key = str_ptr;
                item.data = (void *)info_ptr;
                str_ptr += strlen(str_ptr) + 1;
                info_ptr++;
                        /* put item into table */
                (void) hsearch(item, ENTER);
        }

                        /* access table */
        item.key = name_to_find;
        while (scanf("%s", item.key) != EOF) {
            if ((found_item = hsearch(item, FIND)) != NULL) {
                        /* if item is in the table */
                (void)printf("found %s, age = %d, room = %d\n",
                        found_item->key,
                        ((struct info *)found_item->data)->age,
                        ((struct info *)found_item->data)->room);
            } else {
                (void)printf("no such employee %s\n",
                        name_to_find)
```

EXAMPLE 1 Example to read in strings. *(Continued)*

```
            }
        }
        return 0;
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also

`bsearch(3C)`, `lsearch(3C)`, `malloc(3C)`, `string(3C)`, `tsearch(3C)`, `malloc(3MALLOC)`, `attributes(5)`, `standards(5)`

The Art of Computer Programming, Volume 3, Sorting and Searching by Donald E. Knuth, published by Addison-Wesley Publishing Company, 1973.

Name iconv – code conversion function

Synopsis

Default

```
#include <iconv.h>

extern size_t iconv(iconv_t cd, const char **restrict inbuf,
     size_t *restrict inbytesleft, char **restrict outbuf,
     size_t *restrict outbytesleft);
```

SUSv3

```
#include <iconv.h>

size_t iconv(iconv_t cd, char **restrict inbuf,
     size_t *restrict inbytesleft, char **restrict outbuf,
     size_t *restrict outbytesleft);
```

Description The `iconv()` function converts the sequence of characters from one code set, in the array specified by *inbuf*, into a sequence of corresponding characters in another code set, in the array specified by *outbuf*. The code sets are those specified in the *iconv_open*( ) call that returned the conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first character in the input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer to be converted. The *outbuf* argument points to a variable that points to the first available byte in the output buffer and *outbytesleft* indicates the number of the available bytes to the end of the buffer.

For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When `iconv()` is called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft* points to a positive value, `iconv()` will place, into the output buffer, the byte sequence to change the output buffer to its initial shift state. If the output buffer is not large enough to hold the entire reset sequence, `iconv()` will fail and set `errno` to `E2BIG`. Subsequent calls with *inbuf* as other than a null pointer or a pointer to a null pointer cause the conversion to take place from the current state of the conversion descriptor.

If a sequence of input bytes does not form a valid character in the specified code set, conversion stops after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes. If the output buffer is not large enough to hold the entire converted input, conversion stops just prior to the input bytes that would cause the output buffer to overflow. The variable pointed to by *inbuf* is updated to point to the byte following the last byte successfully used in the conversion. The value pointed to by *inbytesleft* is decremented to reflect the number of bytes still not converted in the input buffer. The variable pointed to by *outbuf* is updated to point to the byte following the last byte of converted output data. The value pointed to by *outbytesleft* is decremented to reflect the number of bytes still available in the output buffer. For state-dependent encodings, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

If `iconv()` encounters a character in the input buffer that is legal, but for which an identical character does not exist in the target code set, `iconv()` performs an implementation-defined conversion on this character.

Return Values The `iconv()` function updates the variables pointed to by the arguments to reflect the extent of the conversion and returns the number of non-identical conversions performed. If the entire string in the input buffer is converted, the value pointed to by *inbytesleft* will be `0`. If the input conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft* will be non-zero and `errno` is set to indicate the condition. If an error occurs `iconv()` returns `(size_t)` `−1` and sets `errno` to indicate the error.

Errors The `iconv()` function will fail if:

`EILSEQ` Input conversion stopped due to an input byte that does not belong to the input code set.

`E2BIG` Input conversion stopped due to lack of space in the output buffer.

`EINVAL` Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

The `iconv()` function may fail if:

`EBADF` The *cd* argument is not a valid open conversion descriptor.

Examples **EXAMPLE 1** Using the `iconv()` Functions

The following example uses the `iconv()` functions:

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <iconv.h>
#include <stdlib.h>

/*
 * For state-dependent encodings, changes the state of the
 * conversion descriptor to initial shift state.  Also, outputs
 * the byte sequence to change the state to initial state.
 * This code is assuming the iconv call for initializing the
 * state will not fail due to lack of space in the output buffer.
 */
#define INIT_SHIFT_STATE(cd, fptr, ileft, tptr, oleft) \
    { \
        fptr = NULL; \
        ileft = 0; \
        tptr = to; \
        oleft = BUFSIZ; \
        (void) iconv(cd, &fptr, &ileft, &tptr, &oleft); \
        (void) fwrite(to, 1, BUFSIZ - oleft, stdout); \
```

EXAMPLE 1 Using the `iconv()` Functions *(Continued)*

```
    }

int
main(int argc, char **argv)
{
    iconv_t cd;
    char    from[BUFSIZ], to[BUFSIZ];
    char    *from_code, *to_code;
    char    *tptr;
    const char  *fptr;
    size_t  ileft, oleft, num, ret;


    if (argc != 3) {
        (void) fprintf(stderr,
            "Usage: %s from_codeset to_codeset\\n", argv[0]);
        return (1);
    }

    from_code = argv[1];
    to_code = argv[2];

    cd = iconv_open((const char *)to_code, (const char *)from_code);
    if (cd == (iconv_t)-1) {
        /*
         * iconv_open failed
         */
        (void) fprintf(stderr,
            "iconv_open(%s, %s) failed\\n", to_code, from_code);
        return (1);
    }

    ileft = 0;
    while ((ileft +=
        (num = fread(from + ileft, 1, BUFSIZ - ileft, stdin))) > 0) {
        if (num == 0) {
            /*
             * Input buffer still contains incomplete character
             * or sequence.  However, no more input character.
             */

            /*
             * Initializes the conversion descriptor and outputs
             * the sequence to change the state to initial state.
             */
```

EXAMPLE 1 Using the `iconv()` Functions *(Continued)*

```
        INIT_SHIFT_STATE(cd, fptr, ileft, tptr, oleft);
        (void) iconv_close(cd);

        (void) fprintf(stderr, "Conversion error\\n");
        return (1);
    }

    fptr = from;
    for (;;) {
        tptr = to;
        oleft = BUFSIZ;

        ret = iconv(cd, &fptr, &ileft, &tptr, &oleft);
        if (ret != (size_t)-1) {
            /*
             * iconv succeeded
             */

            /*
             * Outputs converted characters
             */
            (void) fwrite(to, 1, BUFSIZ - oleft, stdout);
            break;
        }

        /*
         * iconv failed
         */
        if (errno == EINVAL) {
            /*
           * Incomplete character or shift sequence
             */

            /*
             * Outputs converted characters
             */
            (void) fwrite(to, 1, BUFSIZ - oleft, stdout);
            /*
             * Copies remaining characters in input buffer
             * to the top of the input buffer.
             */
            (void) memmove(from, fptr, ileft);
            /*
             * Tries to fill input buffer from stdin
             */
```

EXAMPLE 1 Using the `iconv()` Functions *(Continued)*

```
            break;
        } else if (errno == E2BIG) {
            /*
             * Lack of space in output buffer
             */

            /*
             * Outputs converted characters
             */
            (void) fwrite(to, 1, BUFSIZ - oleft, stdout);
            /*
             * Tries to convert remaining characters in
             * input buffer with emptied output buffer
             */
            continue;
        } else if (errno == EILSEQ) {
            /*
             * Illegal character or shift sequence
             */

            /*
             * Outputs converted characters
             */
            (void) fwrite(to, 1, BUFSIZ - oleft, stdout);
            /*
             * Initializes the conversion descriptor and
             * outputs the sequence to change the state to
             * initial state.
             */
            INIT_SHIFT_STATE(cd, fptr, ileft, tptr, oleft);
            (void) iconv_close(cd);

            (void) fprintf(stderr,
             "Illegal character or sequence\\n");
            return (1);
        } else if (errno == EBADF) {
            /*
             * Invalid conversion descriptor.
             * Actually, this shouldn't happen here.
             */
            (void) fprintf(stderr, "Conversion error\\n");
            return (1);
        } else {
            /*
             * This errno is not defined
```

EXAMPLE 1 Using the `iconv()` Functions *(Continued)*

```
                 */
                (void) fprintf(stderr, "iconv error\\n");
                return (1);
            }
        }
    }

    /*
     * Initializes the conversion descriptor and outputs
     * the sequence to change the state to initial state.
     */
    INIT_SHIFT_STATE(cd, fptr, ileft, tptr, oleft);

    (void) iconv_close(cd);
    return (0);
}
```

Files

`/usr/lib/iconv/*.so`
conversion modules for 32-bit

`/usr/lib/iconv/sparcv9/*.so`
conversion modules for 64-bit sparc

`/usr/lib/iconv/amd64/*.so`
conversion modules for 64-bit amd64

`/usr/lib/iconv/geniconvtbl/binarytables/*.bt`
conversion binary tables

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `geniconvtbl(1)`, `iconv(1)`, `iconv_close(3C)`, `iconv_open(3C)`, `geniconvtbl(4)`, `attributes(5)`, `iconv(5)`, `iconv_unicode(5)`, `standards(5)`

Name iconv_close – code conversion deallocation function

Synopsis

```
#include <iconv.h>

int iconv_close(iconv_t cd);
```

Description The `iconv_close()` function deallocates the conversion descriptor `cd` and all other associated resources allocated by the `iconv_open(3C)` function.

If a file descriptor is used to implement the type `iconv_t`, that file descriptor will be closed.

For examples using the `iconv_close()` function, see `iconv(3C)`.

Return Values Upon successful completion, `iconv_close()` returns `0`; otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `iconv_close()` function may fail if:

`EBADF` The conversion descriptor is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `iconv(3C)`, `iconv_open(3C)`, `attributes(5)`, `standards(5)`

Name iconv_open – code conversion allocation function

Synopsis

```
#include <iconv.h>

iconv_t iconv_open(const char *tocode, const char *fromcode);
```

Description The `iconv_open()` function returns a conversion descriptor that describes a conversion from the codeset specified by the string pointed to by the *fromcode* argument to the codeset specified by the string pointed to by the *tocode* argument. For state-dependent encodings, the conversion descriptor will be in a codeset-dependent initial shift state, ready for immediate use with the `iconv(3C)` function.

Settings of *fromcode* and *tocode* and their permitted combinations are implementation-dependent.

The `iconv_open()` function supports the alias of the encoding name specified in *tocode* and *fromcode*. The alias table of the encoding name is described in the file `/usr/lib/iconv/alias`. See `alias(4)`.

When *fromcode* and *tocode* are indicating the same codeset name and there is no corresponding conversion explicitly defined in the current system, a conversion descriptor for a simple byte-by-byte pass-through iconv code conversion returns as a fallback code conversion.

A conversion descriptor remains valid in a process until that process closes it.

For examples using the `iconv_open()` function, see `iconv(3C)`.

Return Values Upon successful completion `iconv_open()` returns a conversion descriptor for use on subsequent calls to `iconv()`. Otherwise, `iconv_open()` returns `(iconv_t) −1` and sets `errno` to indicate the error.

Errors The `iconv_open` function may fail if:

`EMFILE` `{OPEN_MAX}` files descriptors are currently open in the calling process.

`ENFILE` Too many files are currently open in the system.

`ENOMEM` Insufficient storage space is available.

`EINVAL` The conversion specified by *fromcode* and *tocode* is not supported by the implementation.

Files `/usr/lib/iconv/alias` alias table file of the encoding name

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Standard	See standards(5).

See Also exec(2), iconv(3C), iconv_close(3C), malloc(3C), alias(4), attributes(5), standards(5)

Notes The iconv_open() function uses malloc(3C) to allocate space for internal buffer areas. iconv_open() may fail if there is insufficient storage space to accommodate these buffers.

Portable applications must assume that conversion descriptors are not valid after a call to one of the exec functions (see exec(2)).

With the simple byte-by-byte pass-through iconv code conversion, there is no checking on illegal input byte or incomplete character or shift sequence.

Name imaxabs – return absolute value

Synopsis

```
#include <inttypes.h>

intmax_t imaxabs(intmax_t j);
```

Description The `imaxabs()` function computes the absolute value of an integer *j*. If the result cannot be represented, the behavior is undefined.

Return Values The `imaxabs()` function returns the absolute value.

Errors No errors are defined.

Usage The absolute value of the most negative number cannot be represented in two's complement.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `imaxdiv(3C)`, `attributes(5)`, `standards(5)`

Name imaxdiv – return quotient and remainder

Synopsis

```
#include <inttypes.h>

imaxdiv_t imaxdiv(imaxdiv_t numer, imaxdiv_t denom);
```

Description The `imaxdiv()` function computes *numer* / *denom* and *numer* % *denom* in a single operation.

Return Values The `imaxdiv()` function returns a structure of type `imaxdiv_t`, comprising both the quotient and the remainder. The structure contains (in either order) the members `quot` (the quotient) and `rem` (the remainder), each of which has type `intmax_t`. If either part of the result cannot be represented, the behavior is undefined.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `imaxabs(3C)`, `attributes(5)`, `standards(5)`

Name index, rindex – string operations

Synopsis

```
#include <strings.h>

char *index(const char *s, int c);

char *rindex(const char *s, int c);
```

Description The `index()` and `rindex()` functions operate on null-terminated strings.

The `index()` function returns a pointer to the first occurrence of character *c* in string *s*.

The `rindex()` function returns a pointer to the last occurrence of character *c* in string *s*.

Both `index()` and `rindex()` return 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.

Usage On most modern computer systems, you can *not* use a null pointer to indicate a null string. A null pointer is an error and results in an abort of the program. If you wish to indicate a null string, you must use a pointer that points to an explicit null string. On some machines and with some implementations of the C programming language, a null pointer, if dereferenced, would yield a null string. Though often used, this practice is not always portable. Programmers using a null pointer to represent an empty string should be aware of this portability issue. Even on machines where dereferencing a null pointer does not cause an abort of the program, it does not necessarily yield a null string.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `bstring(3C)`, `malloc(3C)`, `string(3C)`, `attributes(5)`, `standards(5)`

Name initgroups – initialize the supplementary group access list

Synopsis

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

int initgroups(const char *name, gid_t basegid);
```

Description The `initgroups()` function reads the group database to get the group membership for the user specified by *name*, and initializes the supplementary group access list of the calling process (see `getgrnam(3C)` and `getgroups(2)`). The *basegid* group ID is also included in the supplementary group access list. This is typically the real group ID from the user database.

While scanning the group database, if the number of groups, including the *basegid* entry, exceeds `NGROUPS_MAX`, subsequent group entries are ignored.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `initgroups()` function will fail and not change the supplementary group access list if:

`EPERM` The {`PRIV_PROC_SETID`} privilege is not asserted in the effective set of the calling process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `getgroups(2)`, `getgrnam(3C)`, `attributes(5)`

Name insque, remque – insert/remove element from a queue

Synopsis

```
include <search.h>

void insque(struct qelem *elem, struct qelem *pred);

void remque(struct qelem *elem);
```

Description The `insque()` and `remque()` functions manipulate queues built from doubly linked lists. Each element in the queue must be in the following form:

```
struct qelem {
        struct qelem   *q_forw;
        struct qelem   *q_back;
        char           q_data[ ];
};
```

The `insque()` function inserts *elem* in a queue immediately after *pred*. The `remque()` function removes an entry *elem* from a queue.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	See `standards(5)`.

See Also `attributes(5)`, `standards(5)`

Name isaexec – invoke isa-specific executable

Synopsis

```
#include <unistd.h>

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

Description The `isaexec()` function takes the path specified as *path* and breaks it into directory and file name components. It enquires from the running system the list of supported instruction set architectures; see `isalist(5)`. The function traverses the list for an executable file in named subdirectories of the original directory. When such a file is located, `execve()` is invoked with *argv*`[ ]` and *envp*`[ ]`. See `exec(2)`.

Return Values If no file is located, `isaexec()` returns `ENOENT`. Other return values are the same as for `execve()`.

Examples **EXAMPLE 1** Example of `isaexec()` function.

On a system whose `isalist` is

```
sparcv7 sparc
```

the program

```
int
main(int argc, char *argv[], char *envp[])
{
        return (isaexec("/bin/thing", argv, envp));
}
```

will look first for an executable file named `/bin/sparcv7/thing`, then for an executable file named `/bin/sparc/thing`. It will invoke `execve()` on the first executable file it finds named `thing`.

On that same system, a program called `/u/bin/tofu` can cause either `/u/bin/sparcv7/tofu` or `/u/bin/sparc/tofu` to be invoked using the following code:

```
int
main(int argc, char *argv[], char *envp[])
{
              return (isaexec(getexecname(), argv, envp));
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe
Interface Stability	Committed

See Also exec(2), getexecname(3C), attributes(5), isalist(5)

Name isastream – test a file descriptor

Synopsis

```
#include <stropts.h>

int isastream(int fildes);
```

Description The `isastream()` function determines if a file descriptor represents a STREAMS file. The *fildes* argument refers to an open file descriptor.

Return Values Upon successful completion, `isastream()` returns `1` if *fildes* represents a STREAMS file, and `0` if it does not. Otherwise, `−1` is return and `errno` is set to indicate the error.

Errors The `isastream()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `attributes(5)`, , `standards(5)`, `streamio(7I)`

STREAMS Programming Guide

Name isatty – test for a terminal device

Synopsis

```
#include <unistd.h>

int isatty(int fildes);
```

Description The `isatty()` function tests whether *fildes*, an open file descriptor, is associated with a terminal device.

Return Values The `isatty()` function returns `1` if *fildes* is associated with a terminal; otherwise it returns `0` and may set `errno` to indicate the error.

Errors The `isatty()` function may fail if:

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

`ENOTTY` The *fildes* argument is not associated with a terminal.

Usage The `isatty()` function does not necessarily indicate that a human being is available for interaction via *fildes*. It is quite possible that non-terminal devices are connected to the communications line.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `ttyname(3C)`, `attributes(5)`, `standards(5)`

Name isnand, isnanf, finite, fpclass, unordered – determine type of floating-point number

Synopsis

```
#include <ieeefp.h>

int isnand(double dsrc);

int isnanf(float fsrc);

int finite(double dsrc);

fpclass_t fpclass(double dsrc);

int unordered(double dsrc1, double dsrc2);
```

Description The `isnand()` and `isnanf()` functions return `TRUE` (`1`) if the argument *dsrc* or *fsrc* is a NaN; otherwise they return `FALSE` (`0`).

The `fpclass()` function returns one of the following classes to which *dsrc* belongs:

`FP_SNAN`	signaling NaN
`FP_QNAN`	quiet NaN
`FP_NINF`	negative infinity
`FP_PINF`	positive infinity
`FP_NDENORM`	negative denormalized non-zero
`FP_PDENORM`	positive denormalized non-zero
`FP_NZERO`	negative zero
`FP_PZERO`	positive zero
`FP_NNORM`	negative normalized non-zero
`FP_PNORM`	positive normalized non-zero

The `finite()` function returns `TRUE` (`1`) if the argument *dsrc* is neither infinity nor NaN; otherwise it returns `FALSE` (`0`).

The `unordered()` function returns `TRUE` (`1`) if one of its two arguments is unordered with respect to the other argument. This is equivalent to reporting whether either argument is NaN. If neither argument is NaN, `FALSE` (`0`) is returned.

None of these functions generates an exception, even for signaling NaNs.

Return Values See DESCRIPTION.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also fpgetround(3C), isnan(3M), attributes(5)

Name is_system_labeled – determine whether Trusted Extensions software is active

Synopsis

```
#include <tsol/label.h>

int is_system_labeled(void);
```

Description The is_system_labeled function returns TRUE (1) if the Trusted Extensions software is installed and active; otherwise it returns FALSE (0).

Return Values See DESCRIPTION.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also attributes(5)

Name iswalpha, isenglish, isideogram, isnumber, isphonogram, isspecial, iswalnum, iswascii, iswblank, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit – wide-character code classification functions

Synopsis

```
#include <wchar.h>
#include <wctype.h>

int iswalpha(wint_t wc);

int isenglish(wint_t wc);

int isideogram(wint_t wc);

int isnumber(wint_t wc);

int isphonogram(wint_t wc);

int isspecial(wint_t wc);

int iswalnum(wint_t wc);

int iswascii(wint_t wc);

int iswblank(wint_t wc);

int iswcntrl(wint_t wc);

int iswdigit(wint_t wc);

int iswgraph(wint_t wc);

int iswlower(wint_t wc);

int iswprint(wint_t wc);

int iswpunct(wint_t wc);

int iswspace(wint_t wc);

int iswupper(wint_t wc);

int iswxdigit(wint_t wc);
```

Description These functions test whether *wc* is a wide-character code representing a character of a particular class defined in the `LC_CTYPE` category of the current locale.

In all cases, *wc* is a `wint_t`, the value of which must be a wide-character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument has any other values, the behavior is undefined.

`iswalpha(`*wc*`)` Tests whether *wc* is a wide-character code representing a character of class "alpha" in the program's current locale.

`isenglish(`*wc*`)` Tests whether *wc* is a wide-character code representing an English language character, excluding ASCII characters.

`isideogram(`*wc*`)`	Tests whether *wc* is a wide-character code representing an ideographic language character, excluding ASCII characters.
`isnumber(`*wc*`)`	Tests whether *wc* is a wide-character code representing digit [0–9], excluding ASCII characters.
`isphonogram(`*wc*`)`	Tests whether *wc* is a wide-character code representing a phonetic language character, excluding ASCII characters.
`isspecial(`*wc*`)`	Tests whether *wc* is a wide-character code representing a special language character, excluding ASCII characters.
`iswalnum(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class "alpha" or "digit" in the program's current locale.
`iswascii(`*wc*`)`	Tests whether *wc* is a wide-character code representing an ASCII character.
`iswblank(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class “blank” in the program's current locale. This function is not available to applications conforming to standards prior to SUSv3. See `standards(5)`.
`iswlower(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class "lower" in the program's current locale.
`iswcntrl(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class "cntrl" in the program's current locale.
`iswdigit(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class "digit" in the program's current locale.
`iswgraph(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class "graph" in the program's current locale.
`iswprint(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class "print" in the program's current locale.
`iswpunct(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class "punct" in the program's current locale.
`iswspace(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class "space" in the program's current locale.
`iswupper(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class "upper" in the program's current locale.
`iswxdigit(`*wc*`)`	Tests whether *wc* is a wide-character code representing a character of class "xdigit" in the program's current locale.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	See below.
MT-Level	MT-Safe with exceptions

The iswalpha(), iswalnum(), iswblank(), iswcntrl(), iswdigit(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), and iswxdigit() functions are Standard.

See Also localedef(1), setlocale(3C), stdio(3C), ascii(5), attributes(5), standards(5)

Name iswctype – test character for specified class

Synopsis

```
#include <wchar.h>

int iswctype(wint_t wc, wctype_t charclass);
```

Description The `iswctype()` function determines whether the wide-character code *wc* has the character class *charclass*, returning `TRUE` or `FALSE`. The `iswctype()` function is defined on WEOF and wide-character codes corresponding to the valid character encodings in the current locale. If the *wc* argument is not in the domain of the function, the result is undefined. If the value of *charclass* is invalid (that is, not obtained by a call to `wctype`(3C) or *charclass* is invalidated by a subsequent call to `setlocale`(3C) that has affected category `LC_CTYPE`), the result is indeterminate.

Return Values The `iswctype()` function returns `0` for `FALSE` and non-zero for `TRUE`.

Usage There are twelve strings that are reserved for the standard character classes:

"alnum"	"alpha"	"blank"
"cntrl"	"digit"	"graph"
"lower"	"print"	"punct"
"space"	"upper"	"xdigit"

In the table below, the functions in the left column are equivalent to the functions in the right column.

`iswalnum(`*wc*`)`	iswctype(*wc*, `wctype("alnum"))`
iswalpha(*wc*)	iswctype(*wc*, `wctype("alpha"))`
iswcntrl(*wc*)	iswctype(*wc*, `wctype("cntrl"))`
iswdigit(*wc*)	iswctype(*wc*, `wctype("digit"))`
iswgraph(*wc*)	iswctype(*wc*, `wctype("graph"))`
iswlower(*wc*)	iswctype(*wc*, `wctype("lower"))`
iswprint(*wc*)	iswctype(*wc*, `wctype("print"))`
iswpunct(*wc*)	iswctype(*wc*, `wctype("punct"))`
iswspace(*wc*)	iswctype(*wc*, `wctype("space"))`
iswupper(*wc*)	iswctype(*wc*, `wctype("upper"))`
iswxdigit(*wc*)	iswctype(*wc*, `wctype("xdigit"))`

The call

`iswctype(`*wc*`, wctype("blank"))`

does not have an equivalent `isw*( )` function.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe with exceptions

See Also

`iswalpha(3C)`, `setlocale(3C)`, `wctype(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Name killpg – send signal to a process group

Synopsis

```
#include <signal.h>

int killpg(pid_t pgrp, int sig);
```

Description The `killpg()` function sends the signal *sig* to the process group *pgrp*. See `signal.h(3HEAD)` for a list of signals.

The real or effective user ID of the sending process must match the real or saved set-user ID of the receiving process, unless the effective user ID of the sending process is the privileged user. A single exception is the signal `SIGCONT`, which may always be sent to any descendant of the current process.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `killpg()` function will fail and no signal will be sent if:

`EINVAL` The *sig* argument is not a valid signal number.

`EPERM` The effective user ID of the sending process is not privileged user, and neither its real nor effective user ID matches the real or saved set-user ID of one or more of the target processes.

`ESRCH` No processes were found in the specified process group.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `kill(2)`, `setpgrp(2)`, `sigaction(2)`, `signal.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name lckpwdf, ulckpwdf – manipulate shadow password database lock file

Synopsis

```
#include <shadow.h>

int lckpwdf(void);

int ulckpwdf(void);
```

Description The `lckpwdf()` and `ulckpwdf()` functions enable modification access to the password databases through the lock file. A process first uses `lckpwdf()` to lock the lock file, thereby gaining exclusive rights to modify the `/etc/passwd` or `/etc/shadow` password database. See `passwd(4)` and `shadow(4)`. Upon completing modifications, a process should release the lock on the lock file using `ulckpwdf()`. This mechanism prevents simultaneous modification of the password databases. The lock file, `/etc/.pwd.lock`, is used to coordinate modification access to the password databases `/etc/passwd` and `/etc/shadow`.

Return Values If `lckpwdf()` is successful in locking the file within 15 seconds, it returns `0`. If unsuccessful (for example, `/etc/.pwd.lock` is already locked), it returns `−1`.

If `ulckpwdf()` is successful in unlocking the file `/etc/.pwd.lock`, it returns `0`. If unsuccessful (for example, `/etc/.pwd.lock` is already unlocked), it returns `−1`.

Usage These routines are for internal use only; compatibility is not guaranteed.

Files

`/etc/passwd`	password database
`/etc/shadow`	shadow password database
`/etc/.pwd.lock`	lock file

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `getpwnam(3C)`, `getspnam(3C)`, `passwd(4)`, `shadow(4)`, `attributes(5)`

Name lfmt – display error message in standard format and pass to logging and monitoring services

Synopsis

```
#include <pfmt.h>

int lfmt(FILE *stream, long flags, char *format, ... /* arg*/);
```

Description The `lfmt()` function retrieves a format string from a locale-specific message database (unless `MM_NOGET` is specified) and uses it for `printf(3C)` style formatting of *args*. The output is displayed on *stream*. If *stream* is `NULL` no output is displayed.

The `lfmt()` function encapsulates the output in the standard error message format (unless `MM_NOSTD` is specified, in which case the output is like that of `printf()`. It forwards its output to the logging and monitoring facility, even if *stream* is `NULL`. Optionally, `lfmt()` displays the output on the console with a date and time stamp.

If the `printf()` format string is to be retrieved from a message database, the `format` argument must have the following structure:

<*catalog*>:<*msgnum*>:<*defmsg*>.

If `MM_NOGET` is specified, only the <*defmsg*> field must be specified.

The <*catalog*> field indicates the message database that contains the localized version of the format string. This field is limited to 14 characters selected from a set of all characters values, excluding the null character (`\0`) and the ASCII codes for slash (`/`) and colon (`:`).

The <*msgnum*> field is a positive number that indicates the index of the string into the message database.

If the catalog does not exist in the locale (specified by the last call to `setlocale(3C)` using the `LC_ALL` or `LC_MESSAGES` categories), or if the message number is out of bound, `lfmt()` will attempt to retrieve the message from the C locale. If this second retrieval fails, `lfmt()` uses the <*defmsg*> field of the `format` argument.

If <*catalog*> is omitted, `lfmt()` will attempt to retrieve the string from the default catalog specified by the last call to `setcat(3C)`. In this case, the `format` argument has the following structure:

:<*msgnum*>:<*defmsg*>.

The `lfmt()` function will output the message

```
Message not found!!\n
```

as the format string if <*catalog*> is not a valid catalog name, if no catalog is specified (either explicitly or with `setcat()`), if <*msgnum*> is not a valid number, or if no message could be retrieved from the message databases and <*defmsg*> was omitted.

The *flags* argument determines the type of output (whether the `format` should be interpreted as it is or be encapsulated in the standard message format) and the access to message catalogs to retrieve a localized version of `format`.

The *flags* argument is composed of several groups, and can take the following values (one from each group):

Output format control

`MM_NOSTD` Do not use the standard message format but interpret `format` as a `printf()` `format`. Only *catalog access control flags*, *console display control* and *logging information* should be specified if `MM_NOSTD` is used; all other flags will be ignored.

`MM_STD` Output using the standard message format (default value is 0).

Catalog access control

`MM_NOGET` Do not retrieve a localized version of `format`. In this case, only the <*defmsg*> field of `format` is specified.

`MM_GET` Retrieve a localized version of `format` from <*catalog*>, using <*msgid*> as the index and <*defmsg*> as the default message (default value is 0).

Severity (standard message format only)

`MM_HALT` Generate a localized version of `HALT`, but donot halt the machine.

`MM_ERROR` Generate a localized version of `ERROR` (default value is 0).

`MM_WARNING` Generate a localized version of `WARNING`.

`MM_INFO` Generate a localized version of `INFO`.

Additional severities can be defined with the `addsev(3C)` function, using number-string pairs with numeric values in the range [5-255]. The specified severity is formed by the bitwise OR operation of the numeric value and other *flags* arguments.

If the severity is not defined, `lfmt()` uses the string `SEV=`*N* where *N* is the integer severity value passed in *flags*.

Multiple severities passed in *flags* will not be detected as an error. Any combination of severities will be summed and the numeric value will cause the display of either a severity string (if defined) or the string `SEV=`*N* (if undefined).

Action

`MM_ACTION` Specify an action message. Any severity value is superseded and replaced by a localized version of `TO FIX`.

Console display control

`MM_CONSOLE` Display the message to the console in addition to the specified *stream*.

`MM_NOCONSOLE` Do not display the message to the console in addition to the specified *stream* (default value is 0).

Logging information

Major classification	Identify the source of the condition. Identifiers are: `MM_HARD` (hardware), `MM_SOFT` (software), and `MM_FIRM` (firmware).
Message source subclassification	Identify the type of software in which the problem is spotted. Identifiers are: `MM_APPL` (application), `MM_UTIL` (utility), and `MM_OPSYS` (operating system).

Standard Error Message Format

The `lfmt()` function displays error messages in the following format:

label: *severity*: *text*

If no *label* was defined by a call to `setlabel(3C)`, the message is displayed in the format:

severity: *text*

If `lfmt()` is called twice to display an error message and a helpful *action* or recovery message, the output may appear as follows:

label: *severity*: *text*
label: `TO FIX:` *text*

Return Values

Upon successful completion, `lfmt()` returns the number of bytes transmitted. Otherwise, it returns a negative value:

−1 Write the error to *stream*.

−2 Cannot log and/or display at console.

Usage

Since `lfmt()` uses `gettxt(3C)`, it is recommended that `lfmt()` not be used.

Examples

EXAMPLE 1 The following example

```
setlabel("UX:test");
lfmt(stderr, MM_ERROR|MM_CONSOLE|MM_SOFT|MM_UTIL,
     "test:2:Cannot open file: %s\n", strerror(errno));
```

displays the message to `stderr` and to the console and makes it available for logging:

```
UX:test: ERROR: Cannot open file: No such file or directory
```

EXAMPLE 2 The following example

```
setlabel("UX:test");
lfmt(stderr, MM_INFO|MM_SOFT|MM_UTIL,
        "test:23:test facility is enabled\n");
```

displays the message to `stderr` and makes it available for logging:

```
UX:test: INFO: test facility enabled
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also addsev(3C), gettxt(3C), pfmt(3C), printf(3C), setcat(3C), setlabel(3C), setlocale(3C), attributes(5), environ(5)

Name lio_listio – list directed I/O

Synopsis

```
#include <aio.h>

int lio_listio(int mode, struct aiocb *restrict const list[],
     int nent, struct sigevent *restrict sig);
```

Description The `lio_listio()` function allows the calling process, LWP, or thread, to initiate a list of I/O requests within a single function call.

The *mode* argument takes one of the values `LIO_WAIT` or `LIO_NOWAIT` declared in `<aio.h>` and determines whether the function returns when the I/O operations have been completed, or as soon as the operations have been queued. If the *mode* argument is `LIO_WAIT`, the function waits until all I/O is complete and the *sig* argument is ignored.

If the *mode* argument is `LIO_NOWAIT`, the function returns immediately, and asynchronous notification occurs, according to the *sig* argument, when all the I/O operations complete. If *sig* is `NULL`, no asynchronous notification occurs. If *sig* is not `NULL`, asynchronous notification occurs as specified in `signal.h`(3HEAD) when all the requests in *list* have completed.

The I/O requests enumerated by *list* are submitted in an unspecified order.

The *list* argument is an array of pointers to `aiocb` structures. The array contains *nent* elements. The array may contain null elements, which are ignored.

The *aio_lio_opcode* field of each `aiocb` structure specifies the operation to be performed. The supported operations are `LIO_READ`, `LIO_WRITE`, and `LIO_NOP`; these symbols are defined in `<aio.h>`. The `LIO_NOP` operation causes the list entry to be ignored. If the *aio_lio_opcode* element is equal to `LIO_READ`, then an I/O operation is submitted as if by a call to `aio_read`(3C) with the *aiocbp* equal to the address of the `aiocb` structure. If the *aio_lio_opcode* element is equal to `LIO_WRITE`, then an I/O operation is submitted as if by a call to `aio_write`(3C) with the *aiocbp* equal to the address of the `aiocb` structure.

The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.

The *aio_buf* member specifies the address of the buffer to or from which the data is to be transferred.

The *aio_nbytes* member specifies the number of bytes of data to be transferred.

The members of the *aiocb* structure further describe the I/O operation to be performed, in a manner identical to that of the corresponding `aiocb` structure when used by the `aio_read`(3C) and `aio_write`(3C) functions.

The *nent* argument specifies how many elements are members of the list, that is, the length of the array.

The behavior of this function is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file associated with `aio_fildes`. See `fcntl.h`(3HEAD) definitions of `O_DSYNC` and `O_SYNC`.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *aiocbp*→`aio_fildes`.

Return Values If the *mode* argument has the value `LIO_NOWAIT`, and the I/O operations are successfully queued, `lio_listio()` returns `0`; otherwise, it returns −1, and sets `errno` to indicate the error.

If the *mode* argument has the value `LIO_WAIT`, and all the indicated I/O has completed successfully, `lio_listio()` returns `0`; otherwise, it returns −1, and sets `errno` to indicate the error.

In either case, the return value only indicates the success or failure of the `lio_listio()` call itself, not the status of the individual I/O requests. In some cases, one or more of the I/O requests contained in the list may fail. Failure of an individual request does not prevent completion of any other individual request. To determine the outcome of each I/O request, the application must examine the error status associated with each *aiocb* control block. Each error status so returned is identical to that returned as a result of an `aio_read`(3C) or `aio_write`(3C) function.

Errors The `lio_listio()` function will fail if:

`EAGAIN` The resources necessary to queue all the I/O requests were not available. The error status for each request is recorded in the `aio_error` member of the corresponding `aiocb` structure, and can be retrieved using `aio_error`(3C).

`EAGAIN` The number of entries indicated by *nent* would cause the system-wide limit `AIO_MAX` to be exceeded.

`EINVAL` The *mode* argument is an improper value, or the value of *nent* is greater than `AIO_LISTIO_MAX`.

`EINTR` A signal was delivered while waiting for all I/O requests to complete during an `LIO_WAIT` operation. Note that, since each I/O operation invoked by `lio_listio()` may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited. Outstanding I/O requests are not canceled, and the application can use `aio_fsync`(3C) to determine if any request was initiated; `aio_return`(3C) to determine if any request has completed; or `aio_error`(3C) to determine if any request was canceled.

`EIO` One or more of the individual I/O operations failed. The application can use `aio_error`(3C) to check the error status for each `aiocb` structure to determine the individual request(s) that failed.

In addition to the errors returned by the `lio_listio()` function, if the `lio_listio()` function succeeds or fails with errors of `EAGAIN`, `EINTR`, or `EIO`, then some of the I/O specified by the list may have been initiated. If the `lio_listio()` function fails with an error code other than `EAGAIN`, `EINTR`, or `EIO`, no operations from the list have been initiated. The I/O operation indicated by each list element can encounter errors specific to the individual read or write function being performed. In this event, the error status for each `aiocb` control block contains the associated error code. The error codes that can be set are the same as would be set by a `read(2)` or `write(2)` function, with the following additional error codes possible:

`EAGAIN` The requested I/O operation was not queued due to resource limitations.

`ECANCELED` The requested I/O was canceled before the I/O completed due to an explicit `aio_cancel(3C)` request.

`EFBIG` The *aiocbp*→`aio_lio_opcode` is `LIO_WRITE`, the file is a regular file, *aiocbp*→`aio_nbytes` is greater than 0, and the *aiocbp*→`aio_offset` is greater than or equal to the offset maximum in the open file description associated with *aiocbp*→`aio_fildes`.

`EINPROGRESS` The requested I/O is in progress.

`EOVERFLOW` The *aiocbp*→`aio_lio_opcode` is `LIO_READ`, the file is a regular file, *aiocbp*→`aio_nbytes` is greater than 0, and the *aiocbp*→`aio_offset` is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with *aiocbp*→`aio_fildes`.

Usage The `lio_listio()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `close(2)`, `exec(2)`, `exit(2)`, `fork(2)`, `lseek(2)`, `read(2)`, `write(2)`, `aio_cancel(3C)`, `aio_error(3C)`, `aio_fsync(3C)`, `aio_read(3C)`, `aio_return(3C)`, `aio_write(3C)`, `aio.h(3HEAD)`, `fcntl.h(3HEAD)`, `siginfo.h(3HEAD)`, `signal.h(3HEAD)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Name localeconv – get numeric formatting information

Synopsis

```
#include <locale.h>

struct lconv *localeconv(void);
```

Description The `localeconv()` function sets the components of an object with type `struct lconv` (defined in `<locale.h>`) with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale (see `setlocale(3C)`). The definition of `struct lconv` is given below (the values for the fields in the "C" locale are given in comments).

```
char *decimal_point;        /* "." */
char *thousands_sep;        /* "" (zero length string) */
char *grouping;             /* "" */
char *int_curr_symbol;      /* "" */
char *currency_symbol;      /* "" */
char *mon_decimal_point;    /* "" */
char *mon_thousands_sep;    /* "" */
char *mon_grouping;         /* "" */
char *positive_sign;        /* "" */
char *negative_sign;        /* "" */
char int_frac_digits;       /* CHAR_MAX */
char frac_digits;           /* CHAR_MAX */
char p_cs_precedes;         /* CHAR_MAX */
char p_sep_by_space;        /* CHAR_MAX */
char n_cs_precedes;         /* CHAR_MAX */
char n_sep_by_space;        /* CHAR_MAX */
char p_sign_posn;           /* CHAR_MAX*/
char n_sign_posn;           /* CHAR_MAX */
```

The following members are also available to SUSv3–conforming applications. See `standards(5)`

```
char int_p_cs_precedes;     /* CHAR_MAX */
char int_p_sep_by_space;    /* CHAR_MAX */
char int_n_cs_precedes;     /* CHAR_MAX */
char int_n_sep_by_space;    /* CHAR_MAX */
char int_p_sign_posn;       /* CHAR_MAX */
char int_n_sign_posn;       /* CHAR_MAX */
```

The members of the structure with type `char *` are strings, any of which (except `decimal_point`) can point to a null string (""), to indicate that the value is not available in the current locale or is of zero length. The members with type `char` are non-negative numbers, any of which can be `CHAR_MAX` (defined in the `<limits.h>` header) to indicate that the value is not available in the current locale. The members are the following:

`char *decimal_point` The decimal-point character used to format non-monetary quantities.

`char *thousands_sep`	The character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities.
`char *grouping`	A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted non-monetary quantities.
`char *int_curr_symbol`	The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in the ISO 4217: 1995 standard. The fourth character (immediately preceding the null byte) is the character used to separate the international currency symbol from the monetary quantity.
`char *currency_symbol`	The local currency symbol applicable to the current locale.
`char *mon_decimal_point`	The decimal point used to format monetary quantities.
`char *mon_thousands_sep`	The separator for groups of digits to the left of the decimal point in formatted monetary quantities.
`char *mon_grouping`	A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted monetary quantities.
`char *positive_sign`	The string used to indicate a non-negative-valued formatted monetary quantity.
`char *negative_sign`	The string used to indicate a negative-valued formatted monetary quantity.
`char int_frac_digits`	The number of fractional digits (those to the right of the decimal point) to be displayed in an internationally formatted monetary quantity.
`char frac_digits`	The number of fractional digits (those to the right of the decimal point) to be displayed in a formatted monetary quantity.
`char p_cs_precedes`	Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a non-negative formatted monetary quantity.
`char p_sep_by_space`	Set to 0 if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a non-negative formatted monetary quantity. Set to 1 if a space separates the

	symbol from the value; and set to 2 if a space separates the symbol and the sign string, if adjacent.
`char n_cs_precedes`	Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative formatted monetary quantity.
`char n_sep_by_space`	Set to 0 if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a negative formatted monetary quantity. Set to 1 if a space separates the symbol from the value; and set to 2 if a space separates the symbol and the sign string, if adjacent.
`char p_sign_posn`	Set to a value indicating the positioning of the `positive_sign` for a non-negative formatted monetary quantity.
`char n_sign_posn`	Set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity.
`char int_p_cs_precedes`	Set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a non-negative internationally formatted monetary quantity.
`char int_n_cs_precedes`	Set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a negative internationally formatted monetary quantity.
`char int_p_sep_by_space`	Set to a value indicating the separation of the `int_curr_symbol`, the sign string, and the value for a non-negative internationally formatted monetary quantity.
`char int_n_sep_by_space`	Set to a value indicating the separation of the `int_curr_symbol`, the sign string, and the value for a negative internationally formatted monetary quantity.
`char int_p_sign_posn`	Set to a value indicating the positioning of the `positive_sign` for a non-negative internationally formatted monetary quantity.
`char int_n_sign_posn`	Set to a value indicating the positioning of the `negative_sign` for a negative internationally formatted monetary quantity.

The elements of `grouping` and `mon_grouping` are interpreted according to the following:

{`CHAR_MAX`}	No further grouping is to be performed.
0	The previous element is to be repeatedly used for the remainder of the digits.
other	The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits

before the current group.

The values of `p_sep_by_space`, `n_sep_by_space`, `int_p_sep_by_space`, and `int_n_sep_by_space` are interpreted according to the following:

0 No space separates the currency symbol and value.

1 If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.

2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

In an SUSv3–conforming application, for `int_p_sep_by_space` and `int_n_sep_by_space`, the fourth character of `int_curr_symbol` is used instead of a space.

The values of `p_sign_posn`, `n_sign_posn`, `int_p_sign_posn`, and `int_n_sign_posn` are interpreted according to the following:

0 Parentheses surround the quantity and `currency_symbol` or `int_curr_symbol`.

1 The sign string precedes the quantity and `currency_symbol` or `int_curr_symbol`.

2 The sign string succeeds the quantity and `currency_symbol` or `int_curr_symbol`.

3 The sign string immediately precedes the `currency_symbol` or `int_curr_symbol`.

4 The sign string immediately succeeds the `currency_symbol` or `int_curr_symbol`.

Return Values

The `localeconv()` function returns a pointer to the filled-in object. The structure pointed to by the return value may be overwritten by a subsequent call to `localeconv()`.

Examples

EXAMPLE 1 Rules used by four countries to format monetary quantities.

The following table illustrates the rules used by four countries to format monetary quantities.

Country	Positive	Negative	International
Italy (IT)	L.1.234	–L.1.234	ITL.1.234
Netherlands (NE)	F 1.234,56	F –1.234,56	NLG 1.234,56
Norway (NO)	kr1.234,56	kr1.234,56–	NOK 1.234,56
Switzerland (SW)	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

For these four countries, the respective values for the monetary members of the structure returned by `localeconv()` are as follows:

	IT	NE	NO	SW
int_curr_symbol	"ITL."	"NLG "	"NOK "	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2
int_p_cs_precedes	1	1	1	1
int_n_cs_precedes	1	1	1	1
int_p_sep_by_space	0	0	0	0
int_n_sep_by_space	0	0	0	0
int_p_sign_posn	1	1	1	1
int_n_sign_posn	1	4	4	2

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See standards(5).

The `localeconv()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

See Also `setlocale(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Name lockf – record locking on files

Synopsis

```
#include <unistd.h>

int lockf(int fildes, int function, off_t size);
```

Description The `lockf()` function allows sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file (see chmod(2)). Calls to `lockf()` from other threads that attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. See fcntl(2) for more information about record locking.

The *fildes* argument is an open file descriptor. The file descriptor must have `O_WRONLY` or `O_RDWR` permission in order to establish locks with this function call.

The *function* argument is a control value that specifies the action to be taken. The permissible values for *function* are defined in <unistd.h> as follows:

```
#define   F_ULOCK   0   /* unlock previously locked section */
#define   F_LOCK    1   /* lock section for exclusive use */
#define   F_TLOCK   2   /* test & lock section for exclusive use */
#define   F_TEST    3   /* test section for other locks */
```

All other values of *function* are reserved for future extensions and will result in an error if not implemented.

`F_TEST` is used to detect if a lock by another process is present on the specified section. `F_LOCK` and `F_TLOCK` both lock a section of a file if the section is available. `F_ULOCK` removes locks from a section of the file.

The `size` argument is the number of contiguous bytes to be locked or unlocked. The resource to be locked or unlocked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If `size` is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with `F_LOCK` or `F_TLOCK` may, in whole or in part, contain or be contained by a previously locked section for the same process. Locked sections will be unlocked starting at the point of the offset through `size` bytes or to the end of file if `size` is (`off_t`) 0. When this situation occurs, or if this situation occurs in adjacent sections, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

`F_LOCK` and `F_TLOCK` requests differ only by the action taken if the resource is not available. `F_LOCK` blocks the calling thread until the resource is available. `F_TLOCK` causes the function to return −1 and set `errno` to `EAGAIN` if the section is already locked by another process.

File locks are released on first close by the locking process of any file descriptor for the file.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an errno is set to EDEADLK and the requested section is not released.

An F_ULOCK request in which size is non-zero and the offset of the last byte of the requested section is the maximum value for an object of type off_t, when the process has an existing lock in which size is 0 and which includes the last byte of the requested section, will be treated as a request to unlock from the start of the requested section with a size equal to 0. Otherwise, an F_ULOCK request will attempt to unlock only the requested section.

A potential for deadlock occurs if the threads of a process controlling a locked resource is put to sleep by requesting another process's locked resource. Thus calls to lockf() or fcntl(2) scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The alarm(2) function may be used to provide a timeout facility in applications that require this facility.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The lockf() function will fail if:

EBADF	The *fildes* argument is not a valid open file descriptor; or *function* is F_LOCK or F_TLOCK and *fildes* is not a valid file descriptor open for writing.
EACCES or EAGAIN	The *function* argument is F_TLOCK or F_TEST and the section is already locked by another process.
EDEADLK	The *function* argument is F_LOCK and a deadlock is detected.
EINTR	A signal was caught during execution of the function.
ECOMM	The *fildes* argument is on a remote machine and the link to that machine is no longer active.
EINVAL	The *function* argument is not one of F_LOCK, F_TLOCK, F_TEST, or F_ULOCK; or size plus the current file offset is less than 0.
EOVERFLOW	The offset of the first, or if size is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type off_t.

The lockf() function may fail if:

EAGAIN	The *function* argument is F_LOCK or F_TLOCK and the file is mapped with mmap(2).
EDEADLK or ENOLCK	The *function* argument is F_LOCK, F_TLOCK, or F_ULOCK and the request would cause the number of locks to exceed a system-imposed limit.
EOPNOTSUPP or EINVAL	The locking of files of the type indicated by the *fildes* argument is not supported.

Usage Record-locking should not be used in combination with the fopen(3C), fread(3C), fwrite(3C) and other stdio functions. Instead, the more primitive, non-buffered functions (such as open(2)) should be used. Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The stdio functions are the most common source of unexpected buffering.

The alarm(2) function may be used to provide a timeout facility in applications requiring it.

The lockf() function has a transitional interface for 64-bit file offsets. See lf64(5).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also Intro(2), alarm(2), chmod(2), close(2), creat(2), fcntl(2), mmap(2), open(2), read(2), write(2), attributes(5), lf64(5), standards(5)

Name _longjmp, _setjmp – non-local goto

Synopsis

```
#include <setjmp.h>

void _longjmp(jmp_buf env, int val);

int _setjmp(jmp_buf env);
```

Description The `_longjmp()` and `_setjmp()` functions are identical to `longjmp(3C)` and `setjmp(3C)`, respectively, with the additional restriction that `_longjmp()` and `_setjmp()` do not manipulate the signal mask.

If `_longjmp()` is called even though `env` was never initialized by a call to `_setjmp()`, or when the last such call was in a function that has since returned, the results are undefined.

Return Values Refer to `longjmp(3C)` and `setjmp(3C)`.

Errors No errors are defined.

Usage If `_longjmp()` is executed and the environment in which `_setjmp()` was executed no longer exists, errors can occur. The conditions under which the environment of the `_setjmp()` no longer exists include exiting the function that contains the `_setjmp()` call, and exiting an inner block with temporary storage. This condition might not be detectable, in which case the `_longjmp()` occurs and, if the environment no longer exists, the contents of the temporary storage of an inner block are unpredictable. This condition might also cause unexpected process termination. If the function has returned, the results are undefined.

Passing `longjmp()` a pointer to a buffer not created by `setjmp( )`, passing `_longjmp()` a pointer to a buffer not created by `_setjmp()`, passing `siglongjmp(3C)` a pointer to a buffer not created by `sigsetjmp(3C)` or passing any of these three functions a buffer that has been modified by the user can cause all the problems listed above, and more.

The `_longjmp()` and `_setjmp()` functions are included to support programs written to historical system interfaces. New applications should use `siglongjmp(3C)` and `sigsetjmp(3C)` respectively.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `longjmp(3C)`, `setjmp(3C)`, `siglongjmp(3C)`, `sigsetjmp(3C)`, `attributes(5)`, `standards(5)`

Name lsearch, lfind – linear search and update

Synopsis

```
#include <search.h>

void *lsearch(const void *key, void *base, size_t *nelp,
     size_t width, int (*compar)(const void *, const void *));

void *lfind(const void *key, const void *base, size_t *nelp,
     size_t width, int (*compar)(const void *, const void *));
```

Description The `lsearch()` function is a linear search routine generalized from Knuth (6.1) Algorithm S. (see *The Art of Computer Programming, Volume 3, Section 6.1, by Donald E. Knuth.*). It returns a pointer to a table indicating where a datum can be found. If the datum does not occur, it is added at the end of the table. The *key* argument points to the datum to be sought in the table. The *base* argument points to the first element in the table. The *nelp* argument points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. The *width* argument is the size of an element in bytes. The *compar* argument is a pointer to the comparison function that the user must supply (`strcmp(3C)` for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

The `lfind()` function is the same as `lsearch()` except that if the datum is not found, it is not added to the table. Instead, a null pointer is returned.

It is important to note the following:

- The pointers to the key and the element at the base of the table can be pointers to any type.
- The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.
- The value returned should be cast into type pointer-to-element.

Return Values If the searched-for datum is found, both `lsearch()` and `lfind()` return a pointer to it. Otherwise, `lfind()` returns `NULL` and `lsearch()` returns a pointer to the newly added element.

Usage Undefined results can occur if there is not enough room in the table to add a new item.

The `lsearch()` and `lfind()` functions safely allows concurrent access by multiple threads to disjoint data, such as overlapping subtrees or tables.

Examples **EXAMPLE 1** A sample code using the `lsearch()` function.

This program will read in less than `TABSIZE` strings of length less than `ELSIZE` and store them in a table, eliminating duplicates, and then will print each entry.

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

EXAMPLE 1 A sample code using the `lsearch()` function. *(Continued)*

```
#define TABSIZE 50
#define ELSIZE 120

main()
{
    char line[ELSIZE];          /* buffer to hold input string */
    char tab[TABSIZE][ELSIZE];  /* table of strings */
    size_t nel = 0;             /* number of entries in tab */
    int i;

    while (fgets(line, ELSIZE, stdin) != NULL &&
            nel < TABSIZE)
            (void) lsearch(line, tab, &nel, ELSIZE, mycmp);
    for( i = 0; i < nel; i++ )
            (void)fputs(tab[i], stdout);
    return 0;
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also

`bsearch(3C)`, `hsearch(3C)`, `string(3C)`, `tsearch(3C)`, `attributes(5)`, `standards(5)`

The Art of Computer Programming, Volume 3, Sorting and Searching by Donald E. Knuth, published by Addison-Wesley Publishing Company, 1973.

Name madvise – provide advice to VM system

Synopsis

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

int madvise(caddr_t addr, size_t len, int advice);
```

Description The `madvise()` function advises the kernel that a region of user mapped memory in the range [*addr*, *addr* + *len*) will be accessed following a type of pattern. The kernel uses this information to optimize the procedure for manipulating and maintaining the resources associated with the specified mapping range.

Values for *advice* are defined in <`sys/mman.h`> as:

```
#define MADV_NORMAL           0x0  /* No further special treatment */
#define MADV_RANDOM           0x1  /* Expect random page references */
#define MADV_SEQUENTIAL       0x2  /* Expect sequential page references */
#define MADV_WILLNEED         0x3  /* Will need these pages */
#define MADV_DONTNEED         0x4  /* Don't need these pages */
#define MADV_FREE             0x5  /* Contents can be freed */
#define MADV_ACCESS_DEFAULT   0x6  /* default access */
#define MADV_ACCESS_LWP       0x7  /* next LWP to access heavily */
#define MADV_ACCESS_MANY      0x8  /* many processes to access heavily */
```

`MADV_NORMAL` This is the default system characteristic where accessing memory within the address range causes the system to read data from the mapped file. The kernel reads all data from files into pages which are retained for a period of time as a "cache." System pages can be a scarce resource, so the kernel steals pages from other mappings when needed. This is a likely occurrence, but adversely affects system performance only if a large amount of memory is accessed.

`MADV_RANDOM` Tell the kernel to read in a minimum amount of data from a mapped file on any single particular access. If `MADV_NORMAL` is in effect when an address of a mapped file is accessed, the system tries to read in as much data from the file as reasonable, in anticipation of other accesses within a certain locality.

`MADV_SEQUENTIAL` Tell the system that addresses in this range are likely to be accessed only once, so the system will free the resources mapping the address range as quickly as possible.

`MADV_WILLNEED` Tell the system that a certain address range is definitely needed so the kernel will start reading the specified range into memory. This can benefit programs wanting to minimize the time needed to access memory the first time, as the kernel would need to read in from the file.

`MADV_DONTNEED`	Tell the kernel that the specified address range is no longer needed, so the system starts to free the resources associated with the address range.
`MADV_FREE`	Tell the kernel that contents in the specified address range are no longer important and the range will be overwritten. When there is demand for memory, the system will free pages associated with the specified address range. In this instance, the next time a page in the address range is referenced, it will contain all zeroes. Otherwise, it will contain the data that was there prior to the `MADV_FREE` call. References made to the address range will not make the system read from backing store (swap space) until the page is modified again. This value cannot be used on mappings that have underlying file objects.
`MADV_ACCESS_LWP`	Tell the kernel that the next LWP to touch the specified address range will access it most heavily, so the kernel should try to allocate the memory and other resources for this range and the LWP accordingly.
`MADV_ACCESS_MANY`	Tell the kernel that many processes and/or LWPs will access the specified address range randomly across the machine, so the kernel should try to allocate the memory and other resources for this range accordingly.
`MADV_ACCESS_DEFAULT`	Reset the kernel's expectation for how the specified range will be accessed to the default.

The `madvise()` function should be used by applications with specific knowledge of their access patterns over a memory object, such as a mapped file, to increase system performance.

Return Values Upon successful completion, `madvise()` returns `0`; otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors

`EAGAIN`	Some or all mappings in the address range [*addr*, *addr* + *len*) are locked for I/O.
`EBUSY`	Some or all of the addresses in the range [*addr*, *addr* + *len*) are locked and `MS_SYNC` with the `MS_INVALIDATE` option is specified.
`EFAULT`	Some or all of the addresses in the specified range could not be read into memory from the underlying object when performing `MADV_WILLNEED`. The `madvise()` function could return prior to this condition being detected, in which case `errno` will not be set to `EFAULT`.

EINVAL	The *addr* argument is not a multiple of the page size as returned by sysconf(3C), the length of the specified address range is equal to 0, or the *advice* argument was invalid.
EIO	An I/O error occurred while reading from or writing to the file system.
ENOMEM	Addresses in the range [*addr*, *addr* + *len*) are outside the valid range for the address space of a process, or specify one or more pages that are not mapped.
ESTALE	Stale NFS file handle.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also meminfo(2), mmap(2), sysconf(3C), attributes(5)

Name makecontext, swapcontext – manipulate user contexts

Synopsis

```
#include <ucontext.h>

void makecontext(ucontext_t *ucp, void (*func)(), int argc...);

int swapcontext(ucontext_t *restrict oucp,
     const ucontext_t *restrict ucp);
```

Description The `makecontext()` function modifies the context specified by *ucp*, which has been initialized using `getcontext(2)`. When this context is resumed using `swapcontext()` or `setcontext(2)`, execution continues by calling the function *func*, passing it the arguments that follow *argc* in the `makecontext()` call. The value of *argc* must match the number of pointer-sized integer arguments passed to *func*, otherwise the behavior is undefined.

Before a call is made to `makecontext()`, the context being modified should have a stack allocated for it. The stack is assigned to the context by initializing the `uc_stack` member.

The `uc_link` member is used to determine the context that will be resumed when the context being modified by `makecontext()` returns. The `uc_link` member should be initialized prior to the call to `makecontext()`. If the `uc_link` member is initialized to `NULL`, the thread executing *func* will exit when *func* returns. See `pthread_exit(3C)`.

The `swapcontext()` function saves the current context in the context structure pointed to by *oucp* and sets the context to the context structure pointed to by *ucp*.

If the *ucp* or *oucp* argument points to an invalid address, the behavior is undefined and `errno` may be set to `EFAULT`.

Return Values On successful completion, `swapcontext()` returns `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `swapcontext()` function will fail if:

`ENOMEM` The *ucp* argument does not have enough stack left to complete the operation.

The `swapcontext()` function may fail if:

`EFAULT` The *ucp* or *oucp* argument points to an invalid address.

Examples **EXAMPLE 1** Alternate execution context on a stack whose memory was allocated using `mmap()`.

```
#include <stdio.h>
#include <ucontext.h>
#include <sys/mman.h>

void
assign(long a, int *b)
{
        *b = (int)a;
```

EXAMPLE 1 Alternate execution context on a stack whose memory was allocated using `mmap()`. *(Continued)*

```
}

int
main(int argc, char **argv)
{
        ucontext_t uc, back;
        size_t sz = 0x10000;
        int value = 0;

        getcontext(&uc);

        uc.uc_stack.ss_sp = mmap(0, sz,
            PROT_READ | PROT_WRITE | PROT_EXEC,
            MAP_PRIVATE | MAP_ANON, -1, 0);
        uc.uc_stack.ss_size = sz;
        uc.uc_stack.ss_flags = 0;

        uc.uc_link = &back;

        makecontext(&uc, assign, 2, 100L, &value);
        swapcontext(&back, &uc);

        printf("done %d\n", value);

        return (0);
}
```

Usage

These functions are useful for implementing user-level context switching between multiple threads of control within a process (co-processing). More effective multiple threads of control can be obtained by using native support for multithreading. See `threads(5)`.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also

`mmap(2)`, `getcontext(2)`, `sigaction(2)`, `sigprocmask(2)`, `pthread_exit(3C)`, `ucontext.h(3HEAD)`, `attributes(5)`, `standards(5)`, `threads(5)`

Notes The semantics of the `uc_stack` member of the `ucontext_t` structure have changed as they apply to inputs to `makecontext()`. Prior to Solaris 10, the `ss_sp` member of the `uc_stack` structure represented the high memory address of the area reserved for the stack. The `ss_sp` member now represents the base (low memory address), in keeping with other uses of `ss_sp`.

This change in the meaning of `ss_sp` is now the default behavior. The `-D__MAKECONTEXT_V2_SOURCE` compilation flag used in Solaris 9 update releases to access this behavior is obsolete.

Binary compatibility has been preserved with releases prior to Solaris 10. Before recompiling, applications that use `makecontext()` must be updated to reflect this behavior change. The example below demonstates a typical change that must be applied:

```
--- example1_s9.c       Thu Oct  3 11:58:17 2002
+++ example1.c  Thu Jun 27 13:28:16 2002
@@ -27,12 +27,9 @@
        uc.uc_stack.ss_sp = mmap(0, sz,
            PROT_READ | PROT_WRITE | PROT_EXEC,
            MAP_PRIVATE | MAP_ANON, -1, 0);
-       uc.uc_stack.ss_sp = (char *)uc.uc_stack.ss_sp + sz - 8;
        uc.uc_stack.ss_size = sz;
        uc.uc_stack.ss_flags = 0;

        uc.uc_link = &back

        makecontext(&uc, assign, 2, 100L, &value);
```

Name makedev, major, minor – manage a device number

Synopsis

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

dev_t makedev(major_t maj, minor_t min);

major_t major(dev_t device);

minor_t minor(dev_t device);
```

Description The `makedev()` function returns a formatted device number on success and `NODEV` on failure. The *maj* argument is the major number. The *min* argument is the minor number. The `makedev()` function can be used to create a device number for input to `mknod(2)`.

The `major()` function returns the major number component from *device*.

The `minor()` function returns the minor number component from *device*.

Return Values Upon successful completion, `makedev()` returns a formatted device number. Otherwise, `NODEV` is returned and `errno` is set to indicate the error.

Errors The `makedev()` function will fail if:

`EINVAL` One or both of the arguments *maj* and *min* is too large, or the *device* number created from *maj* and *min* is `NODEV`.

The `major()` function will fail if:

`EINVAL` The *device* argument is `NODEV`, or the major number component of *device* is too large.

The `minor()` function will fail if:

`EINVAL` The *device* argument is `NODEV`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `mknod(2)`, `stat(2)`, `attributes(5)`

Name malloc, calloc, free, memalign, realloc, valloc, alloca – memory allocator

Synopsis

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nelem, size_t elsize);

void free(void *ptr);

void *memalign(size_t alignment, size_t size);

void *realloc(void *ptr, size_t size);

void *valloc(size_t size);

#include <alloca.h>

void *alloca(size_t size);
```

Description The `malloc()` and `free()` functions provide a simple, general-purpose memory allocation package. The `malloc()` function returns a pointer to a block of at least *size* bytes suitably aligned for any use. If the space assigned by `malloc()` is overrun, the results are undefined.

The argument to `free()` is a pointer to a block previously allocated by `malloc()`, `calloc()`, or `realloc()`. After `free()` is executed, this space is made available for further allocation by the application, though not returned to the system. Memory is returned to the system only upon termination of the application. If *ptr* is a null pointer, no action occurs. If a random number is passed to `free()`, the results are undefined.

The `calloc()` function allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

The `memalign()` function allocates *size* bytes on a specified alignment boundary and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of *alignment*. The value of *alignment* must be a power of two and must be greater than or equal to the size of a word.

The `realloc()` function changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If the new size of the block requires movement of the block, the space for the previous instantiation of the block is freed. If the new size is larger, the contents of the newly allocated portion of the block are unspecified. If *ptr* is `NULL`, `realloc()` behaves like `malloc()` for the specified size. If *size* is 0 and *ptr* is not a null pointer, the space pointed to is freed.

The `valloc()` function has the same effect as `malloc()`, except that the allocated memory will be aligned to a multiple of the value returned by `sysconf(_SC_PAGESIZE)`.

The `alloca()` function allocates *size* bytes of space in the stack frame of the caller, and returns a pointer to the allocated block. This temporary space is automatically freed when the caller returns. If the allocated block is beyond the current stack limit, the resulting behavior is undefined.

Return Values Upon successful completion, each of the allocation functions returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

If there is no available memory, `malloc()`, `realloc()`, `memalign()`, `valloc()`, and `calloc()` return a null pointer. When `realloc()` is called with *size* > 0 and returns `NULL`, the block pointed to by *ptr* is left intact. If *size*, *nelem*, or *elsize* is 0, either a null pointer or a unique pointer that can be passed to `free()` is returned.

If `malloc()`, `calloc()`, or `realloc()` returns unsuccessfully, `errno` will be set to indicate the error. The `free()` function does not set `errno`.

Errors The `malloc()`, `calloc()`, and `realloc()` functions will fail if:

`ENOMEM` The physical limits of the system are exceeded by *size* bytes of memory which cannot be allocated.

`EAGAIN` There is not enough memory available to allocate *size* bytes of memory; but the application could try again later.

Usage Portable applications should avoid using `valloc()` but should instead use `malloc()` or `mmap(2)`. On systems with a large page size, the number of successful `valloc()` operations might be 0.

These default memory allocation routines are safe for use in multithreaded applications but are not scalable. Concurrent accesses by multiple threads are single-threaded through the use of a single lock. Multithreaded applications that make heavy use of dynamic memory allocation should be linked with allocation libraries designed for concurrent access, such as `libumem(3LIB)` or `libmtmalloc(3LIB)`. Applications that want to avoid using heap allocations (with `brk(2)`) can do so by using either `libumem` or `libmapmalloc(3LIB)`. The allocation libraries `libmalloc(3LIB)` and `libbsdmalloc(3LIB)` are available for special needs.

Comparative features of the various allocation libraries can be found in the `umem_alloc(3MALLOC)` manual page.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See below.

For `malloc()`, `calloc()`, `free()`, `realloc()`, and `valloc()`, see `standards(5)`.

See Also `brk(2)`, `getrlimit(2)`, `libbsdmalloc(3LIB)`, `libmalloc(3LIB)`, `libmapmalloc(3LIB)`, `libmtmalloc(3LIB)`, `libumem(3LIB)`, `umem_alloc(3MALLOC)`, `watchmalloc(3MALLOC)`, `attributes(5)`, `standards(5)`

Warnings Undefined results will occur if the size requested for a block of memory exceeds the maximum size of a process's heap, which can be obtained with `getrlimit(2)`

The `alloca()` function is machine-, compiler-, and most of all, system-dependent. Its use is strongly discouraged.

Name malloc, free, memalign, realloc, valloc, calloc, mallopt, mallinfo – memory allocator

Synopsis

```
cc [ flag ... ] file ... -lmalloc [ library ... ]
#include <stdlib.h>

void *malloc(size_t size);

void free(void *ptr);

void *memalign(size_t alignment, size_t size);

void *realloc(void *ptr, size_t size);

void *valloc(size_t size);

void *calloc(size_t nelem, size_t elsize);

#include <malloc.h>

int mallopt(int cmd, int value);

struct mallinfo mallinfo(void);
```

Description The `malloc()` and `free()` functions provide a simple general-purpose memory allocation package.

The `malloc()` function returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to `free()` is a pointer to a block previously allocated by `malloc()`. After `free()` is performed, this space is made available for further allocation, and its contents have been destroyed. See `mallopt()` below for a way to change this behavior. If *ptr* is a null pointer, no action occurs.

Undefined results occur if the space assigned by `malloc()` is overrun or if some random number is handed to `free()`.

The `free()` function does not set `errno`.

The `memalign()` function allocates *size* bytes on a specified alignment boundary and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of *alignment*. The value of *alignment* must be a power of two and must be greater than or equal to the size of a word.

The `realloc()` function changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If the new size of the block requires movement of the block, the space for the previous instantiation of the block is freed. If the new size is larger, the contents of the newly allocated portion of the block are unspecified. If *ptr* is `NULL`, `realloc()` behaves like `malloc()` for the specified size. If *size* is 0 and *ptr* is not a null pointer, the space pointed to is freed.

The `valloc()` function has the same effect as `malloc()`, except that the allocated memory will be aligned to a multiple of the value returned by `sysconf(_SC_PAGESIZE)`.

The `calloc()` function allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

The `mallopt()` function provides for control over the allocation algorithm. The available values for *cmd* are:

`M_MXFAST` Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 24.

`M_NLBLKS` Set *numlblks* to *value*. The above mentioned “large groups" each contain *numlblks* blocks. *numlblks* must be greater than 0. The default value for *numlblks* is 100.

`M_GRAIN` Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *grain* must be greater than 0. The default value of *grain* is the smallest number of bytes that will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set.

`M_KEEP` Preserve data in a freed block until the next `malloc()`, `realloc()`, or `calloc()`. This option is provided only for compatibility with the old version of `malloc()`, and it is not recommended.

These values are defined in the <`malloc.h`> header.

The `mallopt()` function can be called repeatedly, but cannot be called after the first small block is allocated.

The `mallinfo()` function provides instrumentation describing space usage. It returns the `mallinfo` structure with the following members:

```
unsigned long arena;      /* total space in arena */
unsigned long ordblks;    /* number of ordinary blocks */
unsigned long smblks;     /* number of small blocks */
unsigned long hblkhd;     /* space in holding block headers */
unsigned long hblks;      /* number of holding blocks */
unsigned long usmblks;    /* space in small blocks in use */
unsigned long fsmblks;    /* space in free small blocks */
unsigned long uordblks;   /* space in ordinary blocks in use */
unsigned long fordblks;   /* space in free ordinary blocks */
unsigned long keepcost;   /* space penalty if keep option */
                          /* is used */
```

The `mallinfo` structure is defined in the <`malloc.h`> header.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

Return Values The `malloc()`, `memalign()`, `realloc()`, `valloc()`, and `calloc()` functions return a null pointer if there is not enough available memory. When `realloc()` returns `NULL`, the block pointed to by *ptr* is left intact. If *size*, *nelem*, or *elsize* is 0, either a null pointer or a unique pointer that can be passed to `free()` is returned. If `mallopt()` is called after any allocation or if *cmd* or *value* are invalid, a non-zero value is returned. Otherwise, it returns 0.

Errors If `malloc()`, `calloc()`, or `realloc()` returns unsuccessfully, `errno` is set to indicate the error:

`ENOMEM` *size* bytes of memory exceeds the physical limits of your system, and cannot be allocated.

`EAGAIN` There is not enough memory available at this point in time to allocate *size* bytes of memory; but the application could try again later.

Usage Unlike `malloc(3C)`, this package does not preserve the contents of a block when it is freed, unless the `M_KEEP` option of `mallopt()` is used.

Undocumented features of `malloc(3C)` have not been duplicated.

Function prototypes for `malloc()`, `realloc()`, `calloc()`, and `free()` are also defined in the `<malloc.h>` header for compatibility with old applications. New applications should include `<stdlib.h>` to access the prototypes for these functions.

Comparative features of the various allocation libraries can be found in the `umem_alloc(3MALLOC)` manual page.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `brk(2)`, `bsdmalloc(3MALLOC)`, `libmtmalloc(3LIB)`, `malloc(3C)`, `mapmalloc(3MALLOC)`, `mtmalloc(3MALLOC)`, `umem_alloc(3MALLOC)`, `watchmalloc(3MALLOC)`, `attributes(5)`

Name mapmalloc – memory allocator

Synopsis

```
cc [ flag ... ] file ... -lmapmalloc [ library ... ]
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nelem, size_t elsize);

void free(void * ptr);

void *realloc(void *ptr, size_t size);
```

Description The collection of `malloc` functions in this library use `mmap(2)` instead of `sbrk(2)` for acquiring new heap space. The functions in this library are intended to be used only if necessary, when applications must call `sbrk()`, but need to call other library routines that might call `malloc`. The algorithms used by these functions are not sophisticated. There is no reclaiming of memory.

The `malloc()` and `free()` functions provide a simple general-purpose memory allocation package.

The `malloc()` function returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to `free()` is a pointer to a block previously allocated by `malloc()`, `calloc()` or `realloc()`. If *ptr* is a `NULL` pointer, no action occurs.

Undefined results will occur if the space assigned by `malloc()` is overrun or if some random number is handed to `free()`.

The `calloc()` function allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

The `realloc()` function changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If the new size of the block requires movement of the block, the space for the previous instantiation of the block is freed. If the new size is larger, the contents of the newly allocated portion of the block are unspecified. If *ptr* is `NULL`, `realloc()` behaves like `malloc()` for the specified size. If *size* is 0 and *ptr* is not a null pointer, the space pointed to is freed.

Each of the allocation functions returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

The `malloc()` and `realloc()` functions will fail if there is not enough available memory.

Entry points for `malloc_debug()`, `mallocmap()`, `mallopt()`, `mallinfo()`, `memalign()`, and `valloc()` are empty routines, and are provided only to protect the user from mixing `malloc()` functions from different implementations.

Return Values If there is no available memory, `malloc()`, `realloc()`, and `calloc()` return a null pointer. When `realloc()` returns `NULL`, the block pointed to by *ptr* is left intact. If *size*, *nelem*, or *elsize* is 0, a unique pointer to the arena is returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `brk(2)`, `getrlimit(2)`, `mmap(2)`, `realloc(3C)`, `malloc(3MALLOC)`, `attributes(5)`

Name mblen – get number of bytes in a character

Synopsis

```
#include <stdlib.h>

int mblen(const char *s, size_t n);
```

Description If *s* is not a null pointer, `mblen()` determines the number of bytes constituting the character pointed to by *s*. It is equivalent to:

```
mbtowc((wchar_t *)0, s, n);
```

A call with *s* as a null pointer causes this function to return `0`. The behavior of this function is affected by the `LC_CTYPE` category of the current locale.

Return Values If *s* is a null pointer, `mblen()` returns `0`. It *s* is not a null pointer, `mblen()` returns `0` (if *s* points to the null byte), the number of bytes that constitute the character (if the next *n* or fewer bytes form a valid character), or `−1` (if they do not form a valid character) and may set `errno` to indicate the error. In no case will the value returned be greater than *n* or the value of the `MB_CUR_MAX` macro.

Errors The `mblen()` function may fail if:

`EILSEQ` Invalid character sequence is detected.

Usage The `mblen()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe with exceptions

See Also `mbstowcs(3C)`, `mbtowc(3C)`, `setlocale(3C)`, `wcstombs(3C)`, `wctomb(3C)`, `attributes(5)`, `standards(5)`

Name mbrlen – get number of bytes in a character (restartable)

Synopsis

```
#include <wchar.h>

size_t mbrlen(const char *restrict s, size_t n, mbstate_t *restrict ps);
```

Description If *s* is not a null pointer, `mbrlen()` determines the number of bytes constituting the character pointed to by *s*. It is equivalent to:

```
mbstate_t internal;
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

If *ps* is a null pointer, the `mbrlen()` function uses its own internal `mbstate_t` object, which is initialized at program startup to the initial conversion state. Otherwise, the `mbstate_t` object pointed to by *ps* is used to completely describe the current conversion state of the associated character sequence. Solaris will behave as if no function defined in the Solaris Reference Manual calls `mbrlen()`.

The behavior of this function is affected by the `LC_CTYPE` category of the current locale. See `environ(5)`.

Return Values The `mbrlen()` function returns the first of the following that applies:

`0` If the next *n* or fewer bytes complete the character that corresponds to the null wide-character.

`positive` If the next *n* or fewer bytes complete a valid character; the value returned is the number of bytes that complete the character.

`(size_t)−2` If the next *n* bytes contribute to an incomplete but potentially valid character, and all *n* bytes have been processed. When *n* has at least the value of the `MB_CUR_MAX` macro, this case can only occur if *s* points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

`(size_t)−1` If an encoding error occurs, in which case the next *n* or fewer bytes do not contribute to a complete and valid character. In this case, `EILSEQ` is stored in `errno` and the conversion state is undefined.

Errors The `mbrlen()` function may fail if:

`EINVAL` The *ps* argument points to an object that contains an invalid conversion state.

`EILSEQ` Invalid character sequence is detected.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	See NOTES below
Standard	See `standards(5)`.

See Also `mbrtowc(3C)`, `mbsinit(3C)`, `setlocale(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Notes If *ps* is not a null pointer, `mbrlen()` uses the `mbstate_t` object pointed to by *ps* and the function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale. If *ps* is a null pointer, `mbrlen()` uses its internal `mbstate_t` object and the function is Unsafe in multithreaded applications.

Name mbrtowc – convert a character to a wide-character code (restartable)

Synopsis

```
#include <wchar.h>

size_t mbrtowc(wchar_t *restrict pwc, const char *restrict s, size_t n,
     mbstate_t *restrict ps);
```

Description If *s* is a null pointer, the `mbrtowc()` function is equivalent to the call:

```
mbrtowc(NULL, "", 1, ps)
```

In this case, the values of the arguments *pwc* and *n* are ignored.

If *s* is not a null pointer, the `mbrtowc()` function inspects at most *n* bytes beginning at the byte pointed to by *s* to determine the number of bytes needed to complete the next character (including any shift sequences). If the function determines that the next character is completed, it determines the value of the corresponding wide-character and then, if *pwc* is not a null pointer, stores that value in the object pointed to by *pwc*. If the corresponding wide-character is the null wide-character, the resulting state described is the initial conversion state.

If *ps* is a null pointer, the `mbrtowc()` function uses its own internal `mbstate_t` object, which is initialized at program startup to the initial conversion state. Otherwise, the `mbstate_t` object pointed to by *ps* is used to completely describe the current conversion state of the associated character sequence. Solaris will behave as if no function defined in the Solaris Reference Manual calls `mbrtowc()`.

The behavior of this function is affected by the `LC_CTYPE` category of the current locale. See `environ(5)`.

Return Values The `mbrtowc()` function returns the first of the following that applies:

`0`	If the next *n* or fewer bytes complete the character that corresponds to the null wide-character (which is the value stored).
`positive`	If the next *n* or fewer bytes complete a valid character (which is the value stored); the value returned is the number of bytes that complete the character.
`(size_t)−2`	If the next *n* bytes contribute to an incomplete but potentially valid character, and all *n* bytes have been processed (no value is stored). When *n* has at least the value of the `MB_CUR_MAX` macro, this case can only occur if *s* points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
`(size_t)−1`	If an encoding error occurs, in which case the next *n* or fewer bytes do not contribute to a complete and valid character (no value is stored). In this case, `EILSEQ` is stored in `errno` and the conversion state is undefined.

Errors The `mbrtowc()` function may fail if:

`EINVAL` The *ps* argument points to an object that contains an invalid conversion state.

`EILSEQ` Invalid character sequence is detected.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See NOTES below
Standard	See `standards(5)`.

See Also `mbsinit(3C)`, `setlocale(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Notes If *ps* is not a null pointer, `mbrtowc()` uses the `mbstate_t` object pointed to by *ps* and the function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale. If *ps* is a null pointer, `mbrtowc()` uses its internal `mbstate_t` object and the function is Unsafe in multithreaded applications.

Name mbsinit – determine conversion object status

Synopsis

```
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

Description If `ps` is not a null pointer, the `mbsinit()` function determines whether the object pointed to by `ps` describes an initial conversion state.

Return Values The `mbsinit()` function returns non-zero if `ps` is a null pointer, or if the pointed-to object describes an initial conversion state; otherwise, it returns `0`.

If an `mbstate_t` object is altered by any of the functions described as "restartable", and is then used with a different character sequence, or in the other conversion direction, or with a different `LC_CTYPE` category setting than on earlier function calls, the behavior is undefined. See `environ(5)`.

Errors No errors are defined.

Usage The `mbstate_t` object is used to describe the current conversion state from a particular character sequence to a wide-character sequence (or vice versa) under the rules of a particular setting of the `LC_CTYPE` category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new character sequence in the initial shift state. A zero-valued `mbstate_t` object is at least one way to describe an initial conversion state. A zero-valued `mbstate_t` object can be used to initiate conversion involving any character sequence, in any `LC_CTYPE` category setting.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `mbrlen(3C)`, `mbrtowc(3C)`, `mbsrtowcs(3C)`, `setlocale(3C)`, `wcrtomb(3C)`, `wcsrtombs(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Notes The `mbsinit()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Name mbsrtowcs – convert a character string to a wide-character string (restartable)

Synopsis

```
#include <wchar.h>

size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,
     size_t len, mbstate_t *restrict ps);
```

Description The `mbsrtowcs()` function converts a sequence of characters, beginning in the conversion state described by the object pointed to by *ps*, from the array indirectly pointed to by *src* into a sequence of corresponding wide-characters. If *dst* is not a null pointer, the converted characters are stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops early in either of the following cases:

- When a sequence of bytes is encountered that does not form a valid character.
- When *len* codes have been stored into the array pointed to by *dst* (and *dst* is not a null pointer).

Each conversion takes place as if by a call to the `mbrtowc()` function.

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last character converted (if any). If conversion stopped due to reaching a terminating null character, and if *dst* is not a null pointer, the resulting state described is the initial conversion state.

If *ps* is a null pointer, the `mbsrtowcs()` function uses its own internal `mbstate_t` object, which is initialized at program startup to the initial conversion state. Otherwise, the `mbstate_t` object pointed to by *ps* is used to completely describe the current conversion state of the associated character sequence. Solaris will behave as if no function defined in the Solaris Reference Manual calls `mbsrtowcs()`.

The behavior of this function is affected by the `LC_CTYPE` category of the current locale. See `environ(5)`.

Return Values If the input conversion encounters a sequence of bytes that do not form a valid character, an encoding error occurs. In this case, the `mbsrtowcs()` function stores the value of the macro `EILSEQ` in `errno` and returns `(size_t)−1`; the conversion state is undefined. Otherwise, it returns the number of characters successfully converted, not including the terminating null (if any).

Errors The `mbsrtowcs()` function may fail if:

`EINVAL` The *ps* argument points to an object that contains an invalid conversion state.

`EILSEQ` Invalid character sequence is detected.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See NOTES below
Standard	See standards(5).

See Also mbrtowc(3C), mbsinit(3C), setlocale(3C), attributes(5), environ(5), standards(5)

Notes If *ps* is not a null pointer, `mbsrtowcs()` uses the `mbstate_t` object pointed to by *ps* and the function can be used safely in multithreaded applications, as long as setlocale(3C) is not being called to change the locale. If *ps* is a null pointer, `mbsrtowcs()` uses its internal `mbstate_t` object and the function is Unsafe in multithreaded applications.

Name mbstowcs – convert a character string to a wide-character string

Synopsis

```
#include <stdlib.h>

size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s, size_t n);
```

Description The `mbstowcs()` function converts a sequence of characters from the array pointed to by *s* into a sequence of corresponding wide-character codes and stores not more than *n* wide-character codes into the array pointed to by *pwcs*. No characters that follow a null byte (which is converted into a wide-character code with value `0`) will be examined or converted. Each character is converted as if by a call to `mbtowc(3C)`.

No more than *n* elements will be modified in the array pointed to by *pwcs*. If copying takes place between objects that overlap, the behavior is undefined.

The behavior of this function is affected by the `LC_CTYPE` category of the current locale. If *pwcs* is a null pointer, `mbstowcs()` returns the length required to convert the entire array regardless of the value of *n*, but no values are stored.

Return Values If an invalid character is encountered, `mbstowcs()` returns `(size_t)−1` and may set `errno` to indicate the error. Otherwise, `mbstowcs()` returns the number of the array elements modified (or required if *pwcs* is `NULL`), not including a terminating `0` code, if any. The array will not be zero-terminated if the value returned is *n*.

Errors The `mbstowcs()` function may fail if:

`EILSEQ` Invalid byte sequence is detected.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `mblen(3C)`, `mbtowc(3C)`, `setlocale(3C)`, `wcstombs(3C)`, `wctomb(3C)`, `attributes(5)`, `standards(5)`

Name mbtowc – convert a character to a wide-character code

Synopsis

```
#include <stdlib.h>

int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);
```

Description If *s* is not a null pointer, `mbtowc()` determines the number of the bytes that constitute the character pointed to by *s*. It then determines the wide-character code for the value of type `wchar_t` that corresponds to that character. (The value of the wide-character code corresponding to the null byte is 0.) If the character is valid and *pwc* is not a null pointer, `mbtowc()` stores the wide-character code in the object pointed to by *pwc*.

A call with *s* as a null pointer causes this function to return `0`. The behavior of this function is affected by the `LC_CTYPE` category of the current locale. At most *n* bytes of the array pointed to by *s* will be examined.

Return Values If *s* is a null pointer, `mbtowc()` returns `0`. If *s* is not a null pointer, `mbtowc()` returns `0` (if *s* points to the null byte), the number of bytes that constitute the converted character (if the next *n* or fewer bytes form a valid character), or `−1` and may set `errno` to indicate the error (if they do not form a valid character).

In no case will the value returned be greater than *n* or the value of the `MB_CUR_MAX` macro.

Errors The `mbtowc()` function may fail if:

`EILSEQ` Invalid character sequence is detected.

Usage The `mbtowc()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `mblen(3C)`, `mbstowcs(3C)`, `setlocale(3C)`, `wcstombs(3C)`, `wctomb(3C)`, `attributes(5)`, `standards(5)`

Name membar_ops, membar_enter, membar_exit, membar_producer, membar_consumer – memory access synchronization barrier operations

Synopsis

```
#include <atomic.h>

void membar_enter(void);

void membar_exit(void);

void membar_producer(void);

void membar_consumer(void);
```

Description The `membar_enter()` function is a generic memory barrier used during lock entry. It is placed after the memory operation that acquires the lock to guarantee that the lock protects its data. No stores from after the memory barrier will reach visibility and no loads from after the barrier will be resolved before the lock acquisition reaches global visibility.

The `membar_exit()` function is a generic memory barrier used during lock exit. It is placed before the memory operation that releases the lock to guarantee that the lock protects its data. All loads and stores issued before the barrier will be resolved before the subsequent lock update reaches visibility.

The `membar_enter()` and `membar_exit()` functions are used together to allow regions of code to be in relaxed store order and then ensure that the load or store order is maintained at a higher level. They are useful in the implementation of mutex exclusion locks.

The `membar_producer()` function arranges for all stores issued before this point in the code to reach global visibility before any stores that follow. This is useful in producer modules that update a data item, then set a flag that it is available. The memory barrier guarantees that the available flag is not visible earlier than the updated data, thereby imposing store ordering.

The `membar_consumer()` function arranges for all loads issued before this point in the code to be completed before any subsequent loads. This is useful in consumer modules that check if data is available and read the data. The memory barrier guarantees that the data is not sampled until after the available flag has been seen, thereby imposing load ordering.

Return Values No values are returned.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also atomic_add(3C), atomic_and(3C), atomic_bits(3C), atomic_cas(3C), atomic_dec(3C), atomic_inc(3C), atomic_ops(3C), atomic_or(3C), atomic_swap(3C), attributes(5), atomic_ops(9F)

Notes Atomic instructions (see atomic_ops(3C)) ensure global visibility of atomically-modified variables on completion. In a relaxed store order system, this does not guarantee that the visibility of other variables will be synchronized with the completion of the atomic instruction. If such synchronization is required, memory barrier instructions must be used.

Name memory, memccpy, memchr, memcmp, memcpy, memmove, memset, memmem – memory operations

Synopsis

```
#include <string.h>

void *memccpy(void *restrict s1, const void *restrict s2,
     int c, size_t n);

void *memchr(const void *s, int c, size_t n);

int memcmp(const void *s1, const void *s2, size_t n);

void *memcpy(void *restrict s1, const void *restrict s2, size_t n);

void *memmove(void *s1, const void *s2, size_t n);

void *memset(void *s, int c, size_t n);

void *memmem(const void *haystack, size_t haystacklen, const void *needle,
     size_t needlelen);
```

ISO C++

```
#include <string.h>

const void *memchr(const void *s, int c, size_t n);

#include <cstring>

void *std::memchr(void *s, int c, size_t n);
```

Description These functions operate as efficiently as possible on memory areas (arrays of bytes bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

The `memccpy()` function copies bytes from memory area *s2* into *s1*, stopping after the first occurrence of *c* (converted to an `unsigned char`) has been copied, or after *n* bytes have been copied, whichever comes first. It returns a pointer to the byte after the copy of *c* in *s1*, or a null pointer if *c* was not found in the first *n* bytes of *s2*.

The `memchr()` function returns a pointer to the first occurrence of *c* (converted to an `unsigned char`) in the first *n* bytes (each interpreted as an `unsigned char`) of memory area *s*, or a null pointer if *c* does not occur.

The `memcmp()` function compares its arguments, looking at the first *n* bytes (each interpreted as an `unsigned char`), 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* when taken to be unsigned characters.

The `memcpy()` function copies *n* bytes from memory area *s2* to *s1*. It returns *s1*. If copying takes place between objects that overlap, the behavior is undefined.

The `memmove()` function copies *n* bytes from memory area *s2* to memory area *s1*. Copying between objects that overlap will take place correctly. It returns *s1*.

The `memset()` function sets the first *n* bytes in memory area *s* to the value of *c* (converted to an `unsigned char`). It returns *s*.

The `memmem()` function locates the start of the first occurrence of the substring *needle* of length *needlelen* in the memory area *haystack* of length *haystacklen*. It returns a pointer to the start of the substring, or `NULL` if the substring is not found.

Usage Using `memcpy()` might be faster than using `memmove()` if the application knows that the objects being copied do not overlap.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `string(3C)`, `attributes(5)`, `standards(5)`

Notes Overlap between objects being copied can arise even when their (virtual) address ranges appear to be disjoint; for example, as a result of memory-mapping overlapping portions of the same underlying file, or of attaching the same shared memory segment more than once.

Name mkfifo, mkfifoat – make a FIFO special file

Synopsis

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

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

int mkfifoat(int fd, const char *path, mode_t mode);
```

Description The `mkfifo()` function creates a new FIFO special file named by the pathname pointed to by *path*. The file permission bits of the new FIFO are initialized from *mode*. The file permission bits of the *mode* argument are modified by the process's file creation mask (see `umask(2)`). Bits other than the file permission bits in *mode* are ignored.

If *path* names a symbolic link, `mkfifo()` fails and sets `errno` to `EEXIST`.

The FIFO's user ID is set to the process's effective user ID. The FIFO's group ID is set to the group ID of the parent directory or to the effective group ID of the process.

The `mkfifo()` function calls `mknod(2)` to create the file.

Upon successful completion, `mkfifo()` marks for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the file. Also, the `st_ctime` and `st_mtime` fields of the directory that contains the new entry are marked for update.

The `mkfifoat()` function is equivalent to `mkfifo()` except in the case where *path* specifies a relative path. In this case the newly created FIFO is created relative to the directory associated with the file descriptor *fd* instead of the current working directory. If the file descriptor was opened without `O_SEARCH`, the function checks whether directory searches are permitted using the current permissions of the directory underlying the file descriptor. If the file descriptor was opened with `O_SEARCH`, the function does not perform the check

If `mkfifoat()` is passed the special value `AT_FDCWD` in the *fd* parameter, the current working directory is used and the is be identical to a call to `mkfifo()`.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `mkfifo()` and `mkfifoat()` functions will fail if:

`EACCES`	A component of the path prefix denies search permission, or write permission is denied on the parent directory of the FIFO to be created.
`EEXIST`	The named file already exists.
`ELOOP`	A loop exists in symbolic links encountered during resolution of the *path* argument.
`ENAMETOOLONG`	The length of the *path* argument exceeds {`PATH_MAX`} or a pathname component is longer than {`NAME_MAX`}.

ENOENT — A component of the path prefix specified by *path* does not name an existing directory or *path* is an empty string.

ENOSPC — The directory that would contain the new file cannot be extended or the file system is out of file-allocation resources.

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

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

The `mkfifoat()` functions will fail if:

EACCES — *fd* was not opened with `O_SEARCH` and the permissions of the directory underlying *fd* do not permit directory searches.

EBADF — The *path* argument does not specify an absolute path and the *fd* argument is neither `AT_FDCWD` nor a valid file descriptor open for reading or searching.

The `mkfifo()` and `mkfifoat()` functions may fail if:

ELOOP — Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG — The length of the *path* argument exceeds {`PATH_MAX`} or a pathname component is longer than {`NAME_MAX`}.

The `mkfifoat()` functions may fail if:

ENOTDIR — The *path* argument is not an absolute path and *fd* is neither `AT_FDCWD` nor a file descriptor associated with a directory.

Examples

EXAMPLE 1 Create a FIFO File

The following example demonstrates how to create a FIFO file named `/home/cnd/mod_done` with read and write permissions for the owner and read permissions for the group and others.

```
#include sys/stat.h>
int status;
...
status = mkfifo("/home/cnd/mod_done", S_IWUSR | S_IRUSR |
        S_IRGRP | S_IROTH);
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also mkdir(1), chmod(2), exec(2), mknod(2), umask(2), stat.h(3HEAD), ufs(7FS), attributes(5), standards(5)

Name mkstemp, mkstemps, mkdtemp – make a unique file name from a template and open the file

Synopsis

```
#include <stdlib.h>

int mkstemp(char *template);

int mkstemps(char *template, int slen);

char *mkdtemp(char *template);
```

Description The `mkstemp()` function replaces the contents of the string pointed to by *template* by a unique file name, and returns a file descriptor for the file open for reading and writing. The function thus prevents any possible race condition between testing whether the file exists and opening it for use. The string in *template* should look like a file name with six trailing 'X's; `mkstemp()` replaces each 'X' with a character from the portable file name character set. The characters are chosen such that the resulting name does not duplicate the name of an existing file.

The `mkstemps()` function behaves the same as `mkstemp()`, except it permits a suffix to exist in the template. The template should be of the form `/tmp/tmp`*XXXXXXsuffix*. The *slen* parameter specifies the length of the suffix string.

The `mkdtemp()` function makes the same replacement to the template as in `mktemp(3C)` and creates the template directory using `mkdir(2)`, passing a *mode* argument of 0700.

Return Values Upon successful completion, `mkstemp()` returns an open file descriptor. Otherwise –1 is returned if no suitable file could be created.

Errors The `mkstemp()`, `mkstemps()`, and `mkdtemp()` functions can set `errno` to the same values as `lstat(2)`.

The `mkstemp()` and `mkstemps()` functions can set `errno` to the same values as `open(2)`.

The `mkdtemp()` function can set `errno` to the same values as `mkdir(2)`.

Usage It is possible to run out of letters.

The `mkstemp()` function does not check to determine whether the file name part of *template* exceeds the maximum allowable file name length.

The `tmpfile(3C)` function is preferred over this function.

The `mkstemp()` function is frequently used to create a temporary file that will be removed by the application before the application terminates.

The `mkstemp()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Standard	For `mkstemp()`, see `standards(5)`.

See Also `getpid(2)`, `lstat(2)`, `mkdir(2)`, `open(2)`, `tmpfile(3C)`, `mktemp(3C)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Name mktemp – make a unique file name from a template

Synopsis

```
#include <stdlib.h>

char *mktemp(char *template);
```

Description The `mktemp()` function replaces the contents of the string pointed to by *template* with a unique file name, and returns *template*. The string in *template* should look like a file name with six trailing 'X's; `mktemp()` will replace the 'X's with a character string that can be used to create a unique file name. Only 26 unique file names per thread can be created for each unique *template*.

Return Values The `mktemp()` function returns the pointer *template*. If a unique name cannot be created, *template* points to a null string.

Errors No errors are defined.

Examples **EXAMPLE 1** Generate a filename.

The following example replaces the contents of the "template" string with a 10-character filename beginning with the characters "file" and returns a pointer to the "template" string that contains the new filename.

```
#include <stdlib.h>
...
char *template = "/tmp/fileXXXXXX";
char *ptr;
ptr = mktemp(template);
```

Usage Between the time a pathname is created and the file opened, it is possible for some other process to create a file with the same name. The `mkstemp(3C)` function avoids this problem and is preferred over this function.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `mkstemp(3C)`, `tmpfile(3C)`, `tmpnam(3C)`, `attributes(5)`, `standards(5)`

Name mktime – converts a tm structure to a calendar time

Synopsis

```
#include <time.h>

time_t mktime(struct tm *timeptr);
```

Description The `mktime()` function converts the time represented by the `tm` structure pointed to by *timeptr* into a calendar time (the number of seconds since 00:00:00 UTC, January 1, 1970).

The `tm` structure contains the following members:

```
int  tm_sec;     /* seconds after the minute [0, 60]  */
int  tm_min;     /* minutes after the hour [0, 59] */
int  tm_hour;    /* hour since midnight [0, 23] */
int  tm_mday;    /* day of the month [1, 31] */
int  tm_mon;     /* months since January [0, 11] */
int  tm_year;    /* years since 1900 */
int  tm_wday;    /* days since Sunday [0, 6] */
int  tm_yday;    /* days since January 1 [0, 365] */
int  tm_isdst;   /* flag for daylight savings time */
```

In addition to computing the calendar time, `mktime()` normalizes the supplied `tm` structure. The original values of the `tm_wday` and `tm_yday` components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated in the definition of the structure. On successful completion, the values of the `tm_wday` and `tm_yday` components are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to be within the appropriate ranges. The final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined.

The `tm_year` member must be for year 1901 or later. Calendar times before 20:45:52 UTC, December 13, 1901 or after 03:14:07 UTC, January 19, 2038 cannot be represented. Portable applications should not try to create dates before 00:00:00 UTC, January 1, 1970 or after 00:00:00 UTC, January 1, 2038.

The original values of the components may be either greater than or less than the specified range. For example, a `tm_hour` of –1 means 1 hour before midnight, `tm_mday` of 0 means the day preceding the current month, and `tm_mon` of –2 means 2 months before January of `tm_year`.

If `tm_isdst` is positive, the original values are assumed to be in the alternate timezone. If it turns out that the alternate timezone is not valid for the computed calendar time, then the components are adjusted to the main timezone. Likewise, if `tm_isdst` is zero, the original values are assumed to be in the main timezone and are converted to the alternate timezone if the main timezone is not valid. If `tm_isdst` is negative, `mktime()` attempts to determine whether the alternate timezone is in effect for the specified time.

Local timezone information is used as if `mktime()` had called `tzset()`. See ctime(3C).

Return Values If the calendar time can be represented in an object of type `time_t`, `mktime()` returns the specified calendar time without changing `errno`. If the calendar time cannot be represented, the function returns the value (`time_t`)−1 and sets `errno` to indicate the error.

Errors The `mktime()` function will fail if:

`EOVERFLOW` The date represented by the input `tm` struct cannot be represented in a `time_t`. Note that the `errno` setting may change if future revisions to the standards specify a different value.

Usage The `mktime()` function is MT-Safe in multithreaded applications, as long as no user-defined function directly modifies one of the following variables: `timezone`, `altzone`, `daylight`, and `tzname`. See `ctime(3C)`.

Note that −1 can be a valid return value for the time that is one second before the Epoch. The user should clear `errno` before calling `mktime()`. If `mktime()` then returns −1, the user should check `errno` to determine whether or not an error actually occurred.

The `mktime()` function assumes Gregorian dates. Times before the adoption of the Gregorian calendar will not match historial records.

Examples **EXAMPLE 1** Sample code using `mktime()`.

What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>
static char *const wday[ ] = {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday", "-unknown-"
};
struct tm time_str;
/* . . .*/
time_str.tm_year    = 2001 - 1900;
time_str.tm_mon = 7 - 1;
time_str.tm_mday = 4;
time_str.tm_hour = 0;
time_str.tm_min = 0;
time_str.tm_sec = 1;
time_str.tm_isdst = −1;
if (mktime(&time_str)== −1)
        time_str.tm_wday=7;
printf("%s\n", wday[time_str.tm_wday]);
```

Bugs The `zoneinfo` timezone data files do not transition past Tue Jan 19 03:14:07 2038 UTC. Therefore for 64-bit applications using `zoneinfo` timezones, calculations beyond this date may not use the correct offset from standard time, and could return incorrect values. This affects the 64-bit version of `mktime()`.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See standards(5).

See Also ctime(3C), getenv(3C), TIMEZONE(4), attributes(5), standards(5)

Name mlock, munlock – lock or unlock pages in memory

Synopsis

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

int mlock(caddr_t addr, size_t len);

int munlock(caddr_t addr, size_t len);
```

Standard conforming

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

int mlock(const void * addr, size_t len);

int munlock(const void * addr, size_t len);
```

Description The `mlock()` function uses the mappings established for the address range [*addr, addr* + *len*) to identify pages to be locked in memory. If the page identified by a mapping changes, such as occurs when a copy of a writable `MAP_PRIVATE` page is made upon the first store, the lock will be transferred to the newly copied private page.

The `munlock()` function removes locks established with `mlock()`.

A given page may be locked multiple times by executing an `mlock()` through different mappings. That is, if two different processes lock the same page, then the page will remain locked until both processes remove their locks. However, within a given mapping, page locks do not nest – multiple `mlock()` operations on the same address in the same process will all be removed with a single `munlock()`. Of course, a page locked in one process and mapped in another (or visible through a different mapping in the locking process) is still locked in memory. This fact can be used to create applications that do nothing other than lock important data in memory, thereby avoiding page I/O faults on references from other processes in the system.

The contents of the locked pages will not be transferred to or from disk except when explicitly requested by one of the locking processes. This guarantee applies only to the mapped data, and not to any associated data structures (file descriptors and on-disk metadata, among others).

If the mapping through which an `mlock()` has been performed is removed, an `munlock()` is implicitly performed. An `munlock()` is also performed implicitly when a page is deleted through file removal or truncation.

Locks established with `mlock()` are not inherited by a child process after a `fork()` and are not nested.

Attempts to `mlock()` more memory than a system-specific limit will fail.

Return Values Upon successful completion, the `mlock()` and `munlock()` functions return `0`. Otherwise, no changes are made to any locks in the address space of the process, the functions return `−1` and set `errno` to indicate the error.

Errors The `mlock()` and `munlock()` functions will fail if:

`EINVAL` The *addr* argument is not a multiple of the page size as returned by `sysconf(3C)`.

`ENOMEM` Addresses in the range [*addr, addr* + *len*) are invalid for the address space of a process, or specify one or more pages which are not mapped.

`ENOSYS` The system does not support this memory locking interface.

`EPERM` The {`PRIV_PROC_LOCK_MEMORY`} privilege is not asserted in the effective set of the calling process.

The `mlock()` function will fail if:

`EAGAIN` Some or all of the memory identified by the range [*addr, addr* + *len*) could not be locked because of insufficient system resources or because of a limit or resource control on locked memory.

Usage Because of the impact on system resources, the use of `mlock()` and `munlock()` is restricted to users with the {`PRIV_PROC_LOCK_MEMORY`} privilege.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `fork(2)`, `memcntl(2)`, `mmap(2)`, `plock(3C)`, `mlockall(3C)`, `sysconf(3C)`, `attributes(5)`, `standards(5)`

Name mlockall, munlockall – lock or unlock address space

Synopsis

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

int mlockall(int flags);

int munlockall(void);
```

Description The `mlockall()` function locks in memory all pages mapped by an address space.

The value of *flags* determines whether the pages to be locked are those currently mapped by the address space, those that will be mapped in the future, or both:

```
MCL_CURRENT  Lock current mappings
MCL_FUTURE   Lock future mappings
```

If `MCL_FUTURE` is specified for `mlockall()`, mappings are locked as they are added to the address space (or replace existing mappings), provided sufficient memory is available. Locking in this manner is not persistent across the `exec` family of functions (see `exec(2)`).

Mappings locked using `mlockall()` with any option may be explicitly unlocked with a `munlock()` call (see `mlock(3C)`).

The `munlockall()` function removes address space locks and locks on mappings in the address space.

All conditions and constraints on the use of locked memory that apply to `mlock(3C)` also apply to `mlockall()`.

Locks established with `mlockall()` are not inherited by a child process after a `fork(2)` call, and are not nested.

Return Values Upon successful completion, the `mlockall()` and `munlockall()` functions return `0`. Otherwise, they return `−1` and set `errno` to indicate the error.

Errors The `mlockall()` and `munlockall()` functions will fail if:

`EAGAIN` Some or all of the memory in the address space could not be locked due to sufficient resources. This error condition applies to `mlockall()` only.

`EINVAL` The *flags* argument contains values other than `MCL_CURRENT` and `MCL_FUTURE`.

`EPERM` The {`PRIV_PROC_LOCK_MEMORY`} privilege is not asserted in the effective set of the calling process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Standard	See standards(5).

See Also exec(2), fork(2), memcntl(2), mmap(2), plock(3C), mlock(3C), sysconf(3C), attributes(5), standards(5)

Name monitor – prepare process execution profile

Synopsis

```
#include <mon.h>

void monitor(int (*lowpc(), int (*highpc)(), WORD *buffer, size_t bufsize,
     size_t nfunc);
```

Description The `monitor()` function is an interface to the `profil(2)` function and is called automatically with default parameters by any program created by the `cc` utility with the `-p` option specified. Except to establish further control over profiling activity, it is not necessary to explicitly call `monitor()`.

When used, `monitor()` is called at least at the beginning and the end of a program. The first call to `monitor()` initiates the recording of two different kinds of execution-profile information: execution-time distribution and function call count. Execution-time distribution data is generated by `profil()` and the function call counts are generated by code supplied to the object file (or files) by `cc -p`. Both types of information are collected as a program executes. The last call to `monitor()` writes this collected data to the output file `mon.out`.

The name of the file written by `monitor()` is controlled by the environment variable `PROFDIR`. If `PROFDIR` does not exist, the file `mon.out` is created in the current directory. If `PROFDIR` exists but has no value, `monitor()` does no profiling and creates no output file. If `PROFDIR` is `dirname`, and `monitor()` is called automatically by compilation with `cc -p`, the file created is `dirname/`*pid.progname* where *progname* is the name of the program.

The *lowpc* and *highpc* arguments are the beginning and ending addresses of the region to be profiled.

The *buffer* argument is the address of a user-supplied array of `WORD` (defined in the header `<mon.h>`). The *buffer* argument is used by `monitor()` to store the histogram generated by `profil()` and the call counts.

The *bufsize* argument identifies the number of array elements in *buffer*.

The *nfunc* argument is the number of call count cells that have been reserved in *buffer*. Additional call count cells will be allocated automatically as they are needed.

The *bufsize* argument should be computed using the following formula:

```
size_of_buffer =
        sizeof(struct hdr) +
        nfunc * sizeof(struct cnt) +
        ((highpc-lowpc)/BARSIZE) * sizeof(WORD) +
        sizeof(WORD) – 1 ;
bufsize = (size_of_buffer / sizeof(WORD));
```

where:

- *lowpc*, *highpc*, *nfunc* are the same as the arguments to `monitor()`;

- *BARSIZE* is the number of program bytes that correspond to each histogram bar, or cell, of the `profil()` buffer;
- the `hdr` and `cnt` structures and the type `WORD` are defined in the header `<mon.h>`.

The default call to `monitor()` is as follows:

```
monitor (&eprol, &etext, wbuf, wbufsz, 600);
```

where:

- `eprol` is the beginning of the user's program when linked with `cc -p` (see `end(3C)`);
- `etext` is the end of the user's program (see `end(3C)`);
- `wbuf` is an array of `WORD` with *wbufsz* elements;
- `wbufsz` is computed using the *bufsize* formula shown above with *BARSIZE* of 8;
- `600` is the number of call count cells that have been reserved in *buffer*.

These parameter settings establish the computation of an execution-time distribution histogram that uses `profil()` for the entire program, initially reserves room for 600 call count cells in *buffer*, and provides for enough histogram cells to generate significant distribution-measurement results. For more information on the effects of *bufsize* on execution-distribution measurements, see `profil(2)`.

Examples

EXAMPLE 1 Example to stop execution monitoring and write the results to a file.

To stop execution monitoring and write the results to a file, use the following:

```
monitor( (int (*)( ) )0, (int (*)( ) )0, (WORD *)0, 0, 0);
```

Use `prof` to examine the results.

Usage

Additional calls to `monitor()` after `main()` has been called and before `exit()` has been called will add to the function-call count capacity, but such calls will also replace and restart the `profil()` histogram computation.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also

`profil(2)`, `end(3C)`, `attributes(5)`, `prof(5)`

Name mq_close – close a message queue

Synopsis

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

Description The `mq_close()` function removes the association between the message queue descriptor, *mqdes*, and its message queue. The results of using this message queue descriptor after successful return from this `mq_close()`, and until the return of this message queue descriptor from a subsequent `mq_open(3C)`, are undefined.

If the process (or thread) has successfully attached a notification request to the message queue via this *mqdes*, this attachment is removed and the message queue is available for another process to attach for notification.

Return Values Upon successful completion, `mq_close()` returns `0`; otherwise, the function returns `−1` and sets `errno` to indicate the error condition.

Errors The `mq_close()` function will fail if:

`EBADF` The *mqdes* argument is an invalid message queue descriptor.

`ENOSYS` The `mq_open()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `mqueue.h(3HEAD)`, `mq_notify(3C)`, `mq_open(3C)`, `mq_unlink(3C)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name mq_getattr – get message queue attributes

Synopsis

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
```

Description The *mqdes* argument specifies a message queue descriptor. The `mq_getattr()` function is used to get status information and attributes of the message queue and the open message queue description associated with the message queue descriptor. The results are returned in the *mq_attr* structure referenced by the *mqstat* argument.

Upon return, the following members will have the values associated with the open message queue description as set when the message queue was opened and as modified by subsequent `mq_setattr(3C)` calls:

`mq_flags` message queue flags

The following attributes of the message queue are returned as set at message queue creation:

`mq_maxmsg` maximum number of messages

`mq_msgsize` maximum message size

`mq_curmsgs` number of messages currently on the queue.

Return Values Upon successful completion, the `mq_getattr()` function returns `0`. Otherwise, the function returns `−1` and sets `errno` to indicate the error.

Errors The `mq_getattr()` function will fail if:

`EBADF` The *mqdes* argument is not a valid message queue descriptor.

`ENOSYS` The `mq_getattr()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `msgctl(2)`, `msgget(2)`, `msgrcv(2)`, `msgsnd(2)`, `mqueue.h(3HEAD)`, `mq_open(3C)`, `mq_send(3C)`, `mq_setattr(3C)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name mq_notify – notify process (or thread) that a message is available on a queue

Synopsis

```
#include <mqueue.h>

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

Description The `mq_notify()` function provides an asynchronous mechanism for processes to receive notice that messages are available in a message queue, rather than synchronously blocking (waiting) in `mq_receive(3C)`.

If *notification* is not `NULL`, this function registers the calling process to be notified of message arrival at an empty message queue associated with the message queue descriptor, *mqdes*. The notification specified by *notification* will be sent to the process when the message queue transitions from empty to non-empty. See `signal.h(3HEAD)`. At any time, only one process may be registered for notification by a specific message queue. If the calling process or any other process has already registered for notification of message arrival at the specified message queue, subsequent attempts to register for that message queue will fail.

If *notification* is `NULL` and the process is currently registered for notification by the specified message queue, the existing registration is removed. The message queue is then available for future registration.

When the notification is sent to the registered process, its registration is removed. The message queue is then available for registration.

If a process has registered for notification of message arrival at a message queue and some processes is blocked in `mq_receive(3C)` waiting to receive a message when a message arrives at the queue, the arriving message will be received by the appropriate `mq_receive(3C)`, and no notification will be sent to the registered process. The resulting behavior is as if the message queue remains empty, and this notification will not be sent until the next arrival of a message at this queue.

Any notification registration is removed if the calling process either closes the message queue or exits.

Return Values Upon successful completion, `mq_notify()` returns `0`; otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `mq_notify()` function will fail if:

`EBADF` The *mqdes* argument is not a valid message queue descriptor.

`EBUSY` A process is already registered for notification by the message queue.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `mq_close(3C)`, `mq_open(3C)`, `mq_receive(3C)`, `mq_send(3C)`, `mqueue.h(3HEAD)`, `siginfo.h(3HEAD)`, `signal.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name mq_open – open a message queue

Synopsis

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag,
     /* unsigned long mode, mq_attr attr */  ...);
```

Description The `mq_open()` function establishes the connection between a process and a message queue with a message queue descriptor. It creates a open message queue description that refers to the message queue, and a message queue descriptor that refers to that open message queue description. The message queue descriptor is used by other functions to refer to that message queue.

The *name* argument points to a string naming a message queue. The *name* argument must conform to the construction rules for a path-name. If *name* is not the name of an existing message queue and its creation is not requested, `mq_open()` fails and returns an error. The first character of *name* must be a slash (`/`) character and the remaining characters of *name* cannot include any slash characters. For maximum portability, *name* should include no more than 14 characters, but this limit is not enforced.

The *oflag* argument requests the desired receive and/or send access to the message queue. The requested access permission to receive messages or send messages is granted if the calling process would be granted read or write access, respectively, to a file with the equivalent permissions.

The value of *oflag* is the bitwise inclusive OR of values from the following list. Applications must specify exactly one of the first three values (access modes) below in the value of *oflag*:

`O_RDONLY` Open the message queue for receiving messages. The process can use the returned message queue descriptor with `mq_receive(3C)`, but not `mq_send(3C)`. A message queue may be open multiple times in the same or different processes for receiving messages.

`O_WRONLY` Open the queue for sending messages. The process can use the returned message queue descriptor with `mq_send(3C)` but not `mq_receive(3C)`. A message queue may be open multiple times in the same or different processes for sending messages.

`O_RDWR` Open the queue for both receiving and sending messages. The process can use any of the functions allowed for `O_RDONLY` and `O_WRONLY`. A message queue may be open multiple times in the same or different processes for sending messages.

Any combination of the remaining flags may additionally be specified in the value of *oflag*:

`O_CREAT` This option is used to create a message queue, and it requires two additional arguments: *mode*, which is of type `mode_t`, and *attr*, which is pointer to a `mq_attr` structure. If the pathname, *name*, has already been used to create a

message queue that still exists, then this flag has no effect, except as noted under O_EXCL (see below). Otherwise, a message queue is created without any messages in it.

The user ID of the message queue is set to the effective user ID of process, and the group ID of the message queue is set to the effective group ID of the process. The file permission bits are set to the value of *mode*, and modified by clearing all bits set in the file mode creation mask of the process (see umask(2)).

If *attr* is non-NULL and the calling process has the appropriate privilege on *name*, the message queue *mq_maxmsg* and *mq_msgsize* attributes are set to the values of the corresponding members in the mq_attr structure referred to by *attr*. If *attr* is non-NULL, but the calling process does not have the appropriate privilege on *name*, the mq_open() function fails and returns an error without creating the message queue.

O_EXCL — If both O_EXCL and O_CREAT are set, mq_open() will fail if the message queue *name* exists. The check for the existence of the message queue and the creation of the message queue if it does not exist are atomic with respect to other processes executing mq_open() naming the same *name* with both O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are not set, the result is undefined.

O_NONBLOCK — The setting of this flag is associated with the open message queue description and determines whether a mq_send(3C) or mq_receive(3C) waits for resources or messages that are not currently available, or fails with errno set to EAGAIN. See mq_send(3C) and mq_receive(3C) for details.

Return Values Upon successful completion, mq_open() returns a message queue descriptor; otherwise the function returns (mqd_t)−1 and sets errno to indicate the error condition.

Errors The mq_open() function will fail if:

EACCES — The message queue exists and the permissions specified by *oflag* are denied, or the message queue does not exist and permission to create the message queue is denied.

EEXIST — O_CREAT and O_EXCL are set and the named message queue already exists.

EINTR — The mq_open() operation was interrupted by a signal.

EINVAL — The mq_open() operation is not supported for the given name, or O_CREAT was specified in *oflag*, the value of *attr* is not NULL, and either mq_maxmsg or mq_msgsize was less than or equal to zero.

EMFILE	The number of open message queue descriptors in this process exceeds MQ_OPEN_MAX, of the number of open file descriptors in this process exceeds OPEN_MAX.
ENAMETOOLONG	The length of the *name* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.
ENFILE	Too many message queues are currently open in the system.
ENOENT	O_CREAT is not set and the named message queue does not exist.
ENOSPC	There is insufficient space for the creation of the new message queue.
ENOSYS	The mq_open() function is not supported by the system.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also exec(2), exit(2), umask(2), sysconf(3C), mqueue.h(3HEAD), mq_close(3C), mq_receive(3C), mq_send(3C), mq_setattr(3C), mq_unlink(3C), attributes(5), standards(5)

Notes Due to the manner in which message queues are implemented, they should not be considered secure and should not be used in security-sensitive applications.

Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set errno to ENOSYS.

Name mq_receive, mq_timedreceive, mq_reltimedreceive_np – receive a message from a message queue

Synopsis

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
     unsigned *msg_prio);

#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
     size_t msg_len, unsigned *restrict msg_prio,
     const struct timespec *restrict abs_timeout);

ssize_t mq_reltimedreceive_np(mqd_t mqdes,
     char *restrict msg_ptr, size_t msg_len,
     unsigned *restrict msg_prio,
     const struct timespec *restrict rel_timeout);
```

Description The `mq_receive()` function receives the oldest of the highest priority message(s) from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by *msg_len*, is less than the `mq_msgsize` member of the message queue, the function fails and returns an error. Otherwise, the selected message is removed from the queue and copied to the buffer pointed to by *msg_ptr*.

If the value of *msg_len* is greater than {`SSIZE_MAX`}, the result is implementation-defined.

If *msg_prio* is not `NULL`, the priority of the selected message is stored in the location referenced by *msg_prio*.

If the specified message queue is empty and `O_NONBLOCK` is not set in the message queue description associated with *mqdes*, (see `mq_open(3C)` and `mq_setattr(3C)`), `mq_receive()` blocks, waiting until a message is enqueued on the message queue, or until `mq_receive()` is interrupted by a signal. If more than one process (or thread) is waiting to receive a message when a message arrives at an empty queue, then the process of highest priority that has been waiting the longest is selected to receive the message. If the specified message queue is empty and `O_NONBLOCK` is set in the message queue description associated with *mqdes*, no message is removed from the queue, and `mq_receive()` returns an error.

The `mq_timedreceive()` function receives the oldest of the highest priority messages from the message queue specified by *mqdes* as described for the `mq_receive()` function. However, if `O_NONBLOCK` was not specified when the message queue was opened with the `mq_open(3C)` function, and no message exists on the queue to satisfy the receive, the wait for such a message is terminated when the specified timeout expires. If `O_NONBLOCK` is set, this function is equivalent to `mq_receive()`.

The `mq_reltimedreceive_np()` function is identical to the `mq_timedreceive()` function, except that the timeout is specified as a relative time interval.

For `mq_timedreceive()`, the timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the `CLOCK_REALTIME` clock (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

For `mq_reltimedreceive_np()`, the timeout expires when the time interval specified by *rel_timeout* passes, as measured by the `CLOCK_REALTIME` clock, or if the time interval specified by *rel_timeout* is negative at the time of the call.

The resolution of the timeout is the resolution of the `CLOCK_REALTIME` clock. The `timespec` argument is defined in the <`time.h`> header.

Under no circumstance does the operation fail with a timeout if a message can be removed from the message queue immediately. The validity of the timeout parameter need not be checked if a message can be removed from the message queue immediately.

Return Values Upon successful completion, `mq_receive()`, `mq_timedreceive()`, and `mq_reltimedreceive_np()` return the length of the selected message in bytes and the message is removed from the queue. Otherwise, no message is removed from the queue, the functions return a value of −1, and sets `errno` to indicate the error condition.

Errors The `mq_receive()`, `mq_timedreceive()`, and `mq_reltimedreceive_np()` functions will fail if:

`EAGAIN` `O_NONBLOCK` was set in the message description associated with *mqdes*, and the specified message queue is empty.

`EBADF` The *mqdes* argument is not a valid message queue descriptor open for reading.

`EINTR` The function was interrupted by a signal.

`EINVAL` The process or thread would have blocked, and the timeout parameter specified a nanoseconds field value less than zero or greater than or equal to 1,000 million.

`EMSGSIZE` The specified message buffer size, *msg_len*, is less than the message size member of the message queue.

`ETIMEDOUT` The `O_NONBLOCK` flag was not set when the message queue was opened, but no message arrived on the queue before the specified timeout expired.

The `mq_receive()`, `mq_timedreceive()`, and `mq_reltimedreceive_np()` functions may fail if:

`EBADMSG` A data corruption problem with the message has been detected.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See below.

For mq_receive() and mq_timedreceive(). see standards(5).

See Also mqueue.h(3HEAD), mq_open(3C), mq_send(3C), mq_setattr(3C), attributes(5), standards(5)

Name mq_send, mq_timedsend, mq_reltimedsend_np – send a message to a message queue

Synopsis

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
     unsigned msg_prio);

#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr,
     size_t msg_len, unsigned msg_prio,
     const struct timespec *restrict abs_timeout);

int mq_reltimedsend_np(mqd_t mqdes, const char *msg_ptr,
     size_t msg_len, unsigned msg_prio,
     const struct timespec *restrict rel_timeout);
```

Description The `mq_send()` function adds the message pointed to by the argument *msg_ptr* to the message queue specified by *mqdes*. The *msg_len* argument specifies the length of the message in bytes pointed to by *msg_ptr*. The value of *msg_len* is less than or equal to the *mq_msgsize* attribute of the message queue, or `mq_send()` fails.

If the specified message queue is not full, `mq_send()` behaves as if the message is inserted into the message queue at the position indicated by the *msg_prio* argument. A message with a larger numeric value of *msg_prio* is inserted before messages with lower values of *msg_prio*. A message will be inserted after other messages in the queue, if any, with equal *msg_prio*. The value of *msg_prio* must be greater than zero and less than or equal to `MQ_PRIO_MAX`.

If the specified message queue is full and `O_NONBLOCK` is not set in the message queue description associated with *mqdes* (see `mq_open(3C)` and `mq_setattr(3C)`), `mq_send()` blocks until space becomes available to enqueue the message, or until `mq_send()` is interrupted by a signal. If more than one thread is waiting to send when space becomes available in the message queue, then the thread of the highest priority which has been waiting the longest is unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked. If the specified message queue is full and `O_NONBLOCK` is set in the message queue description associated with *mqdes*, the message is not queued and `mq_send()` returns an error.

The `mq_timedsend()` function adds a message to the message queue specified by *mqdes* in the manner defined for the `mq_send()` function. However, if the specified message queue is full and `O_NONBLOCK` is not set in the message queue description associated with *mqdes*, the wait for sufficient room in the queue is terminated when the specified timeout expires. If `O_NONBLOCK` is set in the message queue description, this function is equivalent to `mq_send()`.

The `mq_reltimedsend_np()` function is identical to the `mq_timedsend()` function, except that the timeout is specified as a relative time interval.

For `mq_timedsend()`, the timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the `CLOCK_REALTIME` clock (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

For `mq_reltimedsend_np()`, the timeout expires when the time interval specified by *rel_timeout* passes, as measured by the `CLOCK_REALTIME` clock, or if the time interval specified by *rel_timeout* is negative at the time of the call.

The resolution of the timeout is the resolution of the `CLOCK_REALTIME` clock. The `timespec` argument is defined in the `<time.h>` header.

Under no circumstance does the operation fail with a timeout if there is sufficient room in the queue to add the message immediately. The validity of the timeout parameter need not be checked when there is sufficient room in the queue.

Return Values Upon successful completion, `mq_send()`, `mq_timedsend()`, and `mq_reltimedsend_np()` return `0`. Otherwise, no message is enqueued, the functions return `−1`, and `errno` is set to indicate the error.

Errors The `mq_send()`, `mq_timedsend()`, and `mq_reltimedsend_np()` functions will fail if:

`EAGAIN` The `O_NONBLOCK` flag is set in the message queue description associated with *mqdes*, and the specified message queue is full.

`EBADF` The *mqdes* argument is not a valid message queue descriptor open for writing.

`EINTR` A signal interrupted the function call.

`EINVAL` The value of *msg_prio* was outside the valid range.

`EINVAL` The process or thread would have blocked, and the timeout parameter specified a nanoseconds field value less than zero or greater than or equal to 1,000 million.

`EMSGSIZE` The specified message length, *msg_len*, exceeds the message size attribute of the message queue.

`ETIMEDOUT` The `O_NONBLOCK` flag was not set when the message queue was opened, but the timeout expired before the message could be added to the queue.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See below.

For `mq_send()` and `mq_timedsend()`, see `standards(5)`.

See Also `sysconf(3C)`, `mqueue.h(3HEAD)`, `mq_open(3C)`, `mq_receive(3C)`, `mq_setattr(3C)`, `attributes(5)`, `standards(5)`

Name mq_setattr – set/get message queue attributes

Synopsis

```
#include <mqueue.h>

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

Description The `mq_setattr()` function is used to set attributes associated with the open message queue description referenced by the message queue descriptor specified by *mqdes*.

The message queue attributes corresponding to the following members defined in the `mq_attr` structure are set to the specified values upon successful completion of `mq_setattr()`:

`mq_flags` The value of this member is either `0` or `O_NONBLOCK`.

The values of `mq_maxmsg`, `mq_msgsize`, and `mq_curmsgs` are ignored by `mq_setattr()`.

If *omqstat* is non-`NULL`, `mq_setattr()` stores, in the location referenced by *omqstat*, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to `mq_getattr()` at that point.

Return Values Upon successful completion, `mq_setattr()` returns `0` and the attributes of the message queue will have been changed as specified. Otherwise, the message queue attributes are unchanged, and the function returns `−1` and sets `errno` to indicate the error.

Errors The `mq_setattr()` function will fail if:

`EBADF` The *mqdes* argument is not a valid message queue descriptor.

`ENOSYS` The `mq_setattr()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `msgctl(2)`, `msgget(2)`, `msgrcv(2)`, `msgsnd(2)`, `mq_getattr(3C)`, `mq_open(3C)`, `mq_receive(3C)`, `mq_send(3C)`, `mqueue.h(3HEAD)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name mq_unlink – remove a message queue

Synopsis

```
#include <mqueue.h>

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

Description The `mq_unlink()` function removes the message queue named by the pathname *name*. After a successful call to `mq_unlink()` with *name*, a call to `mq_open(3C)` with *name* fails if the flag `O_CREAT` is not set in *flags*. If one or more processes have the message queue open when `mq_unlink()` is called, destruction of the message queue is postponed until all references to the message queue have been closed. Calls to `mq_open(3C)` to re-create the message queue may fail until the message queue is actually removed. However, the `mq_unlink()` call need not block until all references have been closed; it may return immediately.

Return Values Upon successful completion, `mq_unlink()` returns `0`; otherwise, the named message queue is not changed by this function call, the function returns `−1` and sets `errno` to indicate the error.

Errors The `mq_unlink()` function will fail if:

`EACCES` Permission is denied to unlink the named message queue.

`ENAMETOOLONG` The length of the *name* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

`ENOENT` The named message queue, *name*, does not exist.

`ENOSYS` `mq_unlink()` is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `mqueue.h(3HEAD)`, `mq_close(3C)`, `mq_open(3C)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name msync – synchronize memory with physical storage

Synopsis

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

int msync(void *addr, size_t len, int flags);
```

Description The `msync()` function writes all modified copies of pages over the range [*addr, addr* + *len*) to the underlying hardware, or invalidates any copies so that further references to the pages will be obtained by the system from their permanent storage locations. The permanent storage for a modified `MAP_SHARED` mapping is the file the page is mapped to; the permanent storage for a modified `MAP_PRIVATE` mapping is its swap area.

The *flags* argument is a bit pattern built from the following values:

MS_ASYNC	perform asynchronous writes
MS_SYNC	perform synchronous writes
MS_INVALIDATE	invalidate mappings

If *flags* is `MS_ASYNC` or `MS_SYNC`, the function synchronizes the file contents to match the current contents of the memory region.

- All write references to the memory region made prior to the call are visible by subsequent read operations on the file.
- All writes to the same portion of the file prior to the call may or may not be visible by read references to the memory region.
- Unmodified pages in the specified range are not written to the underlying hardware.

If *flags* is `MS_ASYNC`, the function may return immediately once all write operations are scheduled; if *flags* is `MS_SYNC`, the function does not return until all write operations are completed.

If *flags* is `MS_INVALIDATE`, the function synchronizes the contents of the memory region to match the current file contents.

- All writes to the mapped portion of the file made prior to the call are visible by subsequent read references to the mapped memory region.
- All write references prior to the call, by any process, to memory regions mapped to the same portion of the file using `MAP_SHARED`, are visible by read references to the region.

If `msync()` causes any write to the file, then the file's `st_ctime` and `st_mtime` fields are marked for update.

Return Values Upon successful completion, `msync()` returns `0`; otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `msync()` function will fail if:

`EBUSY` Some or all of the addresses in the range [*addr*, *addr* + *len*) are locked and `MS_SYNC` with the `MS_INVALIDATE` option is specified.

`EAGAIN` Some or all pages in the range [*addr*, *addr* + *len*) are locked for I/O.

`EINVAL` The *addr* argument is not a multiple of the page size as returned by `sysconf(3C)`.

The *flags* argument is not some combination of `MS_ASYNC` and `MS_INVALIDATE`.

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

`ENOMEM` Addresses in the range [*addr, addr* + *len*) are outside the valid range for the address space of a process, or specify one or more pages that are not mapped.

`EPERM` `MS_INVALIDATE` was specified and one or more of the pages is locked in memory.

Usage The `msync()` function should be used by programs that require a memory object to be in a known state, for example in building transaction facilities.

Normal system activity can cause pages to be written to disk. Therefore, there are no guarantees that `msync()` is the only control over when pages are or are not written to disk.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `memcntl(2)`, `mmap(2)`, `sysconf(3C)`, `attributes(5)`, `standards(5)`

Name mtmalloc, mallocctl – MT hot memory allocator

Synopsis

```
#include <mtmalloc.h>
cc –o a.out –lthread –lmtmalloc

void *malloc(size_t size);

void free(void *ptr);

void *memalign(size_t alignment, size_t size);

void *realloc(void *ptr, size_t size);

void *valloc(size_t size);

void mallocctl(int cmd, long value);
```

Description The `malloc()` and `free()` functions provide a simple general-purpose memory allocation package that is suitable for use in high performance multithreaded applications. The suggested use of this library is in multithreaded applications; it can be used for single threaded applications, but there is no advantage in doing so. This library cannot be dynamically loaded with `dlopen(3C)` during runtime because there must be only one manager of the process heap.

The `malloc()` function returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to `free()` is a pointer to a block previously allocated by `malloc()` or `realloc()`. After `free()` is performed this space is available for further allocation. If *ptr* is a null pointer, no action occurs. The `free()` function does not set `errno`.

Undefined results will occur if the space assigned by `malloc()` is overrun or if a random number is handed to `free()`. A freed pointer that is passed to `free()` will send a `SIGABRT` signal to the calling process. This behavior is controlled by `mallocctl()`.

The `memalign()` function allocates *size* bytes on a specified alignment boundary and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of *alignment*. Note that the value of *alignment* must be a power of two, and must be greater than or equal to the size of a word.

The `realloc()` function changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If the new size of the block requires movement of the block, the space for the previous instantiation of the block is freed. If the new size is larger, the contents of the newly allocated portion of the block are unspecified. If *ptr* is `NULL`, `realloc()` behaves like `malloc()` for the specified size. If *size* is 0 and *ptr* is not a null pointer, the space pointed to is freed.

The `valloc()` function has the same effect as `malloc()`, except that the allocated memory will be aligned to a multiple of the value returned by `sysconf(_SC_PAGESIZE)`.

After possible pointer coercion, each allocation routine returns a pointer to a space that is suitably aligned for storage of any type of object.

The `malloc()`, `realloc()`, `memalign()`, and `valloc()` functions will fail if there is not enough available memory.

The `mallocctl()` function controls the behavior of the `malloc` library. The options fall into two general classes, debugging options and performance options.

`MTDOUBLEFREE` Allows double `free` of a pointer. Setting *value* to `1` means yes and `0` means no. The default behavior of double `free` results in a core dump.

`MTDEBUGPATTERN` Writes misaligned data into the buffer after `free()`. When the buffer is reallocated, the contents are verified to ensure that there was no access to the buffer after the `free`. If the buffer has been dirtied, a `SIGABRT` signal is delivered to the process. Setting *value* to `1` means yes and `0` means no. The default behavior is to *not* write misaligned data. The pattern used is `0xdeadbeef`. Use of this option results in a performance penalty.

`MTINITBUFFER` Writes misaligned data into the newly allocated buffer. This option is useful for detecting some accesses before initialization. Setting *value* to `1` means yes and `0` means no. The default behavior is to *not* write misaligned data to the newly allocated buffer. The pattern used is `0xbaddcafe`. Use of this option results in a performance penalty.

`MTCHUNKSIZE` This option changes the size of allocated memory when a pool has exhausted all available memory in the buffer. Increasing this value allocates more memory for the application. A substantial performance gain can occur because the library makes fewer calls to the OS for more memory. Acceptable number *value*s are between `9` and `256`. The default value is `9`. This value is multiplied by 8192.

Return Values If there is no available memory, `malloc()`, `realloc()`, `memalign()`, and `valloc()` return a null pointer. When `realloc()` is called with *size* > 0 and returns `NULL`, the block pointed to by *ptr* is left intact. If *size*, *nelem*, or *elsize* is 0, either a null pointer or a unique pointer that can be passed to `free()` is returned.

If `malloc()` or `realloc()` returns unsuccessfully, `errno` will be set to indicate the error.

Errors The `malloc()` and `realloc()` functions will fail if:

`ENOMEM` The physical limits of the system are exceeded by *size* bytes of memory which cannot be allocated.

`EAGAIN` There is not enough memory available to allocate *size* bytes of memory; but the application could try again later.

Usage Comparative features of the various allocation libraries can be found in the `umem_alloc(3MALLOC)` manual page.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `brk(2)`, `getrlimit(2)`, `bsdmalloc(3MALLOC)`, `dlopen(3C)`, `malloc(3C)`, `malloc(3MALLOC)`, `mapmalloc(3MALLOC)`, `signal.h(3HEAD)`, `umem_alloc(3MALLOC)`, `watchmalloc(3MALLOC)`, `attributes(5)`

Warnings Undefined results will occur if the size requested for a block of memory exceeds the maximum size of a process's heap. This information may be obtained using `getrlimit()`.

Name mutex_init, mutex_lock, mutex_trylock, mutex_unlock, mutex_consistent, mutex_destroy – mutual exclusion locks

Synopsis

```
cc –mt [ flag... ] file... [ library... ]
#include <thread.h>
#include <synch.h>

int mutex_init(mutex_t *mp, int type, void * arg);

int mutex_lock(mutex_t *mp);

int mutex_trylock(mutex_t *mp);

int mutex_unlock(mutex_t *mp);

int mutex_consistent(mutex_t *mp);

int mutex_destroy(mutex_t *mp);
```

Description Mutual exclusion locks (mutexes) prevent multiple threads from simultaneously executing critical sections of code that access shared data (that is, mutexes are used to serialize the execution of threads). All mutexes must be global. A successful call for a mutex lock by way of `mutex_lock()` will cause another thread that is also trying to lock the same mutex to block until the owner thread unlocks it by way of `mutex_unlock()`. Threads within the same process or within other processes can share mutexes.

Mutexes can synchronize threads within the same process or in other processes. Mutexes can be used to synchronize threads between processes if the mutexes are allocated in writable memory and shared among the cooperating processes (see `mmap(2)`), and have been initialized for this task.

Initialize Mutexes are either intra-process or inter-process, depending upon the argument passed implicitly or explicitly to the initialization of that mutex. A statically allocated mutex does not need to be explicitly initialized; by default, a statically allocated mutex is initialized with all zeros and its scope is set to be within the calling process.

For inter-process synchronization, a mutex needs to be allocated in memory shared between these processes. Since the memory for such a mutex must be allocated dynamically, the mutex needs to be explicitly initialized using `mutex_init()`.

The `mutex_init()` function initializes the mutex referenced by *mp* with the type specified by *type*. Upon successful initialization the state of the mutex becomes initialized and unlocked. Only the attribute type `LOCK_PRIO_PROTECT` uses *arg*. The *type* argument must be one of the following:

`USYNC_THREAD`
: The mutex can synchronize threads only in this process.

`USYNC_PROCESS`
: The mutex can synchronize threads in this process and other processes. The object initialized with this attribute must be allocated in memory shared between processes, either

in System V shared memory (see shmop(2)) or in memory mapped to a file (see mmap(2)). If the object is not allocated in such shared memory, it will not be shared between processes.

The *type* argument can be augmented by the bitwise-inclusive-OR of zero or more of the following flags:

LOCK_ROBUST
: The mutex can synchronize threads robustly. At the time of thread or process death, either by calling thr_exit() or exit() or due to process abnormal termination, the lock is unlocked if is held by the thread or process. The next owner of the mutex will acquire it with an error return of EOWNERDEAD. The application must always check the return value from mutex_lock() for a mutex of this type. The new owner of this mutex should then attempt to make the state protected by the mutex consistent, since this state could have been left inconsistent when the last owner died. If the new owner is able to make the state consistent, it should call mutex_consistent() to restore the state of the mutex and then unlock the mutex. All subsequent calls to mutex_lock()will then behave normally. Only the new owner can make the mutex consistent. If for any reason the new owner is not able to make the state consistent, it should not call mutex_consistent() but should simply unlock the mutex. All waiting processes will be awakened and all subsequent calls to mutex_lock() will fail in acquiring the mutex with an error value of ENOTRECOVERABLE. If the thread or process that acquired the lock with EOWNERDEAD terminates without unlocking the mutex, the next owner will acquire the lock with an error value of EOWNERDEAD.

  The memory for the object to be initialized with this attribute must be zeroed before initialization. Any thread or process interested in the robust lock can call mutex_init() to potentially initialize it, provided that all such callers of mutex_init() specify the same set of attribute flags. In this situation, if mutex_init() is called on a previously initialized robust mutex, mutex_init() will not reinitialize the mutex and will return the error value EBUSY.

LOCK_RECURSIVE
: A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The mutex must be unlocked as many times as it is locked.

LOCK_ERRORCHECK
: Unless LOCK_RECURSIVE is also set, a thread attempting to relock this mutex without first unlocking it will return with an error rather than deadlocking itself. A thread attempting to unlock this mutex without first owning it will return with an error.

LOCK_PRIO_INHERIT
: When a thread is blocking higher priority threads because of owning one or more mutexes with the LOCK_PRIO_INHERIT attribute, it executes at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this attribute.

LOCK_PRIO_PROTECT
: When a thread owns one or more mutexes initialized with the LOCK_PRIO_PROTECT attribute, it executes at the higher of its priority or the highest of the priority ceilings of all

the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes. When this attribute is specified, *arg* must point to an `int` containing the priority ceiling.

See `pthread_mutexattr_getrobust(3C)` for more information about robust mutexes. The `LOCK_ROBUST` attribute is the same as the POSIX `PTHREAD_MUTEX_ROBUST` attribute.

See `pthread_mutexattr_settype(3C)` for more information on recursive and error checking mutex types. The combination (`LOCK_RECURSIVE` | `LOCK_ERRORCHECK`) is the same as the POSIX `PTHREAD_MUTEX_RECURSIVE` type. By itself, `LOCK_ERRORCHECK` is the same as the `POSIX PTHREAD_MUTEX_ERRORCHECK` type.

The `LOCK_PRIO_INHERIT` attribute is the same as the POSIX `PTHREAD_PRIO_INHERIT` attribute. The `LOCK_PRIO_PROTECT` attribute is the same as the POSIX `PTHREAD_PRIO_PROTECT` attribute. See `pthread_mutexattr_getprotocol(3C)`, `pthread_mutexattr_getprioceiling(3C)`, and `pthread_mutex_getprioceiling(3C)` for a full discussion. The `LOCK_PRIO_INHERIT` and `LOCK_PRIO_PROTECT` attributes are mutually exclusive. Specifying both of these attributes causes `mutex_init()` to fail with `EINVAL`.

Initializing mutexes can also be accomplished by allocating in zeroed memory (default), in which case a *type* of `USYNC_THREAD` is assumed. In general, the following rules apply to mutex initialization:

- The same mutex must not be simultaneously initialized by multiple threads.
- A mutex lock must not be reinitialized while in use by other threads.

These rules do not apply to `LOCK_ROBUST` mutexes. See the description for `LOCK_ROBUST`above. If default mutex attributes are used, the macro `DEFAULTMUTEX` can be used to initialize mutexes that are statically allocated.

Default mutex initialization (intra-process):

```
mutex_t mp;
mutex_init(&mp, USYNC_THREAD, NULL);
```

or

```
mutex_t  mp  =  DEFAULTMUTEX;
```

Customized mutex initialization (inter-process):

```
mutex_init(&mp, USYNC_PROCESS, NULL);
```

Customized mutex initialization (inter-process robust):

```
mutex_init(&mp, USYNC_PROCESS | LOCK_ROBUST, NULL);
```

Statically allocated mutexes can also be initialized with macros specifying `LOCK_RECURSIVE` and/or `LOCK_ERRORCHECK`:

```
mutex_t mp = RECURSIVEMUTEX;
```

Same as (`USYNC_THREAD` | `LOCK_RECURSIVE`)

`mutex_t mp = ERRORCHECKMUTEX;`
Same as (`USYNC_THREAD` | `LOCK_ERRORCHECK`)

`mutex_t mp = RECURSIVE_ERRORCHECKMUTEX;`
Same as (`USYNC_THREAD` | `LOCK_RECURSIVE` | `LOCK_ERRORCHECK`)

Lock and Unlock

A critical section of code is enclosed by a the call to lock the mutex and the call to unlock the mutex to protect it from simultaneous access by multiple threads. Only one thread at a time may possess mutually exclusive access to the critical section of code that is enclosed by the mutex-locking call and the mutex-unlocking call, whether the mutex's scope is intra-process or inter-process. A thread calling to lock the mutex either gets exclusive access to the code starting from the successful locking until its call to unlock the mutex, or it waits until the mutex is unlocked by the thread that locked it.

Mutexes have ownership, unlike semaphores. Although any thread, within the scope of a mutex, can get an unlocked mutex and lock access to the same critical section of code, only the thread that locked a mutex should unlock it.

If a thread waiting for a mutex receives a signal, upon return from the signal handler, the thread resumes waiting for the mutex as if there was no interrupt. A mutex protects code, not data; therefore, strongly bind a mutex with the data by putting both within the same structure, or at least within the same procedure.

A call to `mutex_lock()` locks the mutex object referenced by *mp*. If the mutex is already locked, the calling thread blocks until the mutex is freed; this will return with the mutex object referenced by *mp* in the locked state with the calling thread as its owner. If the current owner of a mutex tries to relock the mutex, it will result in deadlock.

The `mutex_trylock()` function is the same as `mutex_lock()`, respectively, except that if the mutex object referenced by *mp* is locked (by any thread, including the current thread), the call returns immediately with an error.

The `mutex_unlock()` function are called by the owner of the mutex object referenced by *mp* to release it. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner). If there are threads blocked on the mutex object referenced by *mp* when `mutex_unlock()` is called, the *mp* is freed, and the scheduling policy will determine which thread gets the mutex. If the calling thread is not the owner of the lock, no error status is returned, and the behavior of the program is undefined.

Destroy

The `mutex_destroy()` function destroys the mutex object referenced by *mp*. The mutex object becomes uninitialized. The space used by the destroyed mutex variable is not freed. It needs to be explicitly reclaimed.

Return Values

If successful, these functions return `0`. Otherwise, an error number is returned.

Errors The `mutex_init()` function will fail if:

`EINVAL` The value specified by *type* is invalid, or the `LOCK_PRIO_INHERIT` and `LOCK_PRIO_PROTECT` attributes are both specified.

The `mutex_init()` function will fail for `LOCK_ROBUST` type mutex if:

`EBUSY` The mutex pointed to by *mp* was previously initialized and has not yet been destroyed.

`EINVAL` The mutex pointed to by *mp* was previously initialized with a different set of attribute flags.

The `mutex_trylock()` function will fail if:

`EBUSY` The mutex pointed to by *mp* is already locked.

The `mutex_lock()` and `mutex_trylock()` functions will fail for a `LOCK_RECURSIVE` mutex if:

`EAGAIN` The mutex could not be acquired because the maximum number of recursive locks for the mutex has been reached.

The `mutex_lock()` function will fail for a `LOCK_ERRORCHECK` and non-`LOCK_RECURSIVE` mutex if:

`EDEADLK` The caller already owns the mutex.

The `mutex_lock()` function may fail for a non-`LOCK_ERRORCHECK` and non-`LOCK_RECURSIVE` mutex if:

`EDEADLK` The caller already owns the mutex.

The `mutex_unlock()` function will fail for a `LOCK_ERRORCHECK` mutex if:

`EPERM` The caller does not own the mutex.

The `mutex_lock()` or `mutex_trylock()` functions will fail for `LOCK_ROBUST` type mutex if:

`EOWNERDEAD` The last owner of this mutex died while holding the mutex. This mutex is now owned by the caller. The caller must now attempt to make the state protected by the mutex consistent. If it is able to clean up the state, then it should restore the state of the mutex by calling `mutex_consistent()` and unlock the mutex. Subsequent calls to `mutex_lock()` will behave normally, as before. If the caller is not able to clean up the state, `mutex_consistent()` should not be called but the mutex should be unlocked. Subsequent calls to `mutex_lock()` will fail to acquire the mutex, returning with the error value `ENOTRECOVERABLE`. If the owner who acquired the lock with `EOWNERDEAD` dies, the next owner will acquire the lock with `EOWNERDEAD`.

ENOTRECOVERABLE The mutex trying to be acquired was protecting the state that has been left unrecoverable when the mutex's last owner could not make the state protected by the mutex consistent. The mutex has not been acquired. This condition occurs when the lock was previously acquired with EOWNERDEAD and the owner was not able to clean up the state and unlocked the mutex without calling `mutex_consistent()`.

The `mutex_consistent()` function will fail if:

EINVAL The caller does not own the mutex or the mutex is not a LOCK_ROBUST mutex having an inconsistent state (EOWNERDEAD).

Examples

Single Gate

The following example uses one global mutex as a gate-keeper to permit each thread exclusive sequential access to the code within the user-defined function "change_global_data." This type of synchronization will protect the state of shared data, but it also prohibits parallelism.

```
/* cc thisfile.c -lthread */
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#define NUM_THREADS 12
void *change_global_data(void *);     /*  for thr_create()   */
main(int argc,char * argv[])    {
       int i=0;
       for (i=0; i< NUM_THREADS; i++)    {
              thr_create(NULL, 0, change_global_data, NULL, 0, NULL);
       }
       while ((thr_join(NULL, NULL, NULL) == 0));
}

void * change_global_data(void *null){
       static mutex_t   Global_mutex;
       static int       Global_data = 0;
       mutex_lock(&Global_mutex);
       Global_data++;
       sleep(1);
       printf("%d is global data\n",Global_data);
       mutex_unlock(&Global_mutex);
       return NULL;
}
```

Multiple Instruction Single Data

The previous example, the mutex, the code it owns, and the data it protects was enclosed in one function. The next example uses C++ features to accommodate many functions that use just one mutex to protect one data:

```
/* CC thisfile.c -lthread   use C++ to compile*/

#define _REENTRANT
#include <stdlib.h>
#include <stdio.h>
#include <thread.h>
#include <errno.h>
#include <iostream.h>
#define NUM_THREADS 16
void *change_global_data(void *);     /*  for thr_create()   */

class Mutected {
       private:
               static mutex_t      Global_mutex;
               static int          Global_data;
       public:
               static int          add_to_global_data(void);
               static int          subtract_from_global_data(void);
};

int Mutected::Global_data = 0;
mutex_t Mutected::Global_mutex;

int Mutected::add_to_global_data()  {
       mutex_lock(&Global_mutex);
       Global_data++;
       mutex_unlock(&Global_mutex);
       return Global_data;
}

int Mutected::subtract_from_global_data()   {
       mutex_lock(&Global_mutex);
       Global_data--;
       mutex_unlock(&Global_mutex);
       return Global_data;
}

void
main(int argc,char * argv[])  {
       int i=0;
       for (i=0;i< NUM_THREADS;i++)  {
            thr_create(NULL,0,change_global_data,NULL,0,NULL);
       }
       while ((thr_join(NULL,NULL,NULL) == 0));
}

void * change_global_data(void *)    {
```

```
        static int switcher = 0;
        if ((switcher++ % 3) == 0)   /* one-in-three threads subtracts */
                cout << Mutected::subtract_from_global_data() << endl;
        else
                cout << Mutected::add_to_global_data() << endl;
        return NULL;
}
```

Interprocess Locking

A mutex can protect data that is shared among processes. The mutex would need to be initialized as `USYNC_PROCESS`. One process initializes the process-shared mutex and writes it to a file to be mapped into memory by all cooperating processes (see `mmap(2)`). Afterwards, other independent processes can run the same program (whether concurrently or not) and share mutex-protected data.

```
/* cc thisfile.c -lthread */
/* To execute, run the command line "a.out 0 &; a.out 1" */

#define _REENTRANT
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <thread.h>
#define INTERPROCESS_FILE "ipc-sharedfile"
#define NUM_ADDTHREADS 12
#define NUM_SUBTRACTTHREADS 10
#define INCREMENT '0'
#define DECREMENT '1'
typedef struct {
                mutex_t     Interprocess_mutex;
                int         Interprocess_data;
} buffer_t;
buffer_t *buffer;

void *add_interprocess_data(), *subtract_interprocess_data();
void create_shared_memory(), test_argv();
int zeroed[sizeof(buffer_t)];
int ipc_fd, i=0;

void
main(int argc,char * argv[]){
    test_argv(argv[1]);

    switch (*argv[1])  {
    case INCREMENT:
         /* Initializes the process-shared mutex */
         /* Should be run prior to running a DECREMENT process */
```

```
            create_shared_memory();
            ipc_fd = open(INTERPROCESS_FILE, O_RDWR);
            buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
                PROT_READ | PROT_WRITE, MAP_SHARED, ipc_fd, 0);
            buffer->Interprocess_data = 0;
            mutex_init(&buffer->Interprocess_mutex, USYNC_PROCESS,0);
            for (i=0; i< NUM_ADDTHREADS; i++)
            thr_create(NULL, 0, add_interprocess_data, argv[1],
                0, NULL);
            break;

        case DECREMENT:
            /* Should be run after the INCREMENT process has run. */
            while(ipc_fd = open(INTERPROCESS_FILE, O_RDWR)) == -1)
                sleep(1);
            buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
                PROT_READ | PROT_WRITE, MAP_SHARED, ipc_fd, 0);
            for (i=0; i< NUM_SUBTRACTTHREADS; i++)
            thr_create(NULL, 0, subtract_interprocess_data, argv[1],
                0, NULL);
            break;
        } /* end switch */

        while ((thr_join(NULL,NULL,NULL) == 0));
    } /* end main */

    void *add_interprocess_data(char argv_1[]){
        mutex_lock(&buffer->Interprocess_mutex);
        buffer->Interprocess_data++;
        sleep(2);
        printf("%d is add-interprocess data, and %c is argv1\n",
            buffer->Interprocess_data, argv_1[0]);
        mutex_unlock(&buffer->Interprocess_mutex);
        return NULL;
    }

    void *subtract_interprocess_data(char argv_1[])    {
        mutex_lock(&buffer->Interprocess_mutex);
        buffer->Interprocess_data--;
        sleep(2);
        printf("%d is subtract-interprocess data, and %c is argv1\n",
            buffer->Interprocess_data, argv_1[0]);
        mutex_unlock(&buffer->Interprocess_mutex);
        return NULL;
    }

    void create_shared_memory(){
```

```
    int i;
    ipc_fd = creat(INTERPROCESS_FILE, O_CREAT | O_RDWR );
    for (i=0; i<sizeof(buffer_t); i++){
        zeroed[i] = 0;
        write(ipc_fd, &zeroed[i],2);
    }
    close(ipc_fd);
    chmod(INTERPROCESS_FILE, S_IRWXU | S_IRWXG | S_IRWXO);
}

void test_argv(char argv1[])   {
    if (argv1 == NULL)   {
    printf("use 0 as arg1 for initial process\n \
    or use 1 as arg1 for the second process\n");
    exit(NULL);
    }
}
```

Solaris Interprocess Robust Locking

A mutex can protect data that is shared among processes robustly. The mutex would need to be initialized as USYNC_PROCESS | LOCK_ROBUST. One process initializes the robust process-shared mutex and writes it to a file to be mapped into memory by all cooperating processes (see mmap(2)). Afterwards, other independent processes can run the same program (whether concurrently or not) and share mutex-protected data.

The following example shows how to use a USYNC_PROCESS | LOCK_ROBUST type mutex.

```
/* cc thisfile.c -lthread */
 /* To execute, run the command line "a.out & a.out 1" */
 #include <sys/types.h>
 #include <sys/mman.h>
 #include <fcntl.h>
 #include <stdio.h>
 #include <thread.h>
 #define INTERPROCESS_FILE "ipc-sharedfile"
 typedef struct {
           mutex_t   Interprocess_mutex;
           int       Interprocess_data;
 } buffer_t;
 buffer_t *buffer;
 int make_date_consistent();
 void create_shared_memory();
 int zeroed[sizeof(buffer_t)];
 int ipc_fd, i=0;
 main(int argc,char * argv[])  {
     int rc;
     if (argc > 1) {
         while((ipc_fd = open(INTERPROCESS_FILE, O_RDWR)) == -1)
             sleep(1);
```

```
            buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
                     PROT_READ | PROT_WRITE, MAP_SHARED, ipc_fd, 0);
            mutex_init(&buffer->Interprocess_mutex,
                       USYNC_PROCESS | LOCK_ROBUST,0);
        } else {
            create_shared_memory();
            ipc_fd = open(INTERPROCESS_FILE, O_RDWR);
            buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
                  PROT_READ | PROT_WRITE, MAP_SHARED, ipc_fd, 0);
            buffer->Interprocess_data = 0;
            mutex_init(&buffer->Interprocess_mutex,
                       USYNC_PROCESS | LOCK_ROBUST,0);
        }
        for(;;) {
            rc = mutex_lock(&buffer->Interprocess_mutex);
            switch (rc) {
                case EOWNERDEAD:
                  /*
                   * The lock is acquired.
                   * The last owner died holding the lock.
                   * Try to make the state associated with
                   * the mutex consistent.
                   * If successful, make the robust lock consistent.
                   */
                  if (make_data_consistent())
                      mutex_consistent(&buffer->Interprocess_mutex);
                      mutex_unlock(&buffer->Interprocess_mutex);
                      break;
                case ENOTRECOVERABLE:
                  /*
                   * The lock is not acquired.
                   * The last owner got the mutex with EOWNERDEAD
                   * and failed to make the data consistent.
                   * There is no way to recover, so just exit.
                   */
                  exit(1);
                case 0:
                  /*
                   * There is no error - data is consistent.
                   * Do something with data.
                   */
                  mutex_unlock(&buffer->Interprocess_mutex);
                  break;
            }
       }
    } /* end main */
    void create_shared_memory() {
```

```
      int i;
      ipc_fd = creat(INTERPROCESS_FILE, O_CREAT | O_RDWR );
      for (i=0; i<sizeof(buffer_t); i++) {
           zeroed[i] = 0;
           write(ipc_fd, &zeroed[i],2);
      }
      close(ipc_fd);
      chmod(INTERPROCESS_FILE, S_IRWXU | S_IRWXG | S_IRWXO);
 }

 /* return 1 if able to make data consistent, otherwise 0. */
 int make_data_consistent () {
       buffer->Interprocess_data = 0;
       return (1);
 }
```

Dynamically Allocated Mutexes

The following example allocates and frees memory in which a mutex is embedded.

```
struct record {
        int field1;
        int field2;
        mutex_t m;
} *r;
r = malloc(sizeof(struct record));
mutex_init(&r->m, USYNC_THREAD, NULL);
/*
  * The fields in this record are accessed concurrently
  * by acquiring the embedded lock.
  */
```

The thread execution in this example is as follows:

Thread 1 executes:

```
...
mutex_lock(&r->m);
r->field1++;
mutex_unlock(&r->m);
...
```

Thread 2 executes:

```
...
mutex_lock(&r->m);
localvar = r->field1;
mutex_unlock(&r->m);
...
```

Later, when a thread decides to free the memory pointed to by *r*, the thread should call `mutex_destroy( )` on the mutexes in this memory.

In the following example, the main thread can do a `thr_join( )` on both of the above threads. If there are no other threads using the memory in *r*, the main thread can now safely free *r*:

```
for (i = 0; i < 2; i++)
       thr_join(0, 0, 0);
mutex_destroy(&r->m);   /* first destroy mutex */
```

```
free(r);                /* then free memory */
```

If the mutex is not destroyed, the program could have memory leaks.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

mmap(2), shmop(2), pthread_mutexattr_getprioceiling(3C), pthread_mutexattr_getprotocol(3C), pthread_mutexattr_getrobust(3C), pthread_mutexattr_gettype(3C), pthread_mutex_getprioceiling(3C), pthread_mutex_init(3C), attributes(5), mutex(5), standards(5)

Notes

Previous releases of Solaris provided the USYNC_PROCESS_ROBUST mutex type. This type is now deprecated but is still supported for source and binary compatibility. When passed to mutex_init(), it is transformed into (USYNC_PROCESS | LOCK_ROBUST). The former method for restoring a USYNC_PROCESS_ROBUST mutex to a consistent state was to reinitialize it by calling mutex_init(). This method is still supported for source and binary compatibility, but the proper method is to call mutex_consistent().

The USYNC_PROCESS_ROBUST type permitted an alternate error value, ELOCKUNMAPPED, to be returned by mutex_lock() if the process containing a locked robust mutex unmapped the memory containing the mutex or performed one of the exec(2) functions. The ELOCKUNMAPPED error value implies all of the consequences of the EOWNERDEAD error value and as such is just a synonym for EOWNERDEAD. For full source and binary compatibility, the ELOCKUNMAPPED error value is still returned from mutex_lock() in these circumstances, but only if the mutex was initialized with the USYNC_PROCESS_ROBUST type. Otherwise, EOWNERDEAD is returned in these circumstances.

The mutex_lock(), mutex_unlock(), and mutex_trylock() functions do not validate the mutex type. An uninitialized mutex or a mutex with an invalid type does not return EINVAL. Interfaces for mutexes with an invalid type have unspecified behavior.

Uninitialized mutexes that are allocated locally could contain junk data. Such mutexes need to be initialized using mutex_init().

By default, if multiple threads are waiting for a mutex, the order of acquisition is undefined.

Name nanosleep – high resolution sleep

Synopsis

```
#include <time.h>

int nanosleep(const struct timespec *rqtp,
     struct timespec *rmtp);
```

Description The `nanosleep()` function causes the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time will not be less than the time specified by *rqtp*, as measured by the system clock, `CLOCK_REALTIME`.

The use of the `nanosleep()` function has no effect on the action or blockage of any signal.

Return Values If the `nanosleep()` function returns because the requested time has elapsed, its return value is `0`.

If the `nanosleep()` function returns because it has been interrupted by a signal, the function returns a value of −1 and sets `errno` to indicate the interruption. If the *rmtp* argument is non-`NULL`, the `timespec` structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* argument is `NULL`, the remaining time is not returned.

If `nanosleep()` fails, it returns −1 and sets `errno` to indicate the error.

Errors The `nanosleep()` function will fail if:

`EINTR` The `nanosleep()` function was interrupted by a signal.

`EINVAL` The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

`ENOSYS` The `nanosleep()` function is not supported by this implementation.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also sleep(3C), time.h(3HEAD), attributes(5), standards(5)

Name ndbm, dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey, dbm_open, dbm_store – database functions

Synopsis

```
#include <ndbm.h>

int dbm_clearerr(DBM *db);

void dbm_close(DBM *db);

int dbm_delete(DBM *db, datum key);

int dbm_error(DBM *db);

datum dbm_fetch(DBM *db, datum key);

datum dbm_firstkey(DBM *db);

datum dbm_nextkey(DBM *db);

DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);

int dbm_store(DBM *db, datum key, datum content, int store_mode);
```

Description These functions create, access and modify a database. They maintain *key*/*content* pairs in a database. The functions will handle large databases (up to a billion blocks) and will access a keyed item in one or two file system accesses. This package replaces the earlier `dbm` library, which managed only a single database.

*key*s and *content*s are described by the `datum` typedef. A `datum` consists of at least two members, `dptr` and `dsize`. The `dptr` member points to an object that is `dsize` bytes in length. Arbitrary binary data, as well as ASCII character strings, may be stored in the object pointed to by `dptr`.

The database is stored in two files. One file is a directory containing a bit map of keys and has `.dir` as its suffix. The second file contains all data and has `.pag` as its suffix.

The `dbm_open()` function opens a database. The `file` argument to the function is the pathname of the database. The function opens two files named `file.dir` and `file.pag`. The *open_flags* argument has the same meaning as the *flags* argument of open(2) except that a database opened for write-only access opens the files for read and write access. The *file_mode* argument has the same meaning as the third argument of open(2).

The `dbm_close()` function closes a database. The argument *db* must be a pointer to a `dbm` structure that has been returned from a call to `dbm_open()`.

The `dbm_fetch()` function reads a record from a database. The argument *db* is a pointer to a database structure that has been returned from a call to `dbm_open()`. The argument *key* is a `datum` that has been initialized by the application program to the value of the key that matches the key of the record the program is fetching.

The `dbm_store()` function writes a record to a database. The argument *db* is a pointer to a database structure that has been returned from a call to `dbm_open()`. The argument *key* is a

datum that has been initialized by the application program to the value of the key that identifies (for subsequent reading, writing or deleting) the record the program is writing. The argument *content* is a datum that has been initialized by the application program to the value of the record the program is writing. The argument *store_mode* controls whether dbm_store() replaces any pre-existing record that has the same key that is specified by the *key* argument. The application program must set *store_mode* to either DBM_INSERT or DBM_REPLACE. If the database contains a record that matches the *key* argument and *store_mode* is DBM_REPLACE, the existing record is replaced with the new record. If the database contains a record that matches the *key* argument and *store_mode* is DBM_INSERT, the existing record is not replaced with the new record. If the database does not contain a record that matches the *key* argument and *store_mode* is either DBM_INSERT or DBM_REPLACE, the new record is inserted in the database.

The dbm_delete() function deletes a record and its key from the database. The argument *db* is a pointer to a database structure that has been returned from a call to dbm_open(). The argument *key* is a datum that has been initialized by the application program to the value of the key that identifies the record the program is deleting.

The dbm_firstkey() function returns the first key in the database. The argument *db* is a pointer to a database structure that has been returned from a call to dbm_open().

The dbm_nextkey() function returns the next key in the database. The argument *db* is a pointer to a database structure that has been returned from a call to dbm_open(). The dbm_firstkey() function must be called before calling dbm_nextkey(). Subsequent calls to dbm_nextkey() return the next key until all of the keys in the database have been returned.

The dbm_error() function returns the error condition of the database. The argument *db* is a pointer to a database structure that has been returned from a call to dbm_open().

The dbm_clearerr() function clears the error condition of the database. The argument *db* is a pointer to a database structure that has been returned from a call to dbm_open().

These database functions support key/content pairs of at least 1024 bytes.

Return Values The dbm_store() and dbm_delete() functions return 0 when they succeed and a negative value when they fail.

The dbm_store() function returns 1 if it is called with a *flags* value of DBM_INSERT and the function finds an existing record with the same key.

The dbm_error() function returns 0 if the error condition is not set and returns a non-zero value if the error condition is set.

The return value of dbm_clearerr() is unspecified .

The dbm_firstkey() and dbm_nextkey() functions return a key datum. When the end of the database is reached, the dptr member of the key is a null pointer. If an error is detected, the dptr member of the key is a null pointer and the error condition of the database is set.

The `dbm_fetch()` function returns a content `datum`. If no record in the database matches the key or if an error condition has been detected in the database, the `dptr` member of the content is a null pointer.

The `dbm_open()` function returns a pointer to a database structure. If an error is detected during the operation, `dbm_open()` returns a (`DBM *`)0.

Errors No errors are defined.

Usage The following code can be used to traverse the database:

```
for(key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

The `dbm_` functions provided in this library should not be confused in any way with those of a general-purpose database management system. These functions do not provide for multiple search keys per entry, they do not protect against multi-user access (in other words they do not lock records or files), and they do not provide the many other useful database functions that are found in more robust database management systems. Creating and updating databases by use of these functions is relatively slow because of data copies that occur upon hash collisions. These functions are useful for applications requiring fast lookup of relatively static information that is to be indexed by a single key.

The `dptr` pointers returned by these functions may point into static storage that may be changed by subsequent calls.

The `dbm_delete()` function does not physically reclaim file space, although it does make it available for reuse.

After calling `dbm_store()` or `dbm_delete()` during a pass through the keys by `dbm_firstkey()` and `dbm_nextkey()`, the application should reset the database by calling `dbm_firstkey()` before again calling `dbm_nextkey()`.

Examples **EXAMPLE 1** Using the Database Functions

The following example stores and retrieves a phone number, using the name as the key. Note that this example does not include error checking.

```
#include <ndbm.h>
#include <stdio.h>
#include <fcntl.h>
#define NAME      "Bill"
#define PHONE_NO          "123-4567"
#define DB_NAME   "phones"
main()
{
     DBM *db;
     datum name = {NAME, sizeof (NAME)};
     datum put_phone_no = {PHONE_NO, sizeof (PHONE_NO)};
     datum get_phone_no;
```

EXAMPLE 1 Using the Database Functions *(Continued)*

```
    /* Open the database and store the record */
    db = dbm_open(DB_NAME, O_RDWR | O_CREAT, 0660);
    (void) dbm_store(db, name, put_phone_no, DBM_INSERT);
    /* Retrieve the record */
    get_phone_no = dbm_fetch(db, name);
    (void) printf("Name: %s, Phone Number: %s\n", name.dptr,
    get_phone_no.dptr);
    /* Close the database */
    dbm_close(db);
    return (0);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	See `standards(5)`.

See Also

`ar(1)`, `cat(1)`, `cp(1)`, `tar(1)`, `open(2)`, `netconfig(4)`, `attributes(5)`, `standards(5)`

Notes

The `.pag` file will contain holes so that its apparent size may be larger than its actual content. Older versions of the UNIX operating system may create real file blocks for these holes when touched. These files cannot be copied by normal means (`cp(1)`, `cat(1)`, `tar(1)`, `ar(1)`) without filling in the holes.

The sum of the sizes of a *key*/*content* pair must not exceed the internal block size (currently 1024 bytes). Moreover all *key*/*content* pairs that hash together must fit on a single block. `dbm_store()` will return an error in the event that a disk block fills with inseparable data.

The order of keys presented by `dbm_firstkey()` and `dbm_nextkey()` depends on a hashing function.

There are no interlocks and no reliable cache flushing; thus concurrent updating and reading is risky.

The database files (`file.dir` and `file.pag`) are binary and are architecture-specific (for example, they depend on the architecture's byte order.) These files are not guaranteed to be portable across architectures.

Name nl_langinfo – language information

Synopsis

```
#include <langinfo.h>

char *nl_langinfo(nl_item item);
```

Description The `nl_langinfo()` function returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area defined in the programs locale. The manifest constant names and values of *item* are defined by <`langinfo.h`>. For example:

```
nl_langinfo (ABDAY_1);
```

would return a pointer to the string "`Dim`" if the identified language was French and a French locale was correctly installed; or "`Sun`" if the identified language was English.

Return Values If `setlocale(3C)` has not been called successfully, or if data for a supported language is either not available, or if *item* is not defined therein, then `nl_langinfo()` returns a pointer to the corresponding string in the C locale. In all locales, `nl_langinfo()` returns a pointer to an empty string if *item* contains an invalid setting.

Usage The `nl_langinfo()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `setlocale(3C)`, `langinfo.h(3HEAD)`, `nl_types.h(3HEAD)`, `attributes(5)`, `standards(5)`

Warnings The array pointed to by the return value should not be modified by the program. Subsequent calls to `nl_langinfo()` may overwrite the array.

Name offsetof – offset of structure member

Synopsis

```
#include <stddef.h>

size_t offsetof(type, member-designator);
```

Description The `offsetof()` macro defined in `<stddef.h>` expands to an integral constant expression that has type `size_t`. The value of this expression is the offset in bytes to the structure member (designated by *member-designator*) from the beginning of its structure (designated by *type*).

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `attributes(5)`

Name opendir, fdopendir – open directory

Synopsis

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

DIR *opendir(const char *dirname);

DIR *fdopendir(int fildes);
```

Description The `opendir()` function opens a directory stream corresponding to the directory named by the *dirname* argument.

The `fdopendir()` function opens a directory stream for the directory file descriptor *fildes*. The directory file descriptor should not be used or closed following a successful function call, as this might cause undefined results from future operations on the directory stream obtained from the call. Use `closedir(3C)` to close a directory stream.

The directory stream is positioned at the first entry. If the type `DIR` is implemented using a file descriptor, applications will only be able to open up to a total of {`OPEN_MAX`} files and directories. A successful call to any of the `exec` functions will close any directory streams that are open in the calling process. See `exec(2)`.

Return Values Upon successful completion, `opendir()` and `fdopendir()` return a pointer to an object of type `DIR`. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

Errors The `opendir()` function will fail if:

`EACCES` Search permission is denied for the component of the path prefix of *dirname* or read permission is denied for *dirname*.

`ELOOP` Too many symbolic links were encountered in resolving *path*.

`ENAMETOOLONG` The length of the *dirname* argument exceeds {`PATH_MAX`}, or a path name component is longer than {`NAME_MAX`} while {`_POSIX_NO_TRUNC`} is in effect.

`ENOENT` A component of *dirname* does not name an existing directory or *dirname* is an empty string.

`ENOTDIR` A component of *dirname* is not a directory.

The `fdopendir()` function will fail if:

`ENOTDIR` The file descriptor *fildes* does not reference a directory.

The `opendir()` function may fail if:

`EMFILE` There are {`OPEN_MAX`} file descriptors currently open in the calling process.

`ENAMETOOLONG` Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

ENFILE Too many files are currently open on the system.

Usage The `opendir()` and `fdopendir()` functions should be used in conjunction with `readdir`(3C), `closedir`(3C) and `rewinddir`(3C) to examine the contents of the directory (see the `EXAMPLES` section in `readdir`(3C)). This method is recommended for portability.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	For opendir, see `standards`(5).

See Also `lstat`(2), `symlink`(2), `closedir`(3C), `readdir`(3C), `rewinddir`(3C), `scandir`(3C), `attributes`(5), `standards`(5)

Name perror, errno – print system error messages

Synopsis

```
#include <stdio.h>

void perror(const char *s)

#include <errno.h>

int errno;
```

Description The `perror()` function produces a message on the standard error output (file descriptor 2) describing the last error encountered during a call to a system or library function. The argument string *s* is printed, followed by a colon and a blank, followed by the message and a NEWLINE character. If *s* is a null pointer or points to a null string, the colon is not printed. The argument string should include the name of the program that incurred the error. The error number is taken from the external variable `errno`, which is set when errors occur but not cleared when non-erroneous calls are made. See `Intro(2)`.

In the case of multithreaded applications, the `-mt` option must be specified on the command line at compilation time (see `threads(5)`). When the `-mt` option is specified, `errno` becomes a macro that enables each thread to have its own `errno`. This `errno` macro can be used on either side of the assignment as though it were a variable.

Usage Messages printed from this function are in the native language specified by the `LC_MESSAGES` locale category. See `setlocale(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `Intro(2)`, `fmtmsg(3C)`, `gettext(3C)`, `setlocale(3C)`, `strerror(3C)`, `attributes(5)`, `standards(5)`, `threads(5)`

Name pfmt – display error message in standard format

Synopsis

```
#include <pfmt.h>

int pfmt(FILE *stream, long flags, char *format, ... /* arg */);
```

Description The `pfmt()` retrieves a format string from a locale-specific message database (unless `MM_NOGET` is specified) and uses it for `printf(3C)` style formatting of *args*. The output is displayed on *stream*.

The `pfmt()` function encapsulates the output in the standard error message format (unless `MM_NOSTD` is specified, in which case the output is similar to `printf()`).

If the `printf()` format string is to be retrieved from a message database, the `format` argument must have the following structure:

<*catalog*>:<*msgnum*>:<*defmsg*>.

If `MM_NOGET` is specified, only the *defmsg* field must be specified.

The *catalog* field is used to indicate the message database that contains the localized version of the format string. This field must be limited to 14 characters selected from the set of all characters values, excluding `\0` (null) and the ASCII codes for `/` (slash) and `:` (colon).

The *msgnum* field is a positive number that indicates the index of the string into the message database.

If the catalog does not exist in the locale (specified by the last call to `setlocale(3C)` using the `LC_ALL` or `LC_MESSAGES` categories), or if the message number is out of bound, `pfmt()` will attempt to retrieve the message from the C locale. If this second retrieval fails, `pfmt()` uses the *defmsg* field of the `format` argument.

If *catalog* is omitted, `pfmt()` will attempt to retrieve the string from the default catalog specified by the last call to `setcat(3C)`. In this case, the `format` argument has the following structure:

:<*msgnum*>:<*defmsg*>.

The `pfmt()` will output `Message not found!!\n` as format string if *catalog* is not a valid catalog name, if no catalog is specified (either explicitely or with `setcat()`), if *msgnum* is not a valid number, or if no message could be retrieved from the message databases and *defmsg* was omitted.

The *flags* argument determine the type of output (such as whether the `format` should be interpreted as is or encapsulated in the standard message format), and the access to message catalogs to retrieve a localized version of `format`.

The *flags* argument is composed of several groups, and can take the following values (one from each group):

Output format control

`MM_NOSTD` Do not use the standard message format, interpret `format` as `printf()` `format`. Only *catalog access control flags* should be specified if `MM_NOSTD` is used; all other flags will be ignored.

`MM_STD` Output using the standard message format (default value 0).

Catalog access control

`MM_NOGET` Do not retrieve a localized version of `format`. In this case, only the *defmsg* field of the `format` is specified.

`MM_GET` Retrieve a localized version of `format` from the *catalog*, using *msgid* as the index and *defmsg* as the default message (default value 0).

Severity (standard message format only)

`MM_HALT` Generate a localized version of `HALT`, but do not halt the machine.

`MM_ERROR` Generate a localized version of `ERROR` (default value 0).

`MM_WARNING` Generate a localized version of `WARNING`.

`MM_INFO` Generate a localized version of `INFO`.

Additional severities can be defined. Add-on severities can be defined with number-string pairs with numeric values from the range [5-255], using `addsev(3C)`. The specified severity will be generated from the bitwise OR operation of the numeric value and other *flags* If the severity is not defined, `pfmt()` uses the string `SEV=`*N*, where *N* is replaced by the integer severity value passed in *flags*.

Multiple severities passed in *flags* will not be detected as an error. Any combination of severities will be summed and the numeric value will cause the display of either a severity string (if defined) or the string `SEV=`*N* (if undefined).

Action

`MM_ACTION` Specify an action message. Any severity value is superseded and replaced by a localized version of `TO FIX`.

Standard Error Message Format

The `pfmt()` function displays error messages in the following format:

label: *severity*: *text*

If no *label* was defined by a call to `setlabel(3C)`, the message is displayed in the format:

severity: *text*

If `pfmt()` is called twice to display an error message and a helpful *action* or recovery message, the output can look like:

label: *severity*: *textlabel*: TO FIX: *text*

Return Values Upon success, `pfmt()` returns the number of bytes transmitted. Upon failure, it returns a negative value:

−1 Write error to *stream*.

Examples **EXAMPLE 1** Example of `pfmt()` function.

Example 1:

```
setlabel("UX:test");
pfmt(stderr, MM_ERROR, "test:2:Cannot open file: %s\n",
     strerror(errno));

displays the message:

UX:test: ERROR: Cannot open file: No such file or directory
```

Example 2:

```
setlabel("UX:test");
setcat("test");
pfmt(stderr, MM_ERROR, ":10:Syntax error\n");
pfmt(stderr, MM_ACTION, "55:Usage ...\n");
```

displays the message

```
UX:test: ERROR: Syntax error
UX:test: TO FIX: Usage ...
```

Usage Since it uses gettxt(3C), `pfmt()` should not be used.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-safe

See Also addsev(3C), gettxt(3C), lfmt(3C), printf(3C), setcat(3C), setlabel(3C), setlocale(3C), attributes(5), environ(5)

Name plock – lock or unlock into memory process, text, or data

Synopsis

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

int plock(int op);
```

Description The `plock()` function allows the calling process to lock or unlock into memory its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock). Locked segments are immune to all routine swapping. The effective user ID of the calling process must be super-user to use this call.

The `plock()` function performs the function specified by *op*:

`PROCLOCK` Lock text and data segments into memory (process lock).

`TXTLOCK` Lock text segment into memory (text lock).

`DATLOCK` Lock data segment into memory (data lock).

`UNLOCK` Remove locks.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `plock()` function fails and does not perform the requested operation if:

`EAGAIN` Not enough memory.

`EINVAL` The *op* argument is equal to `PROCLOCK` and a process lock, a text lock, or a data lock already exists on the calling process; the *op* argument is equal to `TXTLOCK` and a text lock or a process lock already exists on the calling process; the *op* argument is equal to `DATLOCK` and a data lock or a process lock already exists on the calling process; or the *op* argument is equal to `UNLOCK` and no lock exists on the calling process.

`EPERM` The {`PRIV_PROC_LOCK_MEMORY`} privilege is not asserted in the effective set of the calling process.

Usage The `mlock(3C)` and `mlockall(3C)` functions are the preferred interfaces for process locking.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `exec(2)`, `exit(2)`, `fork(2)`, `memcntl(2)`, `mlock(3C)`, `mlockall(3C)`, `attributes(5)`

Name popen, pclose – initiate a pipe to or from a process

Synopsis

```
#include <stdio.h>

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

int pclose(FILE *stream);
```

Description The `popen()` function creates a pipe between the calling program and the command to be executed. The arguments to `popen()` are pointers to null-terminated strings. The *command* argument consists of a shell command line. The *mode* argument is an I/O mode, either `r` for reading or *w* for writing. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is *w*, by writing to the file *stream* (see `Intro(3)`); and one can read from the standard output of the command, if the I/O mode is `r`, by reading from the file *stream*. Because open files are shared, a type `r` command may be used as an input filter and a type *w* as an output filter. A trailing `F` character can also be included in the *mode* argument as described in `fopen(3C)` to enable extended FILE facility.

The environment of the executed command will be as if a child process were created within the `popen()` call using `fork(2)`. If the application is standard-conforming (see `standards(5)`), the child is created as if invoked with the call:

```
execl("/usr/xpg4/bin/sh", "sh", "-c",command, (char *)0);
```

otherwise, the child is created as if invoked with the call:

```
execl("/usr/bin/sh", "sh", "-c",command, (char *)0);
```

The `pclose()` function closes a stream opened by `popen()` by closing the pipe. It waits for the associated process to terminate and returns the termination status of the process running the command language interpreter. This is the value returned by `waitpid(3C)`. See `wait.h(3HEAD)` for more information on termination status. If, however, a call to `waitpid()` with a *pid* argument equal to the process ID of the command line interpreter causes the termination status to be unavailable to `pclose()`, then `pclose()` returns −1 with `errno` set to `ECHILD` to report this condition.

Return Values Upon successful completion, `popen()` returns a pointer to an open stream that can be used to read or write to the pipe. Otherwise, it returns a null pointer and may set `errno` to indicate the error.

Upon successful completion, `pclose()` returns the termination status of the command language interpreter as returned by `waitpid()`. Otherwise, it returns −1 and sets `errno` to indicate the error.

Errors The `pclose()` function will fail if:

`ECHILD` The status of the child process could not be obtained, as described in the DESCRIPTION.

The `popen()` function may fail if:

EMFILE There are currently FOPEN_MAX or STREAM_MAX streams open in the calling process.

EINVAL The *mode* argument is invalid.

The popen() function may also set errno values as described by fork(2) or pipe(2).

Usage If the original and popen() processes concurrently read or write a common file, neither should use buffered I/O. Problems with an output filter may be forestalled by careful buffer flushing, for example, with fflush() (see fclose(3C)). A security hole exists through the IFS and PATH environment variables. Full pathnames should be used (or PATH reset) and IFS should be set to space and tab (" \t").

Even if the process has established a signal handler for SIGCHLD, it will be called when the command terminates. Even if another thread in the same process issues a wait(3C) call, it will interfere with the return value of pclose(). Even if the process's signal handler for SIGCHLD has been set to ignore the signal, there will be no effect on pclose().

Examples **EXAMPLE 1** popen() example

The following program will print on the standard output (see stdio(3C)) the names of files in the current directory with a .c suffix.

```
#include <stdio.h>
#include <stdlib.h>
main( )
{
        char *cmd = "/usr/bin/ls *.c";
        char buf[BUFSIZ];
        FILE *ptr;

        if ((ptr = popen(cmd, "r")) != NULL) {
                while (fgets(buf, BUFSIZ, ptr) != NULL)
                        (void) printf("%s", buf);
                (void) pclose(ptr);
        }
        return 0;
}
```

EXAMPLE 2 system() replacement

The following function can be used in a multithreaded process in place of the most common usage of the Unsafe system(3C) function:

```
int my_system(const char *cmd)
{
        FILE *p;

        if ((p = popen(cmd, "w")) == NULL)
                return (-1);
```

EXAMPLE 2 `system()` replacement *(Continued)*

```
        return (pclose(p));
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See below.

For `pclose()` and all aspects of `popen()` except the `F` character in the *mode* argument, see `standards(5)`.

See Also

`ksh(1)`, `pipe(2)`, `fclose(3C)`, `fopen(3C)`, `posix_spawn(3C)`, `stdio(3C)`, `system(3C)`, `wait(3C)`, `waitpid(3C)`, `wait.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name port_alert – set a port in alert mode

Synopsis

```
#include <port.h>

int port_alert(int port, int flags, int events, void *user);
```

Description The `port_alert()` function transitions a port into or out of alert mode. A port in alert mode immediately awakens all threads blocked in `port_get(3C)` or `port_getn(3C)`. These threads return with an alert notification that consists of a single `port_event_t` structure with the source `PORT_SOURCE_ALERT`. Subsequent threads trying to retrieve events from a port that is in alert mode will return immediately with the alert notification.

A port is transitioned into alert mode by calling the `port_alert()` function with a non-zero *events* parameter. The specified *events* and *user* parameters will be made available in the `portev_events` and the `portev_user` members of the alert notification, respectively. The *flags* argument determines the mode of operation of the alert mode:

- If *flags* is set to `PORT_ALERT_SET`, `port_alert()` sets the port in alert mode independent of the current state of the port. The `portev_events` and `portev_user` members are set or updated accordingly.
- If *flags* is set to `PORT_ALERT_UPDATE` and the port is not in alert mode, `port_alert()` transitions the port into alert mode. The `portev_events` and `portev_user` members are set accordingly.
- If *flags* is set to `PORT_ALERT_UPDATE` and the port is already in alert mode, `port_alert()` returns with an error value of `EBUSY`.

`PORT_ALERT_SET` and `PORT_ALERT_UPDATE` are mutually exclusive.

A port is transitioned out of alert mode by calling the `port_alert()` function with a zero *events* parameter.

Events can be queued to a port that is in alert mode, but they will not be retrievable until the port is transitioned out of alert mode.

Return Values Upon succesful completion, 0 is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors The `port_alert()` function will fail if:

`EBADF` The port identifier is not valid.

`EBADFD` The *port* argument is not an event port file descriptor.

`EBUSY` The port is already in alert mode.

`EINVAL` Mutually exclusive *flags* are set.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs, system/header
Interface Stability	Committed
MT-Level	Safe

See Also port_associate(3C), port_create(3C), port_get(3C), port_send(3C), attributes(5)

Name port_associate, port_dissociate – associate or dissociate the object with the port

Synopsis

```
#include <port.h>

int port_associate(int port, int source, uintptr_t object,
     int events, void *user);

int port_dissociate(int port, int source, uintptr_t object);
```

Description The `port_associate()` function associates specific *events* of a given *object* with a *port*. Only objects associated with a particular port are able to generate events that can be retrieved using `port_get(3C)` or `port_getn(3C)`. The delivery event has its `portev_user` member set to the value specified in the *user* parameter. If the specified object is already associated with the specified port, the `port_associate()` function serves to update the *events* and *user* arguments of the association. The `port_dissociate()` function removes the association of an object with a port.

The objects that can be associated with a port by way of the `port_associate()` function are objects of type `PORT_SOURCE_FD` and `PORT_SOURCE_FILE`. Objects of other types have type-specific association mechanisms. A `port_notify_t` structure, defined in `<port.h>`, is used to specify the event port and an application-defined cookie to associate with these event sources. See `port_create(3C)` and `signal.h(3HEAD)`.

The `port_notify_t` structure contains the following members:

```
int       portntfy_port;  /* bind request(s) to port */
void      *portntfy_user; /* user defined cookie */
```

Objects of type `PORT_SOURCE_FD` are file descriptors. The event types for `PORT_SOURCE_FD` objects are described in `poll(2)`. At most one event notification will be generated per associated file descriptor. For example, if a file descriptor is associated with a port for the `POLLRDNORM` event and data is available on the file descriptor at the time the `port_associate()` function is called, an event is immediately sent to the port. If data is not yet available, one event is sent to the port when data first becomes available.

When an event for a `PORT_SOURCE_FD` object is retrieved, the object no longer has an association with the port. The event can be processed without the possibility that another thread can retrieve a subsequent event for the same object. After processing of the file descriptor is completed, the `port_associate()` function can be called to reassociate the object with the port.

Objects of type `PORT_SOURCE_FILE` are pointer to the structure `file_obj` defined in `<sys/port.h>`. This event source provides event notification when the specified file/directory is accessed or modified or when its status changes. The path name of the file/directory to be watched is passed in the `struct file_obj` along with the `access`, `modification`, and `change` time stamps acquired from a `stat(2)` call. If the file name is a symbolic links, it is followed by default. The `FILE_NOFOLLOW` needs to be passed in along with the specified events if the symbolic link itself needs to be watched and `lstat()` needs to be used to get the file status of the symbolic link file.

The `struct file_obj` contains the following elements:

```
timestruc_t     fo_atime;  /* Access time got from stat() */
timestruc_t     fo_mtime;  /* Modification time from stat() */
timestruc_t     fo_ctime;  /* Change time from stat() */
char            *fo_name;  /* Pointer to a null terminated path name */
```

At the time the `port_associate()` function is called, the time stamps passed in the structure `file_obj` are compared with the file or directory's current time stamps and, if there has been a change, an event is immediately sent to the port. If not, an event will be sent when such a change occurs.

The event types that can be specified at `port_associate()` time for `PORT_SOURCE_FILE` are `FILE_ACCESS`, `FILE_MODIFIED`, and `FILE_ATTRIB`, corresponding to the three time stamps. An `fo_atime` change results in the `FILE_ACCESS` event, an `fo_mtime` change results in the `FILE_MODIFIED` event, and an `fo_time` change results in the `FILE_ATTRIB` event.

The following exception events are delivered when they occur. These event types cannot be filtered.

```
FILE_DELETE       /* Monitored file/directory was deleted */
FILE_RENAME_TO    /* Monitored file/directory was renamed */
FILE_RENAME_FROM  /* Monitored file/directory was renamed */
UNMOUNTED         /* Monitored file system got unmounted */
MOUNTEDOVER       /* Monitored file/directory was mounted over */
```

At most one event notification will be generated per associated `file_obj`. When the event for the associated `file_obj` is retrieved, the object is no longer associated with the port. The event can be processed without the possibility that another thread can retrieve a subsequent event for the same object. The `port_associate()` can be called to reassociate the `file_obj` object with the port.

The association is also removed if the port gets closed or when `port_dissociate()` is called.

The parent and child processes are allowed to retrieve events from file descriptors shared after a call to `fork(2)`. The process performing the first association with a port (parent or child process) is designated as the owner of the association. Only the owner of an association is allowed to dissociate the file descriptor from a port. The association is removed if the owner of the association closes the port .

On NFS file systems, events from only the client side (local) access/modifications to files or directories will be delivered.

Return Values Upon succesful completion, 0 is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `port_associate()` and `port_dissociate()` functions will fail if:

`EBADF` The *port* identifier is not valid.

`EBADFD` The *source* argument is of type `PORT_SOURCE_FD` and the object argument is not a valid file descriptor.

`EINVAL` The *source* argument is not valid.

The `port_associate()` function will fail if:

`EACCES` The source argument is `PORT_SOURCE_FILE` and, Search permission is denied on a component of path prefix or the file exists and the permissions, corresponding to the events argument, are denied.

`EAGAIN` The maximum number of objects associated with the port was exceeded. The maximum allowable number of events or association of objects per port is the minimum value of the `process.max-port-events` resource control at the time `port_create(3C)` was used to create the port. See `setrctl(2)` and `rctladm(1M)` for information on using resource controls.

The number of objects associated with a port is composed of all supported resource types. Some of the source types do not explicitly use the `port_associate()` function.

`ENOENT` The source argument is `PORT_SOURCE_FILE` and the file does not exist or the path prefix does not exist or the path points to an empty string.

`ENOMEM` The physical memory limits of the system have been exceeded.

`ENOTSUP` The source argument is `PORT_SOURCE_FILE` and the file system on which the specified file resides, does not support watching for file events notifications.

The `port_dissociate()` function will fail if:

`EACCES` The process is not the owner of the association.

`ENOENT` The specified object is not associated with the port.

Examples **EXAMPLE 1** Retrieve data from a pipe file descriptor.

The following example retrieves data from a pipe file descriptor.

```
#include <port.h>

int               port;
int               fd;
int               error;
int               index;
void              *mypointer;
port_event_t      pev;
```

EXAMPLE 1 Retrieve data from a pipe file descriptor. *(Continued)*

```
struct timespec_t timeout;
char              rbuf[STRSIZE];
int               fds[MAXINDEX];

/* create a port */
port = port_create();

for (index = 0; index < MAXINDEX; index++) {
    error = mkfifo(name[index], S_IRWXU | S_IRWXG | S_IRWXO);
    if (error)
            /* handle error code */
    fds[index] = open(name[index], O_RDWR);

    /* associate pipe file descriptor with the port */
    error = port_associate(port, PORT_SOURCE_FD, fds[index],
        POLLIN, mypointer);
}
...
timeout.tv_sec = 1;     /* user defined */
timeout.tv_nsec = 0;

/* loop to retrieve data from the list of pipe file descriptors */
for (...) {
    /* retrieve a single event */
    error = port_get(port, &pev, &timeout);
    if (error) {
            /* handle error code */
    }
    fd = pev.portev_object;
    if (read(fd, rbuf, STRSIZE)) {
            /* handle error code */
    }
    if (fd-still-accepting-data) {
            /*
             * re-associate the file descriptor with the port.
             * The re-association is required for the
             * re-activation of the data detection.
             * Internals events and user arguments are set to the
             * new (or the same) values delivered here.
             */
            error = port_associate(port, PORT_SOURCE_FD, fd, POLLIN,
                pev.portev_user);
    } else {
            /*
             * If file descriptor is no longer required,
```

EXAMPLE 1 Retrieve data from a pipe file descriptor. *(Continued)*

```
             * - it can remain disabled but still associated with
             *   the port, or
             * - it can be dissociated from the port.
             */
        }
```

EXAMPLE 2 Bind AIO transaction to a specific port.

The following example binds the AIO transaction to a specific port.

```
#include <port.h>

int             port;
port_notify_t   pn;
aiocb_t         aiocb;
aiocb_t         *aiocbp;
void            *mypointer;
int             error;
int             my_errno;
int             my_status;
struct timespec_t timeout;
port_event_t    pev;

port = port_create();
...
/* fill AIO specific part */
aiocb.aio_fildes = fd;
aiocb.aio_nbytes = BUFSIZE;
aiocb.aio_buf = bufp;
aiocb.aio_offset = 0;

/* port specific part */
pn.portnfy_port = port;
pn.portnfy_user = mypointer;
aiocb.aio_sigevent.sigev_notify = SIGEV_PORT;
aiocb.aio_sigevent.sigev_value.sival_ptr = &pn

/*
 * The aio_read() function binds internally the asynchronous I/O
 * transaction with the port delivered in port_notify_t.
 */
error = aio_read(&aiocb);

timeout.tv_sec = 1;     /* user defined */
timeout.tv_nsec = 0;
```

EXAMPLE 2 Bind AIO transaction to a specific port. *(Continued)*

```
/* retrieve a single event */
error = port_get(port, &pev, &timeout);
if (error) {
        /* handle error code */
}

/*
 * pev.portev_object contains a pointer to the aiocb structure
 * delivered in port_notify_t (see aio_read()).
 */
aiocbp = pev.portev_object;

/* check error code and return value in
my_errno = aio_error(aiocbp);
...
my_status = aio_return(aiocbp);
...
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs, system/header
Interface Stability	Committed
MT-Level	Safe

See Also

`rctladm(1M)`, `poll(2)`, `setrctl(2)`, `port_alert(3C)`, `port_create(3C)`, `port_get(3C)`, `port_send(3C)`, `signal.h(3HEAD)`, `attributes(5)`

Name port_create – create a port

Synopsis

```
#include <port.h>

int port_create(void);
```

Description The `port_create()` function establishes a queue that multiplexes events from disjoint sources. Each source has a corresponding object type and source-specific mechanism for associating an object with a port.

source	object type	association mechanism
PORT_SOURCE_AIO	struct aiocb	aio_read(3C), aio_write(3C), lio_listio(3C)
PORT_SOURCE_FD	file descriptor	port_associate(3C)
PORT_SOURCE_MQ	mqd_t	mq_notify(3C)
PORT_SOURCE_TIMER	timer_t	timer_create(3C)
PORT_SOURCE_USER	uintptr_t	port_send(3C)
PORT_SOURCE_ALERT	uintptr_t	port_alert(3C)
PORT_SOURCE_FILE	file_obj_t	port_associate(3C)

`PORT_SOURCE_AIO` events represent the completion of an asynchronous I/O transaction. An asynchronous I/O transaction is associated with a port by specifying `SIGEV_PORT` as its notification mechanism. See `aio_read`(3C), `aio_write`(3C), `lio_listio`(3C), and `aio.h`(3HEAD) for details.

`PORT_SOURCE_FD` events represent a transition in the `poll`(2) status of a given file descriptor. Once an event is delivered, the file descriptor is no longer associated with the port. A file descriptor is associated (or re-associated) with a port using the `port_associate`(3C) function.

`PORT_SOURCE_MQ` events represent a message queue transition from empty to non-empty. A message queue is associated with a port by specifying `SIGEV_PORT` as its notification mechanism. See `mq_notify`(3C) for more information.

`PORT_SOURCE_TIMER` events represent one or more timer expirations for a given timer. A timer is associated with a port by specifying `SIGEV_PORT` as its notification mechanism. See `timer_create`(3C) for more information.

`PORT_SOURCE_USER` events represent user-defined events. These events are generated by `port_send`(3C) or `port_sendn`(3C).

PORT_SOURCE_ALERT events indicate that the port itself is in alert mode. The mode of the port is changed with port_alert(3C).The port_create() function returns a file descriptor that represents a newly created port. The close(2) function destroys the port and frees all allocated resources.

PORT_SOURCE_FILE events represent file/directory status change. Once an event is delivered, the file object associated with the port is no longer active. It has to be reassociated to activate. A file object is associated or reassociated with a port using the port_associate(3C).

The port_get(3C) and port_getn(3C) functions retrieve events from a port. They ignore non retrievable events (non-own or non-shareable events).

As a port is represented by a file descriptor, ports are shared between child and parent processes after fork(). Both can continue to associate sources with the port, both can receive events from the port, and events associated with and/or generated by either process are retrievable in the other. Since some events might not have meaning in both parent and child, care must be taken when using ports after fork().

If a port is exported to other processes, the port is destroyed on last close.

PORT_SOURCE_USER and PORT_SOURCE_ALERT events can be distributed across processes. PORT_SOURCE_FD events can only be shared between processes when child processes inherit opened file decriptors from the parent process. See fork(2). PORT_SOURCE_TIMER and PORT_SOURCE_AIO cannot be shared between processes.

Return Values Upon succesful completion, the port_create() function returns a non-negative value, the port identifier. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The port_create() function will fail if:

EAGAIN The maximum allowable number of ports is currently open in the system. The maximum allowable number of ports is the minimum value of the project.max-port-ids resource control. See setrctl(2) and rctladm(1M) for information on using resource controls.

EMFILE The process has too many open descriptors.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs, system/header
Interface Stability	Committed
MT-Level	Safe

See Also rctladm(1M), close(2), exit(2), fork(2), poll(2), setrctl(2), aio_read(3C), aio_write(3C), aio.h(3HEAD), lio_listio(3C), mq_notify(3C), port_associate(3C), port_get(3C), timer_create(3C), attributes(5)

Name port_get, port_getn – retrieve event information from a port

Synopsis

```
#include <port.h>

int port_get(int port, port_event_t *pe,
     const timespec_t *timeout);

int port_getn(int port, port_event_t list[], uint_t max,
     uint_t *nget, const timespec_t *timeout);
```

Description The `port_get()` and `port_getn()` functions retrieve events from a port. The `port_get()` function retrieves at most a single event. The `port_getn()` function can retrieve multiple events.

The *pe* argument points to an uninitialized `port_event_t` structure that is filled in by the system when the `port_get()` function returns successfully.

The `port_event_t` structure contains the following members:

```
int       portev_events;   /* detected events           */
ushort_t  portev_source;   /* event source              */
uintptr_t portev_object;   /* specific to event source  */
void      *portev_user;    /* user defined cookie       */
```

The `portev_events` and `portev_object` members are specific to the event source. The `portev_events` denotes the delivered events. The `portev_object` refers to the associated object (see `port_create(3C)`). The `portev_source` member specifies the source of the event. The `portev_user` member is a user-specified value.

If the *timeout* pointer is `NULL`, the `port_get()` function blocks until an event is available. To poll for an event without waiting, *timeout* should point to a zeroed `timespec`. A non-zeroed `timespec` specifies the desired time to wait for events. The `port_get()` function returns before the timeout elapses if an event is available, a signal occurs, a port is closed by another thread, or the port is in or enters alert mode. See `port_alert(3C)` for details on alert mode.

The `port_getn()` function can retrieve multiple events from a port. The *list* argument is an array of uninitialized `port_event_t` structures that is filled in by the system when the `port_getn()` function returns succesfully. The *nget* argument points to the desired number of events to be retrieved. The *max* parameter specifies the maximum number of events that can be returned in *list*[]. If *max* is 0, the value pointed to by *nget* is set to the number of events available on the port. The `port_getn()` function returns immediately but no events are retrieved.

The `port_getn()` function block until the desired number of events are available, the timeout elapses, a signal occurs, a port is closed by another thread, or the port is in or enters alert mode.

On return, the value pointed to by *nget* is updated to the actual number of events retrieved in list.

Threads calling the `port_get()` function might starve threads waiting in the `port_getn()` function for more than one event. Similarly, threads calling the `port_getn()` function for *n* events might starve threads waiting in the `port_getn()` function for more than *n* events.

The `port_get()` and the `port_getn()` functions ignore non-shareable events (see `port_create(3C)`) generated by other processes.

Return Values Upon succesful completion, 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error.

Errors The `port_get()` and `port_getn()` functions will fail if:

`EBADF` The *port* identifier is not valid.

`EBADFD` The *port* argument is not an event port file descriptor.

`EFAULT` Event or event list can not be delivered (*list*[] pointer and/or user space reserved to accomodate the list of events is not reasonable), or the *timeout* argument is not reasonable.

`EINTR` A signal was caught during the execution of the function.

`EINVAL` The *timeout* element `tv_sec` is < 0 or the *timeout* element `tv_nsec` is < 0 or > 1000000000.

`ETIME` The time interval expired before the expected number of events have been posted to the port.

The `port_getn()` function will fail if:

`EINVAL` The *list*[] argument is `NULL`, the *nget* argument is `NULL`, or the content of *nget* is > *max* and *max* is > 0.

`EFAULT` The *timeout* argument is not reasonable.

`ETIME` The time interval expired before any events were posted to the port.

Examples **EXAMPLE 1** Send a user event (PORT_SOURCE_USER) to a port and retrieve it with `port_get()`.

The following example sends a user event (`PORT_SOURCE_USER`) to a port and retrieves it with `port_get()`. The `portev_user` and `portev_events` members of the `port_event_t` structure are the same as the corresponding user and events arguments of the `port_send(3C)` function.

```
#include <port.h>

int             myport;
port_event_t    pe;
struct timespec timeout;
int             ret;
void            *user;
uintptr_t       object;
```

EXAMPLE 1 Send a user event (PORT_SOURCE_USER) to a port and retrieve it with `port_get()`. *(Continued)*

```
myport = port_create();
if (myport < 0) {
        /* port creation failed ... */
        ...
        return(...);
}
...
events = 0x01;          /* own event definition(s) */
object = <myobject>;
user = <my_own_value>;
ret = port_send(myport, events, user);
if (ret == -1) {
        /* error detected ... */
        ...
        close(myport);
        return (...);
}

/*
 * The following code could also be executed in another thread or
 * process.
 */
timeout.tv_sec = 1;     /* user defined */
timeout.tv_nsec = 0;
ret = port_get(myport, &pe, &timeout);
if (ret == -1) {
        /*
         * error detected :
         * - EINTR or ETIME : log error code and try again ...
         * - Other kind of errors : may have to close the port ...
         */
        return(...);
}

/*
 * After port_get() returns successfully, the port_event_t
 * structure will be filled with:
 * pe.portev_source =   PORT_SOURCE_USER
 * pe.portev_events = 0x01
 * pe.portev_object = <myobject>
 * pe.portev_user = <my_own_value>
 */
...
```

EXAMPLE 1 Send a user event (PORT_SOURCE_USER) to a port and retrieve it with `port_get()`. *(Continued)*

```
close(myport);
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs, system/header
Interface Stability	Committed
MT-Level	Safe

See Also

`port_alert(3C)`, `port_associate(3C)`, `port_create(3C)`, `port_send(3C)`, `attributes(5)`

Name port_send, port_sendn – send a user-defined event to a port or list of ports

Synopsis

```
#include <port.h>

int port_send(int port, int events, void *user);

int port_sendn(int ports[], int errors[], uint_t nent,
     int events, void *user);
```

Description The `port_send()` function submits a user-defined event to a specified port. The *port* argument is a file descriptor that represents a port. The sent event has its `portev_events` member set to the value specified in the *events* parameter and its `portev_user` member set to the value specified in the *user* parameter. The `portev_object` member of an event sent with `port_send()` is unspecified.

The `port_sendn()` function submits a user-defined event to multiple ports. The *ports* argument is an array of file descriptors that represents ports (see `port_create(3C)`). The *nent* argument specifies the number of file descriptors in the *ports*[] array. An event is submitted to each specified port. Each event has its `portev_events` member set to the value specified in the *events* parameter and its `portev_user` member set to the value specified in the *user* parameter. The `portev_object` member of *events* sent with `port_sendn()` is unspecified.

A port that is in alert mode can be sent an event, but that event will not be retrievable until the port has resumed normal operation. See `port_alert(3C)`.

Return Values Upon successful completion, the `port_send()` function returns 0. Otherwise, it returns −1 and sets `errno` to indicate the error.

The `port_sendn()` function returns the number of successfully submitted events. A non-negative return value less than the *nent* argument indicates that at least one error occurred. In this case, each element of the *errors*[] array is filled in. An element of the *errors*[] array is set to 0 if the event was successfully sent to the corresponding port in the *ports*[] array, or is set to indicate the error if the event was not successfully sent. If an error occurs, the `port_sendn()` function returns −1 and sets `errno` to indicate the error.

Errors The `port_send()` and `port_sendn()` functions will fail if:

`EAGAIN` The maximum number of events per port is exceeded. The maximum allowable number of events per port is the minimum value of the `process.max-port-events` resource control at the time `port_create(3C)` was used to create the port.

`EBADF` The port file descriptor is not valid.

`EBADFD` The *port* argument is not an event port file descriptor.

`ENOMEM` There is not enough memory available to satisfy the request.

The `port_sendn()` function will fail if:

`EFAULT` The *ports*[] pointer or *errors*[] pointer is not reasonable.

`EINVAL` The value of the *nent* argument is 0.

Examples

EXAMPLE 1 Use `port_send()` to send a user event (PORT_SOURCE_USER) to a port.

The following example uses `port_send()` to send a user event (`PORT_SOURCE_USER`) to a port and `port_get()` to retrieve it. The `portev_user` and `portev_events` members of the `port_event_t` structure are the same as the corresponding user and events arguments of the `port_send()` function.

```
#include <port.h>

int             myport;
port_event_t    pe;
struct timespec timeout;
int             ret;
void            *user;

myport = port_create();
if (myport) {
        /* port creation failed ... */
        ...
        return(...);
}
...
events = 0x01;          /* own event definition(s) */
user = <my_own_value>;
ret = port_send(myport, events, user);
if (ret == -1) {
        /* error detected ... */
        ...
        close(myport);
        return (...);
}

/*
 * The following code could also be executed from another thread or
 * process.
 */
timeout.tv_sec = 1;     /* user defined */
timeout.tv_nsec = 0;
ret = port_get(myport, &pe, &timeout);
if (ret == -1) {
        /*
         * error detected :
         * - EINTR or ETIME : log error code and try again ...
         * - Other kind of errors : may have to close the port ...
         */
        return(...);
```

EXAMPLE 1 Use `port_send()` to send a user event (PORT_SOURCE_USER) to a port. *(Continued)*

```
}

/*
 * After port_get() returns successfully, the port_event_t
 * structure will be filled with:
 * pe.portev_source =   PORT_SOURCE_USER
 * pe.portev_events = 0x01
 * pe.portev_object = unspecified
 * pe.portev_user = <my_own_value>
 */
...
close(myport);
```

Usage See `setrctl(2)` and `rctladm(1M)` for information on using resource controls.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcs, system/header
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also `rctladm(1M)`, `setrctl(2)`, `port_alert(3C)`, `port_associate(3C)`, `port_create(3C)`, `port_get(3C)`, `attributes(5)`

Name posix_fadvise – file advisory information

Synopsis

```
#include <fcntl.h>

int posix_fadvise(int fd, off_t offset, off_t len, int advice);
```

Description The `posix_fadvise()` function advises the system on the expected behavior of the application with respect to the data in the file associated with the open file descriptor, *fd*, starting at *offset* and continuing for *len* bytes. The specified range need not currently exist in the file. If *len* is zero, all data following offset is specified. The system may use this information to optimize handling of the specified data. The `posix_fadvise()` function has no effect on the semantics of other operations on the specified data, although it may affect the performance of other operations.

The advice to be applied to the data is specified by the *advice* parameter and may be one of the following values:

`POSIX_FADV_NORMAL`	Specifies that the application has no advice to give on its behavior with respect to the specified data. It is the default characteristic if no advice is given for an open file.
`POSIX_FADV_SEQUENTIAL`	Specifies that the application expects to access the specified data sequentially from lower offsets to higher offsets.
`POSIX_FADV_RANDOM`	Specifies that the application expects to access the specified data in a random order.
`POSIX_FADV_WILLNEED`	Specifies that the application expects to access the specified data in the near future.
`POSIX_FADV_DONTNEED`	Specifies that the application expects that it will not access the specified data in the near future.
`POSIX_FADV_NOREUSE`	Specifies that the application expects to access the specified data once and then not reuse it thereafter.

These values are defined in `<fcntl.h>`

Return Values Upon successful completion, `posix_fadvise()` returns zero. Otherwise, an error number is returned to indicate the error.

Errors The `posix_fadvise()` function will fail if:

`EBADF`	The *fd* argument is not a valid file descriptor.
`EINVAL`	The value of *advice* is invalid, or the value of *len* is less than zero.
`ESPIPE`	The *fd* argument is associated with a pipe or FIFO.

Usage The `posix_fadvise()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `posix_madvise(3C)`, `attributes(5)`, `standards(5)`

Name posix_fallocate – file space control

Synopsis

```
#include <fcntl.h>

int posix_fallocate(int fd, off_t offset, off_t len);
```

Description The `posix_fallocate()` function ensures that any required storage for regular file data starting at *offset* and continuing for *len* bytes is allocated on the file system storage media. If `posix_fallocate()` returns successfully, subsequent writes to the specified file data will not fail due to the lack of free space on the file system storage media.

If the *offset+len* is beyond the current file size, then `posix_fallocate()` adjusts the file size to *offset+len*. Otherwise, the file size is not changed.

Space allocated with `posix_fallocate()` is freed by a successful call to `creat(2)` or `open(2)` that truncates the size of the file. Space allocated with `posix_fallocate()` may be freed by a successful call to `ftruncate(3C)` that reduces the file size to a size smaller than *offset+len*.

Return Values Upon successful completion, `posix_fallocate()` returns zero. Otherwise, an error number is returned to indicate the error.

Errors The `posix_fallocate()` function will fail if:

`EBADF` The *fd* argument is not a valid file descriptor or references a file that was opened without write permission.

`EFBIG` The value of *offset+len* is greater than the maximum file size.

`EINTR` A signal was caught during execution.

`EINVAL` The *len* argument is less than or equal to zero, or the *offset* argument is less than zero, or the underlying file system does not support this operation.

`EIO` An I/O error occurred while reading from or writing to a file system.

`ENODEV` The *fd* argument does not refer to a regular file.

`ENOSPC` There is insufficient free space remaining on the file system storage media.

`ESPIPE` The *fd* argument is associated with a pipe or FIFO.

Usage The `posix_fallocate()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also creat(2), open(2), unlink(2), ftruncate(3C), attributes(5), standards(5)

Name posix_madvise – memory advisory information

Synopsis

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

int posix_madvise(void *addr, size_t len, int advice);
```

Description The `posix_madvise()` function advises the system on the expected behavior of the application with respect to the data in the memory starting at address *addr*, and continuing for *len* bytes. The system may use this information to optimize handling of the specified data. The `posix_madvise()` function has no effect on the semantics of access to memory in the specified range, although it may affect the performance of access.

The advice to be applied to the memory range is specified by the *advice* parameter and may be one of the following values:

`POSIX_MADV_NORMAL`	Specifies that the application has no advice to give on its behavior with respect to the specified range. It is the default characteristic if no advice is given for a range of memory.
`POSIX_MADV_SEQUENTIAL`	Specifies that the application expects to access the specified range sequentially from lower addresses to higher addresses.
`POSIX_MADV_RANDOM`	Specifies that the application expects to access the specified range in a random order.
`POSIX_MADV_WILLNEED`	Specifies that the application expects to access the specified range in the near future.
`POSIX_MADV_DONTNEED`	Specifies that the application expects that it will not access the specified range in the near future.

These values are defined in `<sys/mman.h>`

Return Values Upon successful completion, `posix_madvise()` returns zero. Otherwise, an error number is returned to indicate the error.

Errors The `posix_madvise()` function will fail if:

`EINVAL`	The value of *advice* is invalid.
`ENOMEM`	Addresses in the range starting at *addr* and continuing for *len* bytes are partly or completely outside the range allowed for the address space of the calling process.

The `posix_madvise()` function may fail if:

`EINVAL`	The value of *len* is zero.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also mmap(2), madvise(3C), posix_madvise(3C), attributes(5), standards(5)

Name posix_memalign – aligned memory allocation

Synopsis

```
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);
```

Description The `posix_memalign()` function allocates size bytes aligned on a boundary specified by alignment, and returns a pointer to the allocated memory in *memptr*. The value of alignment must be a power of two multiple of `sizeof(void *)`.

Upon successful completion, the value pointed to by *memptr* will be a multiple of alignment.

If the size of the space requested is 0, the value returned in *memptr* will be a null pointer.

The `free(3C)` function will deallocate memory that has previously been allocated by `posix_memalign()`.

Return Values Upon successful completion, `posix_memalign()` returns zero. Otherwise, an error number is returned to indicate the error.

Errors The `posix_memalign()` function will fail if:

`EINVAL` The value of the alignment parameter is not a power of two multiple of `sizeof(void *)`.

`ENOMEM` There is insufficient memory available with the requested alignment.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `free(3C)`, `malloc(3C)`, `memalign(3C)`, `attributes(5)`, `standards(5)`

Name posix_openpt – open a pseudo terminal device

Synopsis

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

int posix_openpt(int oflag);
```

Description The `posix_openpt()` function establishes a connection between a master device for a pseudo-terminal and a file descriptor. The file descriptor is used by other I/O functions that refer to that pseudo-terminal.

The file status flags and file access modes of the open file description are set according to the value of *oflag*.

Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in `<fcntl.h>`.

`O_RDWR` Open for reading and writing.

`O_NOCTTY` If set, `posix_openpt()` does not cause the terminal device to become the controlling terminal for the process.

The behavior of other values for the *oflag* argument is unspecified.

Return Values Upon successful completion, the `posix_openpt()` function opens a master pseudo-terminal device and returns a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors The `posix_openpt()` function will fail if:

`EMFILE` {`OPEN_MAX`} file descriptors are currently open in the calling process.

`ENFILE` The maximum allowable number of files is currently open in the system.

The `posix_openpt()` function may fail if:

`EINVAL` The value of *oflag* is not valid.

`EAGAIN` Out of pseudo-terminal resources.

`ENOSR` Out of STREAMS resources.

Examples **EXAMPLE 1** Open a pseudo-terminal.

The following example opens a pseudo-terminal and returns the name of the slave device and a file descriptor.

```
#include fcntl.h>
#include stdio.h>

int masterfd, slavefd;
char *slavedevice;
```

EXAMPLE 1 Open a pseudo-terminal. *(Continued)*

```
masterfd = posix_openpt(O_RDWR|O_NOCTTY);

if (masterfd == -1
      || grantpt (masterfd) == -1
      || unlockpt (masterfd) == -1
      || (slavedevice = ptsname (masterfd)) == NULL)
      return -1;

printf("slave device is: %s\n", slavedevice);

slavefd = open(slave, O_RDWR|O_NOCTTY);
if (slavefd < 0)
      return -1;
```

Usage This function provides a method for portably obtaining a file descriptor of a master terminal device for a pseudo-terminal. The `grantpt(3C)` and `ptsname(3C)` functions can be used to manipulate mode and ownership permissions and to obtain the name of the slave device, respectively.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `open(2)`, `grantpt(3C)`, `ptsname(3C)`, `unlockpt(3C)`, `attributes(5)`, `standards(5)`

Name posix_spawn, posix_spawnp – spawn a process

Synopsis

```
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
     const posix_spawn_file_actions_t *file_actions,
     const posix_spawnattr_t *restrict attrp,
     char *const argv[restrict], char *const envp[restrict]);

int posix_spawnp(pid_t *restrict pid, const char *restrict file,
     const posix_spawn_file_actions_t *file_actions,
     const posix_spawnattr_t *restrict attrp,
     char *const argv[restrict], char *const envp[restrict]);
```

Description The `posix_spawn()` and `posix_spawnp()` functions create a new process (child process) from the specified process image. The new process image is constructed from a regular executable file called the new process image file.

When a C program is executed as the result of this call, it is entered as a C language function call as follows:

```
int main(int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable

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

is initialized as a pointer to an array of character pointers to the environment strings.

The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array is a null pointer and is not counted in *argc*. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a filename that is associated with the process image being started by the `posix_spawn()` or `posix_spawnp()` function.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a null pointer.

The number of bytes available for the child process's combined argument and environment lists is {`ARG_MAX`}, counting all character pointers, the strings they point to, the trailing null bytes in the strings, and the list-terminating null pointers. There is no additional system overhead included in this total.

The *path* argument to `posix_spawn()` is a pathname that identifies the new process image file to execute.

The *file* parameter to `posix_spawnp()` is used to construct a pathname that identifies the new process image file. If the file parameter contains a slash character, the file parameter is used as

the pathname for the new process image file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable PATH. If this environment variable is not defined, the results of the search are implementation-defined.

If *file_actions* is a null pointer, then file descriptors open in the calling process remain open in the child process, except for those whose close-on-exec flag FD_CLOEXEC is set (see fcntl(2)). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks (see fcntl(2)), remain unchanged.

If *file_actions* is not NULL, then the file descriptors open in the child process are those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions are:

1. The set of open file descriptors for the child process are initially the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks (see fcntl(2)), remain unchanged.
2. The signal mask, signal default or ignore actions, and the effective user and group IDs for the child process are changed as specified in the attributes object referenced by *attrp*.
3. The file actions specified by the spawn file actions object are performed in the order in which they were added to the spawn file actions object.
4. Any file descriptor that has its FD_CLOEXEC flag set (see fcntl(2)) is closed.

The posix_spawnattr_t spawn attributes object type is defined in <spawn.h>. It contains at least the attributes defined below.

If the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is non-zero, then the child's process group is as specified in the *spawn-pgroup* attribute of the object referenced by *attrp*.

As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is set to zero, then the child will be in a new process group with a process group ID equal to its process ID.

If the POSIX_SPAWN_SETPGROUP flag is not set in the *spawn-flags* attribute of the object referenced by *attrp*, the new child process inherits the parent's process group.

If the POSIX_SPAWN_SETSCHEDPARAM flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image initially has the scheduling policy of the calling process with the scheduling parameters specified in the *spawn-schedparam* attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSCHEDULER flag is set in *spawn-flags* attribute of the object referenced by *attrp* (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag), the new process image initially has the scheduling policy specified in the *spawn-schedpolicy* attribute of the object referenced by *attrp* and the scheduling parameters specified in the *spawn-schedparam* attribute of the same object.

The POSIX_SPAWN_RESETIDS flag in the *spawn-flags* attribute of the object referenced by *attrp* governs the effective user ID of the child process. If this flag is not set, the child process inherits the parent process's effective user ID. If this flag is set, the child process's effective user ID is reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process becomes that file's owner ID before the new process image begins execution. If this flag is set, the child process's effective user ID is reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process becomes that file's owner ID before the new process image begins execution.

The POSIX_SPAWN_RESETIDS flag in the *spawn-flags* attribute of the object referenced by *attrp* also governs the effective group ID of the child process. If this flag is not set, the child process inherits the parent process's effective group ID. If this flag is set, the child process's effective group ID is reset to the parent's real group ID. In either case, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the child process becomes that file's group ID before the new process image begins execution.

If the POSIX_SPAWN_SETSIGMASK flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, the child process initially has the signal mask specified in the *spawn-sigmask* attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSIGDEF flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, the signals specified in the *spawn-sigdefault* attribute of the same object is set to their default actions in the child process.

If the POSIX_SPAWN_SETSIGIGN_NP flag is set in the spawn-flags attribute of the object referenced by *attrp*, the signals specified in the *spawn-sigignore* attribute of the same object are set to be ignored in the child process.

If both POSIX_SPAWN_SETSIGDEF and POSIX_SPAWN_SETSIGIGN_NP flags are set in the spawn-flags attribute of the object referenced by *attrp*, the actions for POSIX_SPAWN_SETSIGDEF take precedence over the actions for POSIX_SPAWN_SETSIGIGN_NP.

If the POSIX_SPAWN_NOSIGCHLD_NP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, no SIGCHLD signal will be posted to the parent process when the child process terminates, regardless of the disposition of the SIGCHLD signal in the parent. SIGCHLD signals are still possible for job control stop and continue actions if the parent has requested them.

If the POSIX_SPAWN_WAITPID_NP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, no wait-for-multiple-pids operation by the parent, as in wait(), waitid(P_ALL), or waitid(P_PGID), will succeed in reaping the child, and the child will not be reaped automatically due the disposition of the SIGCHLD signal being set to be ignored in the parent. Only a specific wait for the child, as in waitid(P_PID, pid), is allowed and it is required, else when the child exits it will remain a zombie until the parent exits.

If the POSIX_SPAWN_NOEXECERR_NP flag is set in the spawn-flags attribute of the object referenced by *attrp*, and if the specified process image file cannot be executed, then the posix_spawn() and posix_spawnp() functions do not fail with one of the exec(2) error codes, as is normal, but rather return successfully having created a child process that exits immediately with exit status 127. This flag permits system(3C) and popen(3C) to be implemented with posix_spawn() and still conform strictly to their POSIX specifications.

Signals set to be caught or set to the default action in the calling process are set to the default action in the child process, unless the POSIX_SPAWN_SETSIGIGN_NP flag is set in the spawn-flags attribute of the object referenced by *attrp* and the signals are specified in the *spawn-sigignore* attribute of the same object.

Except for SIGCHLD, signals set to be ignored by the calling process image are set to be ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the *spawn-flags* attribute of the object referenced by *attrp* and the signals being indicated in the *spawn-sigdefault* attribute of the object referenced by *attrp*.

If the SIGCHLD signal is set to be ignored by the calling process, it is unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the *spawn-flags* attribute of the object referenced by *attrp* and the SIGCHLD signal being indicated in the *spawn-sigdefault* attribute of the object referenced by *attrp*.

If the value of the *attrp* pointer is NULL, then the default values are used.

All process attributes, other than those influenced by the attributes set in the object referenced by *attrp* as specified above or by the file descriptor manipulations specified in *file_actions* appear in the new process image as though fork() had been called to create a child process and then a member of the exec family of functions had been called by the child process to execute the new process image.

The fork handlers are not run when posix_spawn() or posix_spawnp() is called.

Return Values Upon successful completion, posix_spawn() and posix_spawnp() return the process ID of the child process to the parent process in the variable pointed to by a non-null *pid* argument, and return zero as the function return value. Otherwise, no child process is created, the value stored into the variable pointed to by a non-null *pid* is unspecified, and an error number is returned as the function return value to indicate the error. If the *pid* argument is a null pointer, the process ID of the child is not returned to the caller.

Errors The posix_spawn() and posix_spawnp() functions will fail if:

EINVAL The value specified by *file_actions* or *attrp* is invalid.

If posix_spawn() or posix_spawnp() fails for any of the reasons that would cause fork() or one of the exec family of functions to fail, an error value is returned as described by fork(2) and exec(2), respectively

If POSIX_SPAWN_SETPGROUP is set in the *spawn-flags* attribute of the object referenced by *attrp*, and posix_spawn() or posix_spawnp() fails while changing the child's process group, an error value is returned as described by setpgid(2).

If POSIX_SPAWN_SETSCHEDPARAM is set and POSIX_SPAWN_SETSCHEDULER is not set in the *spawn-flags* attribute of the object referenced by *attrp*, then if posix_spawn() or posix_spawnp() fails for any of the reasons that would cause sched_setparam() to fail, an error value is returned as described by sched_setparam(3C).

If POSIX_SPAWN_SETSCHEDULER is set in the *spawn-flags* attribute of the object referenced by *attrp*, and if posix_spawn() or posix_spawnp() fails for any of the reasons that would cause sched_setscheduler() to fail, an error value is returned as described by sched_setscheduler(3C).

If the *file_actions* argument is not NULL and specifies any close(), dup2(), or open() actions to be performed, and if posix_spawn() or posix_spawnp() fails for any of the reasons that would cause close(), dup2(), or open() to fail, an error value is returned as described by close(2), dup2(3C), or open(2), respectively. An open file action might, by itself, result in any of the errors described by close() or dup2(), in addition to those described by open().

If a close(2) operation is specified to be performed for a file descriptor that is not open at the time of the call to posix_spawn() or posix_spawnp(), the action does not cause posix_spawn() or posix_spawnp() to fail.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also

alarm(2), chmod(2), close(2), dup(2), exec(2), exit(2), fcntl(2), fork(2), kill(2), open(2), setpgid(2), setuid(2), stat(2), times(2), dup2(3C), popen(3C), posix_spawn_file_actions_addclose(3C), posix_spawn_file_actions_adddup2(3C), posix_spawn_file_actions_addopen(3C), posix_spawn_file_actions_destroy(3C), posix_spawn_file_actions_init(3C), posix_spawnattr_destroy(3C), posix_spawnattr_getflags(3C), posix_spawnattr_getpgroup(3C), posix_spawnattr_getschedparam(3C), posix_spawnattr_getschedpolicy(3C), posix_spawnattr_getsigdefault(3C), posix_spawnattr_getsigignore_np(3C), posix_spawnattr_getsigmask(3C), posix_spawnattr_init(3C), posix_spawnattr_setflags(3C), posix_spawnattr_setpgroup(3C), posix_spawnattr_setschedparam(3C), posix_spawnattr_setschedpolicy(3C), posix_spawnattr_setsigdefault(3C), posix_spawnattr_setsigignore_np(3C),

posix_spawnattr_setsigmask(3C), sched_setparam(3C), sched_setscheduler(3C), system(3C), wait(3C), attributes(5), standards(5)

Notes The SUSv3 POSIX standard (The Open Group Base Specifications Issue 6, IEEE Std 1003.1-2001) permits the posix_spawn() and posix_spawnp() functions to return successfully before some of the above-described errors are detected, allowing the child process to fail instead:

```
... if the error occurs after the calling process
successfully returns, the child process exits with
exit status 127.
```

With the one exception of when the POSIX_SPAWN_NOEXECERR_NP flag is passed in the attributes structure, this behavior is not present in the Solaris implementation. Any error that occurs before the new process image is successfully constructed causes the posix_spawn() and posix_spawnp() functions to return the corresponding non-zero error value without creating a child process.

The POSIX_SPAWN_NOSIGCHLD_NP, POSIX_SPAWN_WAITPID_NP, POSIX_SPAWN_NOEXECERR_NP, and POSIX_SPAWN_SETSIGIGN_NP flags and the posix_spawnattr_getsigignore_np() and posix_spawnattr_setsigignore_np() functions are non-portable Solaris extensions to the posix_spawn() and posix_spawnp() interfaces.

Name posix_spawnattr_destroy, posix_spawnattr_init – destroy and initialize spawn attributes object

Synopsis

```
#include <spawn.h>

int posix_spawnattr_destroy(posix_spawnattr_t *attr);

int posix_spawnattr_init(posix_spawnattr_t *attr);
```

Description The `posix_spawnattr_destroy()` function destroys a spawn attributes object. A destroyed *attr* attributes object can be reinitialized using `posix_spawnattr_init()`. The results of otherwise referencing the object after it has been destroyed are undefined. An implementation can cause `posix_spawnattr_destroy()` to set the object referenced by *attr* to an invalid value.

The `posix_spawnattr_init()` function initializes a spawn attributes object *attr* with the default value for all of the individual attributes used by the implementation. Results are undefined if `posix_spawnattr_init()` is called specifying an already initialized *attr* attributes object.

A spawn attributes object is of type `posix_spawnattr_t` (defined in `<spawn.h>`) and is used to specify the inheritance of process attributes across a spawn operation.

No attributes other than those defined by IEEE Std 1003.1-200x are provided.

The resulting spawn attributes object (possibly modified by setting individual attribute values), is used to modify the behavior of `posix_spawn(3C)` or `posix_spawnp(3C)`. After a spawn attributes object has been used to spawn a process by a call to `posix_spawn()` or `posix_spawnp()`, any function affecting the attributes object (including destruction) will not affect any process that has been spawned in this way.

Return Values Upon successful completion, `posix_spawnattr_destroy()` and `posix_spawnattr_init()` return 0. Otherwise, an error number is returned to indicate the error.

Errors The `posix_spawnattr_init()` function will fail if:

`ENOMEM` Insufficient memory exists to initialize the spawn attributes object.

The `posix_spawnattr_destroy()` function may fail if:

`EINVAL` The value specified by *attr* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also posix_spawn(3C), posix_spawnattr_getflags(3C), posix_spawnattr_getpgroup(3C), posix_spawnattr_getschedparam(3C), posix_spawnattr_getschedpolicy(3C), posix_spawnattr_getsigdefault(3C), posix_spawnattr_getsigmask(3C), attributes(5), standards(5)

Name posix_spawnattr_getflags, posix_spawnattr_setflags – get and set spawn-flags attribute of spawn attributes object

Synopsis

```
#include <spawn.h>

int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
     short *restrict flags);

int posix_spawnattr_setflags(posix_spawnattr_t * attr, short flags);
```

Description The `posix_spawnattr_getflags()` function obtains the value of the *spawn-flags* attribute from the attributes object referenced by *attr*.

The `posix_spawnattr_setflags()` function sets the *spawn-flags* attribute in an initialized attributes object referenced by *attr*.

The *spawn-flags* attribute is used to indicate which process attributes are to be changed in the new process image when invoking `posix_spawn(3C)` or `posix_spawnp(3C)`. It is the bitwise inclusive-OR of zero or more of the following flags:

```
POSIX_SPAWN_RESETIDS
POSIX_SPAWN_SETPGROUP
POSIX_SPAWN_SETSIGDEF
POSIX_SPAWN_SETSIGMASK
POSIX_SPAWN_SETSCHEDPARAM
POSIX_SPAWN_SETSCHEDULER
POSIX_SPAWN_NOSIGCHLD_NP
POSIX_SPAWN_WAITPID_NP
POSIX_SPAWN_NOEXECERR_NP
```

These flags are defined in `<spawn.h>`. The default value of this attribute is as if no flags were set.

Return Values Upon successful completion, `posix_spawnattr_getflags()` returns 0 and stores the value of the spawn-flags attribute of attr into the object referenced by the flags parameter. Otherwise, an error number is returned to indicate the error.

Upon successful completion, `posix_spawnattr_setflags()` returns 0. Otherwise, an error number is returned to indicate the error.

Errors These functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `posix_spawnattr_setflags()` function may fail if:

`EINVAL` The value of the attribute being set is not valid.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also posix_spawn(3C), posix_spawnattr_destroy(3C), posix_spawnattr_getpgroup(3C), posix_spawnattr_getschedparam(3C), posix_spawnattr_getschedpolicy(3C), posix_spawnattr_getsigdefault(3C), posix_spawnattr_getsigmask(3C), attributes(5), standards(5)

Name posix_spawnattr_getpgroup, posix_spawnattr_setpgroup – get and set spawn-pgroup attribute of spawn attributes object

Synopsis

```
#include <spawn.h>

int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
     pid_t *restrict pgroup);

int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

Description The `posix_spawnattr_getpgroup()` function obtains the value of the *spawn-pgroup* attribute from the attributes object referenced by *attr*.

The `posix_spawnattr_setpgroup()` function sets the *spawn-pgroup* attribute in an initialized attributes object referenced by *attr*.

The *spawn-pgroup* attribute represents the process group to be joined by the new process image in a spawn operation (if `POSIX_SPAWN_SETPGROUP` is set in the *spawn-flags* attribute). The default value of this attribute is zero.

Return Values Upon successful completion, `posix_spawnattr_getpgroup()` returns 0 and stores the value of the *spawn-pgroup* attribute of *attr* into the object referenced by the *pgroup* parameter. Otherwise, an error number is returned to indicate the error.

Upon successful completion, `posix_spawnattr_setpgroup()` returns 0. Otherwise, an error number is returned to indicate the error.

Errors These functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `posix_spawnattr_setpgroup()` function may fail if:

`EINVAL` The value of the attribute being set is not valid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `posix_spawn(3C)`, `posix_spawnattr_getpgroup(3C)`, `posix_spawnattr_getpgroup(3C)`, `posix_spawnattr_getschedparam(3C)`, `posix_spawnattr_getschedpolicy(3C)`, `posix_spawnattr_getsigdefault(3C)`, `posix_spawnattr_getsigmask(3C)`, `attributes(5)`, `standards(5)`

Name posix_spawnattr_getschedparam, posix_spawnattr_setschedparam – get and set spawn-schedparam attribute of spawn attributes object

Synopsis

```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedparam(const posix_spawnattr_t *restrict attr,
     struct sched_param *restrict schedparam);

int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
     const struct sched_param *restrict schedparam);
```

Description The `posix_spawnattr_getschedparam()` function obtains the value of the *spawn-schedparam* attribute from the attributes object referenced by *attr*.

The `posix_spawnattr_setschedparam()` function sets the *spawn-schedparam* attribute in an initialized attributes object referenced by *attr*.

The *spawn-schedparam* attribute represents the scheduling parameters to be assigned to the new process image in a spawn operation (if `POSIX_SPAWN_SETSCHEDULER` or `POSIX_SPAWN_SETSCHEDPARAM` is set in the *spawn-flags* attribute). The default value of this attribute is unspecified.

Return Values Upon successful completion, `posix_spawnattr_getschedparam()` returns 0 and stores the value of the *spawn-schedparam* attribute of *attr* into the object referenced by the *schedparam* parameter. Otherwise, an error number is returned to indicate the error.

Upon successful completion, `posix_spawnattr_setschedparam()` returns 0. Otherwise, an error number is returned to indicate the error.

Errors These functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `posix_spawnattr_setschedparam()` function may fail if:

`EINVAL` The value of the attribute being set is not valid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also posix_spawn(3C), posix_spawnattr_destroy(3C), posix_spawnattr_getflags(3C), posix_spawnattr_getpgroup(3C), posix_spawnattr_getschedpolicy(3C), posix_spawnattr_getsigdefault(3C), posix_spawnattr_getsigmask(3C), attributes(5), standards(5)

Name posix_spawnattr_getschedpolicy, posix_spawnattr_setschedpolicy – get and set spawn-schedpolicy attribute of spawn attributes object

Synopsis

```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedpolicy(
     const posix_spawnattr_t *restrict attr,
     int *restrict schedpolicy);

int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
     int schedpolicy);
```

Description The `posix_spawnattr_getschedpolicy()` function obtains the value of the *spawn-schedpolicy* attribute from the attributes object referenced by *attr*.

The `posix_spawnattr_setschedpolicy()` function sets the *spawn-schedpolicy* attribute in an initialized attributes object referenced by *attr*.

The *spawn-schedpolicy* attribute represents the scheduling policy to be assigned to the new process image in a spawn operation (if `POSIX_SPAWN_SETSCHEDULER` is set in the *spawn-flags* attribute). The default value of this attribute is unspecified.

Return Values Upon successful completion, `posix_spawnattr_getschedpolicy()` returns 0 and stores the value of the *spawn-schedpolicy* attribute of *attr* into the object referenced by the *schedpolicy* parameter. Otherwise, an error number is returned to indicate the error.

Upon successful completion, `posix_spawnattr_setschedpolicy()` returns 0. Otherwise, an error number is returned to indicate the error.

Errors These functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `posix_spawnattr_setschedpolicy()` function may fail if:

`EINVAL` The value of the attribute being set is not valid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also posix_spawn(3C), posix_spawnattr_destroy(3C), posix_spawnattr_getflags(3C), posix_spawnattr_getpgroup(3C), posix_spawnattr_getschedparam(3C), posix_spawnattr_getsigdefault(3C), posix_spawnattr_getsigmask(3C), attributes(5), standards(5)

Name posix_spawnattr_getsigdefault, posix_spawnattr_setsigdefault – get and set spawn-sigdefault attribute of spawn attributes object

Synopsis

```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigdefault(const posix_spawnattr_t *restrict attr,
     sigset_t *restrict sigdefault);

int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
     const sigset_t *restrict sigdefault);
```

Description The `posix_spawnattr_getsigdefault()` function obtains the value of the *spawn-sigdefault* attribute from the attributes object referenced by *attr*.

The `posix_spawnattr_setsigdefault()` function sets the *spawn-sigdefault* attribute in an initialized attributes object referenced by *attr*.

The *spawn-sigdefault* attribute represents the set of signals to be forced to default signal handling in the new process image (if `POSIX_SPAWN_SETSIGDEF` is set in the *spawn-flags* attribute) by a spawn operation. The default value of this attribute is an empty signal set.

Return Values Upon successful completion, `posix_spawnattr_getsigdefault()` returns 0 and stores the value of the *spawn-sigdefault* attribute of *attr* into the object referenced by the *sigdefault* parameter. Otherwise, an error number is returned to indicate the error.

Upon successful completion, `posix_spawnattr_setsigdefault()` returns 0. Otherwise, an error number is returned to indicate the error.

Errors These functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `posix_spawnattr_setsigdefault()` function may fail if:

`EINVAL` The value of the attribute being set is not valid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also posix_spawn(3C), posix_spawnattr_destroy(3C), posix_spawnattr_getflags(3C), posix_spawnattr_getpgroup(3C), posix_spawnattr_getschedparam(3C), posix_spawnattr_getschedpolicy(3C), posix_spawnattr_getsigmask(3C), attributes(5), standards(5)

Name posix_spawnattr_getsigignore_np, posix_spawnattr_setsigignore_np – get and set spawn-sigignore attribute of spawn attributes object

Synopsis

```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigignore_np(
     const posix_spawnattr_t *restrict attr,
     sigset_t *restrict sigignore);

int posix_spawnattr_setsigignore_np(
     posix_spawnattr_t *restrict attr,
     const sigset_t *restrict sigignore);
```

Description The `posix_spawnattr_getsigignore_np()` function obtains the value of the *spawn-sigignore* attribute from the attributes object referenced by *attr*.

The `posix_spawnattr_setsigignore_np()` function sets the *spawn-sigignore* attribute in an initialized attributes object referenced by *attr*.

The *spawn-sigignore* attribute represents the set of signals to be forced to be ignored in the new process image (if `POSIX_SPAWN_SETSIGIGN_NP` is set in the spawn-flags attribute) by a spawn operation. The default value of this attribute is an empty signal set.

Return Values Upon successful completion, `posix_spawnattr_getsigignore_np()` returns 0 and stores the value of the *spawn-sigignore* attribute of *attr* into the object referenced by the *sigignore* parameter. Otherwise, an error value is returned to indicate the error.

Upon successful completion, `posix_spawnattr_setsigignore_np()` returns 0. Otherwise, an error value is returned to indicate the error.

Errors These functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `posix_spawnattr_setsigignore_np()` function may fail if:

`EINVAL` The value of the attribute being set is not valid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also posix_spawn(3C), posix_spawnattr_destroy(3C), posix_spawnattr_getflags(3C), posix_spawnattr_getpgroup(3C), posix_spawnattr_getschedparam(3C), posix_spawnattr_getschedpolicy(3C), posix_spawnattr_setsigdefault(3C), posix_spawnattr_setsigmask(3C), attributes(5)

Notes The POSIX_SPAWN_SETSIGIGN_NP flag and the posix_spawnattr_getsigignore_np() and posix_spawnattr_setsigignore_np() functions are non-portable Solaris extensions to the posix_spawn(3C) and posix_spawnp() interfaces.

Name posix_spawnattr_getsigmask, posix_spawnattr_setsigmask – get and set spawn-sigmask attribute of spawn attributes object

Synopsis

```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr,
     sigset_t *restrict sigmask);

int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
     const sigset_t *restrict sigmask);
```

Description The `posix_spawnattr_getsigmask()` function obtains the value of the *spawn-sigmask* attribute from the attributes object referenced by *attr*.

The `posix_spawnattr_setsigmask()` function sets the *spawn-sigmask* attribute in an initialized attributes object referenced by *attr*.

The *spawn-sigmask* attribute represents the signal mask in effect in the new process image of a spawn operation (if `POSIX_SPAWN_SETSIGMASK` is set in the *spawn-flags* attribute). The default value of this attribute is unspecified.

Return Values Upon successful completion, `posix_spawnattr_getsigmask()` returns 0 and stores the value of the *spawn-sigmask* attribute of *attr* into the object referenced by the *sigmask* parameter. Otherwise, an error number is returned to indicate the error.

Upon successful completion, `posix_spawnattr_setsigmask()` returns 0. Otherwise, an error number is returned to indicate the error.

Errors These functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `posix_spawnattr_setsigmask()` function may fail if:

`EINVAL` The value of the attribute being set is not valid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also posix_spawn(3C), posix_spawnattr_destroy(3C), posix_spawnattr_getflags(3C), posix_spawnattr_getpgroup(3C), posix_spawnattr_getschedparam(3C), posix_spawnattr_getschedpolicy(3C), posix_spawnattr_getsigmask(3C), attributes(5), standards(5)

Name posix_spawn_file_actions_addclose, posix_spawn_file_actions_addopen – add close or open action to spawn file actions object

Synopsis

```
#include <spawn.h>

int posix_spawn_file_actions_addclose(
     posix_spawn_file_actions_t *file_actions, int fildes);

int posix_spawn_file_actions_addopen(
     posix_spawn_file_actions_t *restrict file_actions, int fildes,
     const char *restrict path, int oflag, mode_t mode);
```

Description These functions add or delete a close or open action to a spawn file actions object.

A spawn file actions object is of type `posix_spawn_file_actions_t` (defined in `<spawn.h>`) and is used to specify a series of actions to be performed by a `posix_spawn`(3C) or `posix_spawn`(3C) operation to arrive at the set of open file descriptors for the child process given the set of open file descriptors of the parent.

A spawn file actions object, when passed to `posix_spawn()` or `posix_spawnp()`, specifies how the set of open file descriptors in the calling process is transformed into a set of potentially open file descriptors for the spawned process. This transformation occurs as though the specified sequence of actions was performed exactly once, in the context of the spawned process (prior to execution of the new process image), in the order in which the actions were added to the object. Additionally, when the new process image is executed, any file descriptor (from this new set) which has its `FD_CLOEXEC` flag set is closed (see `posix_spawn`(3C)).

The `posix_spawn_file_actions_addclose()` function adds a close action to the object referenced by *file_actions* that causes the file descriptor fildes to be closed (as if `close`(*fildes*) had been called) when a new process is spawned using this file actions object.

The `posix_spawn_file_actions_addopen()` function adds an open action to the object referenced by *file_actions* that causes the file named by path to be opened (as if `open`(*path*, *oflag*, *mode*) had been called, and the returned file descriptor, if not *fildes*, had been changed to *fildes*) when a new process is spawned using this file actions object. If *fildes* was already an open file descriptor, it is closed before the new file is opened.

The string described by *path* is copied by the `posix_spawn_file_actions_addopen()` function.

Return Values Upon successful completion, these functions return 0. Otherwise, an error number is returned to indicate the error.

Errors These functions will fail if:

`EBADF` The value specified by *fildes* is negative or greater than or equal to {`OPEN_MAX`}.

These functions may fail if:

`EINVAL` The value specified by *file_actions* is invalid.

ENOMEM Insufficient memory exists to add to the spawn file actions object.

It is not considered an error for the *fildes* argument passed to these functions to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is later used during a `posix_spawn()` or `posix_spawnp()` operation.

If a `close(2)` operation is specified for a file descriptor that is not open at the time of the call to `posix_spawn()` or `posix_spawnp()`, the close operation will not cause the `posix_spawn()` or `posix_spawnp()` operation to fail.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also

`close(2)`, `dup(2)`, `open(2)`, `posix_spawn(3C)`, `posix_spawn_file_actions_adddup2(3C)`, `posix_spawn_file_actions_destroy(3C)`, `attributes(5)`, `standards(5)`

Name posix_spawn_file_actions_addclosefrom_np – add closefrom action to spawn file actions object

Synopsis

```
#include <spawn.h>

int posix_spawn_file_actions_addclosefrom_np(
     posix_spawn_file_actions_t *file_actions, int lowfildes);
```

Description The `posix_spawn_file_actions_addclosefrom_np()` function adds a closefrom action to the object referenced by *file_actions* that causes all open file descriptors greater than or equal to *lowfildes* to be closed when a new process is spawned using this file actions object (see `closefrom(3C)`).

A spawn file actions object is as defined in `posix_spawn_file_actions_addclose(3C)`.

Return Values Upon successful completion, the `posix_spawn_file_actions_addclosefrom_np()` function returns 0. Otherwise, an error number is returned to indicate the error.

Errors The `posix_spawn_file_actions_addclosefrom_np()` function will fail if:

`EBADF` The value specified by *lowfildes* is negative.

The `posix_spawn_file_actions_addclosefrom_np()` function may fail if:

`EINVAL` The value specified by *file_actions* is invalid.

`ENOMEM` Insufficient memory exists to add to the spawn file actions object.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `close(2)`, `closefrom(3C)`, `posix_spawn(3C)`, `posix_spawn_file_actions_addclose(3C)`, `attributes(5)`

Name posix_spawn_file_actions_adddup2 – add dup2 action to spawn file actions object

Synopsis

```
#include <spawn.h>

int posix_spawn_file_actions_adddup2(
     posix_spawn_file_actions_t *file_actions, int fildes,
     int newfildes);
```

Description The `posix_spawn_file_actions_adddup2()` function adds a `dup2(3C)` action to the object referenced by *file_actions* that causes the file descriptor *fildes* to be duplicated as *newfildes* (as if `dup2`(*fildes*, *newfildes*) had been called) when a new process is spawned using this file actions object.

A spawn file actions object is as defined in `posix_spawn_file_actions_addclose(3C)`.

Return Values Upon successful completion, the `posix_spawn_file_actions_adddup2()` function returns 0. Otherwise, an error number is be returned to indicate the error.

Errors The `posix_spawn_file_actions_adddup2()` function will fail if:

`EBADF` The value specified by *fildes* or *newfildes* is negative or greater than or equal to {`OPEN_MAX`}.

`ENOMEM` Insufficient memory exists to add to the spawn file actions object.

The `posix_spawn_file_actions_adddup2()` function may fail if:

`EINVAL` The value specified by *file_actions* is invalid.

It is not considered an error for the *fildes* argument passed to `posix_spawn_file_actions_adddup2()` to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is later used during a `posix_spawn(3C)` or `posix_spawnp(3C)` operation.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `dup2(3C)`, `posix_spawn(3C)`, `posix_spawn_file_actions_addclose(3C)`, `posix_spawn_file_actions_destroy(3C)`, `attributes(5)`, `standards(5)`

Name posix_spawn_file_actions_destroy, posix_spawn_file_actions_init – destroy and initialize spawn file actions object

Synopsis

```
#include <spawn.h>

int posix_spawn_file_actions_destroy(
     posix_spawn_file_actions_t *file_actions);

int posix_spawn_file_actions_init(
     posix_spawn_file_actions_t *file_actions);
```

Description The `posix_spawn_file_actions_destroy()` function destroys the object referenced by *file_actions*. The object becomes, in effect, uninitialized. An implementation can cause `posix_spawn_file_actions_destroy()` to set the object referenced by file_actions to an invalid value. A destroyed spawn file actions object can be reinitialized using `posix_spawn_file_actions_init()`. The results of otherwise referencing the object after it has been destroyed are undefined.

The `posix_spawn_file_actions_init()` function initializes the object referenced by *file_actions* to contain no file actions for `posix_spawn`(3C) or `posix_spawnp`(3C) to perform.

A spawn file actions object is as defined in `posix_spawn_file_actions_addclose`(3C).

The effect of initializing an already initialized spawn file actions object is undefined.

Return Values Upon successful completion, these functions return 0. Otherwise, an error number is returned to indicate the error.

Errors The `posix_spawn_file_actions_init()` function will fail if:

`ENOMEM` Insufficient memory exists to initialize the spawn file actions object.

The `posix_spawn_file_actions_destroy()` function will may if:

`EINVAL` The value specified by *file_actions* is invalid.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards`(5).

See Also `posix_spawn`(3C), `posix_spawn_file_actions_addclose`(3C), `attributes`(5), `standards`(5)

Name printf, fprintf, sprintf, snprintf, asprintf – print formatted output

Synopsis

```
#include <stdio.h>

int printf(const char *restrict format,
     /* args*/ ...);

int fprintf(FILE *restrict stream, const char *restrict format,
     /* args*/ ...);

int sprintf(char *restrict s, const char *restrict format,
     /* args*/ ...);

int snprintf(char *restrict s, size_t n,
     const char *restrict format, /* args*/ ...);

int asprintf(char ** ret, const char *restrict format,
     /* args*/ ...);
```

Description The `printf()` function places output on the standard output stream `stdout`.

The `fprintf()` function places output on on the named output stream *stream*.

The `sprintf()` function places output, followed by the null byte (`\0`), in consecutive bytes starting at *s*; it is the user's responsibility to ensure that enough storage is available.

The `snprintf()` function is identical to `sprintf()` with the addition of the argument *n*, which specifies the size of the buffer referred to by *s*. If *n* is 0, nothing is written and *s* can be a null pointer. Otherwise, output bytes beyond the *n*-1st are discarded instead of being written to the array and a null byte is written at the end of the bytes actually written into the array.

The `asprintf()` function is the same as the `sprintf()` function except that it returns, in the *ret* argument, a pointer to a buffer sufficiently large to hold the output string. This pointer should be passed to `free(3C)` to release the allocated storage when it is no longer needed. If sufficient space cannot be allocated, the `asprintf()` function returns -1 and sets *ret* to be a `NULL` pointer.

Each of these functions converts, formats, and prints its arguments under control of the *format*. The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is composed of zero or more directives: *ordinary characters*, which are simply copied to the output stream and *conversion specifications*, each of which results in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier `%` (see below) is replaced by the sequence `%`*n*`$`, where *n* is a decimal integer in the range [1, `NL_ARGMAX`], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the `EXAMPLES` section).

In format strings containing the `%n$` form of conversion specifications, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings containing the `%` form of conversion specifications, each argument in the argument list is used exactly once.

All forms of the `printf()` functions allow for the insertion of a language-dependent radix character in the output string. The radix character is defined by the program's locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (`.`).

Conversion Specifications

Each conversion specification is introduced by the `%` character or by the character sequence `%n$`, after which the following appear in sequence:

- An optional field, consisting of a decimal digit string followed by a `$`, specifying the next argument to be converted. If this field is not provided, the *args* following the last argument converted will be used.
- Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
- An optional minimum *field width*. If the converted value has fewer bytes than the field width, it will be padded with spaces by default on the left; it will be padded on the right, if the left-adjustment flag (`-`), described below, is given to the field width. The field width takes the form of an asterisk (`*`), described below, or a decimal integer.

  If the conversion specifier is `s`, a standard-conforming application (see `standards(5)`) interprets the field width as the minimum number of bytes to be printed; an application that is not standard-conforming interprets the field width as the minimum number of columns of screen display. For an application that is not standard-conforming, `%10s` means if the converted value has a screen width of 7 columns, 3 spaces would be padded on the right.

  If the format is `%ws`, then the field width should be interpreted as the minimum number of columns of screen display.
- An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions (the field is padded with leading zeros); the number of digits to appear after the radix character for the `a`, `A`, `e`, `E`, `f`, and `F` conversions, the maximum number of significant digits for the `g` and `G` conversions; or the maximum number of bytes to be printed from a string in `s` and `S` conversions. The precision takes the form of a period (.) followed either by an asterisk (`*`), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion specifier, the behavior is undefined.

  If the conversion specifier is `s` or `S`, a standard-conforming application (see `standards(5)`) interprets the precision as the maximum number of bytes to be written; an application that is not standard-conforming interprets the precision as the maximum number of columns

of screen display. For an application that is not standard-conforming, `%.5s` would print only the portion of the string that would display in 5 screen columns. Only complete characters are written.

For `%ws`, the precision should be interpreted as the maximum number of columns of screen display. The precision takes the form of a period (`.`) followed by a decimal digit string; a null digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.

- An optional *length modifier* that specified the size of the argument.
- A *conversion specifier* that indicates the type of conversion to be applied.

A field width, or precision, or both can be indicated by an asterisk (*) . In this case, an argument of type `int` supplies the field width or precision. Arguments specifying field width, or precision, or both must appear in that order before the argument, if any, to be converted. A negative field width is taken as a – flag followed by a positive field width. A negative precision is taken as if the precision were omitted. In format strings containing the `%`*n*`$` form of a conversion specification, a field width or precision may be indicated by the sequence *`*`*m*`$`, where *m* is a decimal integer in the range [1, `NL_ARGMAX`] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The *format* can contain either numbered argument specifications (that is, `%`*n*`$` and *`*`*m*`$`), or unnumbered argument specifications (that is, `%` and *), but normally not both. The only exception to this is that `%%` can be mixed with the `%`*n*`$` form. The results of mixing numbered and unnumbered argument specifications in a *format* string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N–1*)th, are specified in the format string.

Flag Characters The flag characters and their meanings are:

`'` The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%F`, `%g`, or `%G`) will be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.

– The result of the conversion will be left-justified within the field. The conversion will be right-justified if this flag is not specified.

`+` The result of a signed conversion will always begin with a sign (+ or –). The conversion will begin with a sign only when a negative value is converted if this flag is not specified.

`space` If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a space will be placed before the result. This means that if the `space` and + flags both appear, the space flag will be ignored.

`#` The value is to be converted to an alternate form. For `c`, `d`, `i`, `s`, and `u` conversions, the flag has no effect. For an `o` conversion, it increases the precision (if necessary) to force the first digit of the result to be a zero. For `x` or `X` conversion, a non-zero result will have `0x` (or `0X`) prepended to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result will always contain a radix character, even if no digits follow the radix character. Without this flag, the radix character appears in the result of these conversions only if a digit follows it. For `g` and `G` conversions, trailing zeros will not be removed from the result as they normally are.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `0` and – flags both appear, the `0` flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag will be ignored. If the `0` and `'` flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.

Length Modifiers The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `signed char` or `unsigned char` argument (the argument will have been promoted according to the integer promotions, but its value will be converted to `signed char` or `unsigned char` before printing); or that a following `n` conversion specifier applies to a pointer to a `signed char` argument.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `short` or `unsigned short` argument (the argument will have been promoted according to the integer promotions, but its value will be converted to `short` or `unsigned short` before printing); or that a following `n` conversion specifier applies to a pointer to a `short` argument.

`l (ell)` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long` or `unsigned long` argument; that a following `n` conversion specifier applies to a pointer to a `long` argument; that a following `c` conversion specifier applies to a `wint_t` argument; that a following `s` conversion specifier applies to a pointer to a `wchar_t` argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

`ll (ell-ell)` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long long` or `unsigned long long` argument; or that a following `n` conversion specifier applies to a pointer to a `long long` argument.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an `intmax_t` or `uintmax_t` argument; or that a following `n` conversion specifier applies to a pointer to an `intmax_t` argument. See NOTES.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `size_t` or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to `size_t` argument.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned type argument; or that a following `n` conversion specifier applies to a pointer to a `ptrdiff_t` argument.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

Conversion Specifiers

Each conversion specifier results in fetching zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are ignored.

The conversion specifiers and their meanings are:

`d`, `i` The `int` argument is converted to a signed decimal in the style `[−]`*dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

`o` The `unsigned int` argument is converted to unsigned octal format in the style *dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

`u` The `unsigned int` argument is converted to unsigned decimal format in the style *dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

`x` The `unsigned int` argument is converted to unsigned hexadecimal format in the style *dddd*; the letters `abcdef` are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

`X` Behaves the same as the `x` conversion specifier except that letters `ABCDEF` are used instead of `abcdef`.

`f`, `F` The `double` argument is converted to decimal notation in the style [−]*ddd*.*ddd*, where the number of digits after the radix character (see `setlocale(3C)`) is equal to the precision specification. If the precision is missing it is taken as 6; if the precision is explicitly 0 and the # flag is not specified, no radix character appears. If a radix character appears, at least 1 digit appears before it. The converted value is rounded to fit the specified output format according to the prevailing floating point rounding direction mode. If the conversion is not exact, an inexact exception is raised.

For the `f` specifier, a double argument representing an infinity or NaN is converted in the style of the `e` conversion specifier, except that for an infinite argument, “infinity” or “Infinity” is printed when the precision is at least 8 and “inf” or “Inf” is printed otherwise.

For the F specifier, a double argument representing an infinity or NaN is converted in the SUSv3 style of the E conversion specifier, except that for an infinite argument, “INFINITY” is printed when the precision is at least 8 and or “INF” is printed otherwise.

`e`, `E` The `double` argument is converted to the style [−]*d*.*ddd*`e`±*dd*, where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision. When the precision is missing it is taken as 6; if the precision is 0 and the # flag is not specified, no radix character appears. The `E` conversion specifier will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. The converted value is rounded to fit the specified output format according to the prevailing floating point rounding direction mode. If the conversion is not exact, an inexact exception is raised.

Infinity and NaN values are handled in one of the following ways:

SUSv3 For the `e` specifier, a `double` argument representing an infinity is printed as “[−]`infinity`”, when the precision for the conversion is at least 7 and as “[−]`inf`” otherwise. A `double` argument representing a NaN is printed as “[−]`nan`”. For the `E` specifier, “`INF`”, “`INFINITY`”, and “`NAN`” are printed instead of “`inf`”, “`infinity`”, and “`nan`”, respectively. Printing of the sign follows the rules described above.

Default A `double` argument representing an infinity is printed as “[−]`Infinity`”, when the precision for the conversion is at least 7 and as “[−]`Inf`” otherwise. A double argument representing a NaN is printed as “[−]`NaN`”. Printing of the sign follows the rules described above.

`g`, `G` The `double` argument is printed in style `f` or `e` (or in style `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted: style `e` (or `E`) will be used only if the exponent resulting from the

conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result. A radix character appears only if it is followed by a digit.

A `double` argument representing an infinity or NaN is converted in the style of the `e` or `E` conversion specifier, except that for an infinite argument, “infinity”, “INFINITY”, or “Infinity” is printed when the precision is at least 8 and “inf”, “INF”, or “Inf” is printed otherwise.

`a, A` A `double` argument representing a floating-point number is converted in the style “[-]0*xh.hhhhp±d*”, where the single hexadecimal digit preceding the radix point is 0 if the value converted is zero and 1 otherwise and the number of hexadecimal digits after it is equal to the precision; if the precision is missing, the number of digits printed after the radix point is 13 for the conversion of a double value, 16 for the conversion of a long double value on x86, and 28 for the conversion of a long double value on SPARC; if the precision is zero and the '#' flag is not specified, no decimal-point character will appear. The letters “`abcdef`” are used for `a` conversion and the letters “`ABCDEF`” for `A` conversion. The `A` conversion specifier produces a number with 'X' and 'P' instead of 'x' and 'p'. The exponent will always contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

The converted value is rounded to fit the specified output format according to the prevailing floating point rounding direction mode. If the conversion is not exact, an inexact exception is raised.

A `double` argument representing an infinity or NaN is converted in the SUSv3 style of an `e` or `E` conversion specifier.

`c` The `int` argument is converted to an `unsigned char`, and the resulting byte is printed.

If an `l` (ell) qualifier is present, the `wint_t` argument is converted as if by an `ls` conversion specification with no precision and an argument that points to a two-element array of type `wchar_t`, the first element of which contains the `wint_t` argument to the `ls` conversion specification and the second element contains a null wide-character.

`C` Same as `lc`.

`wc` The `int` argument is converted to a wide character (`wchar_t`), and the resulting wide character is printed.

`s` The argument must be a pointer to an array of `char`. Bytes from the array are written up to (but not including) any terminating null byte. If a precision is specified, a standard-conforming application (see `standards(5)`) will write only the number of bytes specified by precision; an application that is not standard-conforming will write only the portion of the string that will display in the number of columns of screen

display specified by precision. If the precision is not specified, it is taken to be infinite, so all bytes up to the first null byte are printed. An argument with a null value will yield undefined results.

If an `l` (ell) qualifier is present, the argument must be a pointer to an array of type `wchar_t`. Wide-characters from the array are converted to characters (each as if by a call to the `wcrtomb(3C)` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide-character is converted) up to and including a terminating null wide-character. The resulting characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array must contain a null wide-character. If a precision is specified, no more than that many characters (bytes) are written (including shift sequences, if any), and the array must contain a null wide-character if, to equal the character sequence length given by the precision, the function would need to access a wide-character one past the end of the array. In no case is a partial character written.

`S` Same as `ls`.

`ws` The argument must be a pointer to an array of `wchar_t`. Bytes from the array are written up to (but not including) any terminating null character. If the precision is specified, only that portion of the wide-character array that will display in the number of columns of screen display specified by precision will be written. If the precision is not specified, it is taken to be infinite, so all wide characters up to the first null character are printed. An argument with a null value will yield undefined results.

`p` The argument must be a pointer to `void`. The value of the pointer is converted to a set of sequences of printable characters, which should be the same as the set of sequences that are matched by the `%p` conversion of the `scanf(3C)` function.

`n` The argument must be a pointer to an integer into which is written the number of bytes written to the output standard I/O stream so far by this call to one of the `printf()` functions. No argument is converted.

`%` Print a `%`; no argument is converted. The entire conversion specification must be %%.

If a conversion specification does not match one of the above forms, the behavior is undefined.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by `printf()` and `fprintf()` are printed as if the `putc(3C)` function had been called.

The `st_ctime` and `st_mtime` fields of the file will be marked for update between the call to a successful execution of `printf()` or `fprintf()` and the next successful completion of a call to `fflush(3C)` or `fclose(3C)` on the same stream or a call to `exit(3C)` or `abort(3C)`.

Return Values The `printf()`, `fprintf()`, `sprintf()`, and `asprintf()` functions return the number of bytes transmitted (excluding the terminating null byte in the case of `sprintf()` and `asprintf()`).

The `snprintf()` function returns the number of bytes that would have been written to *s* if *n* had been sufficiently large (excluding the terminating null byte.) If the value of *n* is 0 on a call to `snprintf()`, *s* can be a null pointer and the number of bytes that would have been written if *n* had been sufficiently large (excluding the terminating null byte) is returned.

Each function returns a negative value if an output error was encountered.

Errors For the conditions under which `printf()` and `fprintf()` will fail and may fail, refer to `fputc(3C)` or `fputwc(3C)`.

The `snprintf()` function will fail if:

`EOVERFLOW` The value of *n* is greater than `INT_MAX` or the number of bytes needed to hold the output excluding the terminating null is greater than `INT_MAX`.

The `printf()`, `fprintf()`, `sprintf()`, and `snprintf()` functions may fail if:

`EILSEQ` A wide-character code that does not correspond to a valid character has been detected.

`EINVAL` There are insufficient arguments.

The `printf()`, `fprintf()`, and `asprintf()` functions may fail due to an underlying `malloc(3C)` failure if:

`EAGAIN` Storage space is temporarily unavailable.

`ENOMEM` Insufficient storage space is available.

Usage If the application calling the `printf()` functions has any objects of type `wint_t` or `wchar_t`, it must also include the header `<wchar.h>` to have these objects defined.

Escape Character Sequences It is common to use the following escape sequences built into the C language when entering format strings for the `printf()` functions, but these sequences are processed by the C compiler, not by the `printf()` function.

`\a` Alert. Ring the bell.

`\b` Backspace. Move the printing position to one character before the current position, unless the current position is the start of a line.

`\f` Form feed. Move the printing position to the initial printing position of the next logical page.

`\n` Newline. Move the printing position to the start of the next line.

`\r` Carriage return. Move the printing position to the start of the current line.

`\t` Horizontal tab. Move the printing position to the next implementation-defined horizontal tab position on the current line.

`\v` Vertical tab. Move the printing position to the start of the next implementation-defined vertical tab position.

In addition, the C language supports character sequences of the form

\octal-number

and

\hex-number

which translates into the character represented by the octal or hexadecimal number. For example, if ASCII representations are being used, the letter 'a' may be written as '\141' and 'Z' as '\132'. This syntax is most frequently used to represent the null character as '\0'. This is exactly equivalent to the numeric constant zero (0). Note that the octal number does not include the zero prefix as it would for a normal octal constant. To specify a hexadecimal number, omit the zero so that the prefix is an 'x' (uppercase 'X' is not allowed in this context). Support for hexadecimal sequences is an ANSI extension. See `standards(5)`.

Examples

EXAMPLE 1 To print the language-independent date and time format, the following statement could be used:

```
printf (format, weekday, month, day, hour, min);
```

For American usage, *format* could be a pointer to the string:

```
"%s, %s %d, %d:%.2d\n"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, *format* could be a pointer to the string:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the message:

```
Sonntag, 3. Juli, 10:02
```

EXAMPLE 2 To print a date and time in the form `Sunday, July 3, 10:02`, where `weekday` and `month` are pointers to null-terminated strings:

```
printf("%s, %s %i, %d:%.2d", weekday, month, day, hour, min);
```

EXAMPLE 3 To print pi to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

Default **EXAMPLE 4** The following example applies only to applications that are not standard-conforming. To print a list of names in columns which are 20 characters wide:

```
printf("%20s%20s%20s", lastname, firstname, middlename);
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	See below.
Standard	See below.

All of these functions can be used safely in multithreaded applications, as long as setlocale(3C) is not being called to change the locale. The `sprintf()` and `snprintf()` functions are Async-Signal-Safe.

See standards(5) for the standards conformance of `printf()`, `fprintf()`, `sprintf()`, and `snprintf()`. The `asprintf()` function is modeled on the one that appears in the FreeBSD, NetBSD, and GNU C libraries.

See Also

exit(2), lseek(2), write(2), abort(3C), ecvt(3C), exit(3C), fclose(3C), fflush(3C), fputwc(3C), free(3C), malloc(3C), putc(3C), scanf(3C), setlocale(3C), stdio(3C), vprintf(3C), wcstombs(3C), wctomb(3C), attributes(5), environ(5), standards(5)

Notes

If the `j` length modifier is used, 32-bit applications that were compiled using `c89` on releases prior to Solaris 10 will experience undefined behavior.

The `snprintf()` return value when $n = 0$ was changed in the Solaris 10 release. The change was based on the SUSv3 specification. The previous behavior was based on the initial SUSv2 specification, where `snprintf()` when $n = 0$ returns an unspecified value less than 1.

Name priv_addset, priv_allocset, priv_copyset, priv_delset, priv_emptyset, priv_basicset, priv_fillset, priv_freeset, priv_intersect, priv_inverse, priv_isemptyset, priv_isequalset, priv_isfullset, priv_ismember, priv_issubset, priv_union – privilege set manipulation functions

Synopsis

```
#include <priv.h>

int priv_addset(priv_set_t *sp, const char *priv);

priv_set_t *priv_allocset(void);

void priv_copyset(const priv_set_t *src, priv_set_t *dst);

int priv_delset(priv_set_t *sp, const char *priv);

void priv_emptyset(priv_set_t *sp);

void priv_basicset(priv_set_t *sp);

void priv_fillset(priv_set_t *sp);

void priv_freeset(priv_set_t *sp);

void priv_intersect(const priv_set_t *src, priv_set_t *dst);

void priv_inverse(priv_set_t *sp);

boolean_t priv_isemptyset(const priv_set_t *sp);

boolean_t priv_isequalset(const priv_set_t *src, const priv_set_t *dst);

boolean_t priv_isfullset(const priv_set_t *sp);

boolean_t priv_ismember(const priv_set_t *sp, const char *priv);

boolean_t priv_issubset(const priv_set_t *src, const priv_set_t *dst);

void priv_union(const priv_set_t *src, priv_set_t *dst);
```

Description The *sp*, *src*, and *dst* arguments point to privilege sets. The *priv* argument points to a named privilege.

The `priv_addset()` function adds the named privilege *priv* to *sp*.

The `priv_allocset()` function allocates sufficient memory to contain a privilege set. The value of the returned privilege set is indeterminate. The function returns `NULL` and sets `errno` when it fails to allocate memory.

The `priv_copyset()` function copies the set *src* to *dst*.

The `priv_delset()` function removes the named privilege *priv* from *sp*.

The `priv_emptyset()` function clears all privileges from *sp*.

The `priv_basicset()` function copies the basic privilege set to *sp*.

The `priv_fillset()` function asserts all privileges in *sp*, including the privileges not currently defined in the system.

The `priv_freeset()` function frees the storage allocated by `priv_allocset()`.

The `priv_intersect()` function intersects *src* with *dst* and places the results in *dst*.

The `priv_inverse()` function inverts the privilege set given as argument in place.

The `priv_isemptyset()` function checks whether the argument is an empty set.

The `priv_isequalset()` function checks whether the privilege set *src* is equal to *dst*.

The `priv_isfullset()` function checks whether the argument is a full set. A full set is a set with all bits set, regardless of whether the privilege is currently defined in the system.

The `priv_ismember()` function checks whether the named privilege *priv* is a member of *sp*.

The `priv_issubset()` function checks whether *src* is a subset of *dst*.

The `priv_union()` function takes the union of *src* and *dst* and places the result in *dst*.

Return Values Upon successful completion, `priv_allocset()` returns a pointer to an opaque data structure. It returns `NULL` if memory allocation fails and sets `errno` to indicate the error.

Upon successful completion, `priv_isemptyset()`, `priv_isfullset()`, `priv_isequalset()`, `priv_issubset()`, and `priv_ismember()` return `B_TRUE`. Otherwise, they return `B_FALSE`.

Upon successful completion, `priv_delset()` and `priv_addset()` return 0. Otherwise, they return -1 and set `errno` to indicate the error.

Errors The `priv_allocset()` function will fail if:

`ENOMEM` The physical limits of the system are exceeded by the memory allocation needed to hold a privilege set.

`EAGAIN` There is insufficient memory for allocation to hold a privilege set. The application can try again later.

The `priv_delset()` and `priv_addset()` functions will fail if:

`EINVAL` The privilege argument is not a valid privilege name.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also setppriv(2), malloc(3C), priv_str_to_set(3C), attributes(5), privileges(5)

Notes The functions that compare sets operate on all bits of the set, regardless of whether the specific privileges are currently defined in the system.

Name priv_set, priv_ineffect – change privilege sets and check whether privileges are set

Synopsis

```
#include <priv.h>

int priv_set(priv_op_t op, priv_ptype_t  which...);

boolean_t priv_ineffect(const char *priv);
```

Description The `priv_set()` function is a convenient wrapper for the `setppriv(2)` function. It takes three or more arguments. The operation argument, *op*, can be one of `PRIV_OFF`, `PRIV_ON` or `PRIV_SET`. The *which* argument is the name of the privilege set to change. The third argument is a list of zero or more privilege names terminated with a null pointer. If *which* is the special pseudo set `PRIV_ALLSETS`, the operation should be applied to all privilege sets.

The specified privileges are converted to a binary privilege set and `setppriv()` is called with the same *op* and *which* arguments. When called with `PRIV_ALLSETS` as the value for the *which* argument, `setppriv()` is called for each set in turn, aborting on the first failed call.

The `priv_ineffect()` function is a conventient wrapper for the `getppriv(2)` function. The *priv* argument specifies the name of the privilege for which this function checks its presence in the effective set.

Return Values Upon successful completion, `priv_set()` return 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

If *priv* is a valid privilege that is a member of the effective set, `priv_ineffect()` returns `B_TRUE`. Otherwise, it returns `B_FALSE` and sets `errno` to incicate the error.

Errors The `priv_set()` function will fail if:

`EINVAL` The value of *op* or *which* is out of range.

`ENOMEM` Insufficient memory was allocated.

`EPERM` The application attempted to add privileges to `PRIV_LIMIT` or `PRIV_PERMITTED`, or the application attempted to add privileges to `PRIV_INHERITABLE` or `PRIV_EFFECTIVE` that were not in `PRIV_PERMITTED`.

The `priv_ineffect()` function will fail if:

`EINVAL` The privilege specified by *priv* is invalid.

`ENOMEM` Insufficient memory was allocated.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also setppriv(2), priv_str_to_set(3C), attributes(5), privileges(5)

Name priv_str_to_set, priv_set_to_str, priv_getbyname, priv_getbynum, priv_getsetbyname, priv_getsetbynum, priv_gettext – privilege name functions

Synopsis

```
#include <priv.h>

priv_set_t *priv_str_to_set(const char *buf, const char *sep,
     const char **endptr);

char *priv_set_to_str(const priv_set_t *set, char sep, int flag);

int priv_getbyname(const char *privname);

const char *priv_getbynum(int privnum);

int priv_getsetbyname(const char *privsetname);

const char *priv_getsetbynum(int privname);

char *priv_gettext(const char *privname);
```

Description The `priv_str_to_set()` function maps the privilege specification in *buf* to a privilege set. It returns a privilege set on success or `NULL` on failure. If an error occurs when parsing the string, a pointer to the remainder of the string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer. If an error occurs when allocating memory, `errno` is set and the object pointed to by *endptr* is set to the null pointer, provided that *endptr* is not a null pointer.

The application is responsible for freeing the returned privilege set using `priv_freeset(3C)`.

A privilege specification should contain one or more privilege names, separated by characters in *sep* using the same algorithm as `strtok(3C)`. Privileges can optionally be preceded by a dash (-) or an exclamation mark (!), in which case they are excluded from the resulting set. The special strings “none” for the empty set, “all” for the set of all privileges, “zone” for the set of all privileges available within the caller's zone, and “basic” for the set of basic privileges are also recognized. Set specifications are interpreted from left to right.

The `priv_set_to_str()` function converts the privilege set to a sequence of privileges separated by *sep*, returning the a pointer to the dynamically allocated result. The application is responsible for freeing the memory using `free(3C)`.

To maintain future compatibility, the “basic” set of privileges is included as “basic,!missing_basic_priv1,...”. When further currently unprivileged operations migrate to the basic privilege set, the conversion back of the result with `priv_str_to_set()` includes the additional basic privileges, guaranteeing that the resulting privilege set carries the same privileges. This behavior is the default and is equivalent to specifying a *flag* argument of `PRIV_STR_PORT`. When specifying a *flag* argument of `PRIV_STR_LIT`, the result does not treat basic privileges differently and the privileges present are all literally presented in the output. A flag argument of `PRIV_STR_SHORT` attempts to arrive at the shortest output, using the tokens “basic”, “zone”, “all”, and negated privileges. This output is most useful for trace output.

The `priv_getbyname()` and `priv_getsetbyname()` functions map privilege names and privilege set names to numbers. The numbers returned are valid for the current kernel instance only and could change at the next boot. Only the privilege names should be committed to persistent storage. The numbers should not be committed to persistent storage. Both functions return -1 on error, setting `errno` to `EINVAL`.

The `priv_getbynum()` and `priv_getsetbynum()` functions map privileges numbers to names. The strings returned point to shared storage that should not be modified and is valid for the lifetime of the process. Both functions return `NULL` on error, setting `errno` to `EINVAL`.

The `priv_gettext()` function returns a pointer to a string consisting of one or more newline-separated lines of text describing the privilege. The text is localized using {`LC_MESSAGES`}. The application is responsibe for freeing the memory returned.

These functions pick up privileges allocated during the lifetime of the process using `priv_getbyname(9F)` by refreshing the internal data structures when necessary.

Return Values Upon successful completion, `priv_str_to_set()` and `priv_set_to_str()` return a non-null pointer to allocated memory that should be freed by the application using the appropriate functions when it is no longer referenced.

The `priv_getbynum()` and `priv_getsetbynum()` functions return non-null pointers to constant memory that should not be modified or freed by the application. Otherwise, `NULL` is returned and `errno` is set to indicate the error.

Upon successful completion, `priv_getbyname()` and `priv_getsetbyname()` return a non-negative integer. Otherwise, -1 is returned and `errno` is set to indicate the error.

Upon successful completion, `priv_gettext()` returns a non-null value. It returns `NULL` if an error occurs or no descriptive text for the specified privilege can be found.

Errors The `priv_str_to_set()` and `priv_set_to_str()` functions will fail if:

`ENOMEM` The physical limits of the system are exceeded by the memory allocation needed to hold a privilege set.

`EAGAIN` There is not enough memory available to allocate sufficient memory to hold a privilege set, but the application could try again later.

All of these functions will fail if:

`EINVAL` One or more of the arguments is invalid.

Examples **EXAMPLE 1** List all the sets and privileges defined in the system.

The following example lists all the sets and privileges defined in the system.

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

EXAMPLE 1 List all the sets and privileges defined in the system. *(Continued)*

```
/* list all the sets and privileges defined in the system */

const char *name;
int i;

printf("Each process has the following privilege sets:\n");
for (i = 0; (name = priv_getsetbynum(i++)) != NULL; )
        printf("\t%s\n", name);

printf("Each set can contain the following privileges:\n");
for (i = 0; (name = priv_getbynum(i++)) != NULL; )
        printf("\t%s\n", name);
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

free(3C), priv_set(3C), attributes(5), privileges(5), priv_getbyname(9F)

Name pset_getloadavg – get system load averages for a processor set

Synopsis

```
#include <sys/pset.h>
#include <sys/loadavg.h>

int pset_getloadavg(psetid_t pset, double loadavg[ ], int nelem);
```

Description The `pset_getloadavg()` function returns the number of processes assigned to the specified processor set that are in the system run queue, averaged over various periods of time. Up to *nelem* samples are retrieved and assigned to successive elements of *loadavg*[ ]. The system imposes a maximum of 3 samples, representing averages over the last 1, 5, and 15 minutes, respectively.

The `LOADAVG_1MIN`, `LOADAVG_5MIN`, and `LOADAVG_15MIN` indices, defined in `<sys/loadavg.h>`, can be used to extract the data from the appropriate element of the *loadavg*[ ] array.

If pset is `PS_NONE`, the load average for processes not assigned to a processor set is returned.

If pset is `PS_MYID`, the load average for the processor set to which the caller is bound is returned. If the caller is not bound to a processor set, the result is the same as if `PS_NONE` was specified.

Return Values Upon successful completion, the number of samples actually retrieved is returned. If the load average was unobtainable or the processor set does not exist, −1 is returned and `errno` is set to indicate the error.

Errors The `pset_getloadavg()` function will fail if:

`EINVAL` The number of elements specified is less than 0, or an invalid processor set ID was specified.

The caller is in a non-global zone, the pools facility is active, and the specified processor set is not that of the zone's pool.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also `uptime(1)`, `w(1)`, `psrset(1M)`, `prstat(1M)`, `pset_bind(2)`, `pset_create(2)`, `Kstat(3PERL)`, `attributes(5)`

Name psignal, psiginfo – system signal messages

Synopsis

```
#include <siginfo.h>

void psignal(int sig, const char *s);

void psiginfo(siginfo_t *pinfo, char *s);
```

Description The `psignal()` and `psiginfo()` functions produce messages on the standard error output describing a signal. The *sig* argument is a signal that may have been passed as the first argument to a signal handler. The *pinfo* argument is a pointer to a `siginfo` structure that may have been passed as the second argument to an enhanced signal handler. See `sigaction(2)`. The argument string *s* is printed first, followed by a colon and a blank, followed by the message and a NEWLINE character.

Usage Messages printed from these functions are in the native language specified by the `LC_MESSAGES` locale category. See `setlocale(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `sigaction(2)`, `gettext(3C)`, `perror(3C)`, `setlocale(3C)`, `siginfo.h(3HEAD)`, `signal.h(3HEAD)`, `attributes(5)`

Name pthread_atfork – register fork handlers

Synopsis

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

int pthread_atfork(void (*prepare) (void), void (*parent) (void),
     void (*child) (void));
```

Description The `pthread_atfork()` function declares fork handlers to be called prior to and following `fork(2)`, within the thread that called `fork()`. The order of calls to `pthread_atfork()` is significant.

Before `fork()` processing begins, the *prepare* fork handler is called. The *prepare* handler is not called if its address is `NULL`.

The *parent* fork handler is called after `fork()` processing finishes in the parent process, and the *child* fork handler is called after `fork()` processing finishes in the child process. If the address of *parent* or *child* is `NULL`, then its handler is not called.

The *prepare* fork handler is called in LIFO (last-in first-out) order, whereas the *parent* and *child* fork handlers are called in FIFO (first-in first-out) order. This calling order allows applications to preserve locking order.

Return Values Upon successful completion, `pthread_atfork()` returns `0`. Otherwise, an error number is returned.

Errors The `pthread_atfork()` function will fail if:

`ENOMEM` Insufficient table space exists to record the fork handler addresses.

Usage Solaris threads do not offer `pthread_atfork()` functionality (there is no `thr_atfork()` interface). However, a Solaris threads application can call `pthread_atfork()` to ensure `fork()`–safety, since the two thread APIs are interoperable. See`fork(2)` for information relating to `fork()` in a Solaris threads environment in Solaris 10 relative to previous releases.

Examples **EXAMPLE 1** Make a library safe with respect to `fork()`.

All multithreaded applications that call `fork()` in a POSIX threads program and do more than simply call `exec(2)` in the child of the fork need to ensure that the child is protected from deadlock.

Since the "fork-one" model results in duplicating only the thread that called `fork()`, it is possible that at the time of the call another thread in the parent owns a lock. This thread is not duplicated in the child, so no thread will unlock this lock in the child. Deadlock occurs if the single thread in the child needs this lock.

The problem is more serious with locks in libraries. Since a library writer does not know if the application using the library calls `fork()`, the library must protect itself from such a deadlock

EXAMPLE 1 Make a library safe with respect to `fork()`. *(Continued)*

scenario. If the application that links with this library calls `fork()` and does not call `exec()` in the child, and if it needs a library lock that may be held by some other thread in the parent that is inside the library at the time of the fork, the application deadlocks inside the library.

The following describes how to make a library safe with respect to `fork()` by using `pthread_atfork()`.

1. Identify all locks used by the library (for example `{L1, . . .Ln}`). Identify also the locking order for these locks (for example `{L1 . . .Ln}, as well.`)
2. Add a call to `pthread_atfork(f1, f2, f3)` in the library's *.init* section. `f1`, `f2`, `f3` are defined as follows:

```
f1( )
{
        /* ordered in lock order */
        pthread_mutex_lock(L1);
        pthread_mutex_lock( . . .);
        pthread_mutex_lock(Ln);
}

f2( )
{
        pthread_mutex_unlock(L1);
        pthread_mutex_unlock( . . .);
        pthread_mutex_unlock(Ln);
}

f3( )
{
        pthread_mutex_unlock(L1);
        pthread_mutex_unlock( . . .);
        pthread_mutex_unlock(Ln);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also exec(2), fork(2), atexit(3C), attributes(5), standards(5)

Name pthread_attr_getdetachstate, pthread_attr_setdetachstate – get or set detachstate attribute

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr,
     int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
     int *detachstate);
```

Description The *detachstate* attribute controls whether the thread is created in a detached state. If the thread is created detached, then use of the ID of the newly created thread by the `pthread_detach()` or `pthread_join()` function is an error.

The `pthread_attr_setdetachstate()` and `pthread_attr_getdetachstate()`, respectively, set and get the *detachstate* attribute in the *attr* object.

The *detachstate* can be set to either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`. A value of `PTHREAD_CREATE_DETACHED` causes all threads created with *attr* to be in the detached state, whereas using a value of `PTHREAD_CREATE_JOINABLE` causes all threads created with *attr* to be in the joinable state. The default value of the *detachstate* attribute is `PTHREAD_CREATE_JOINABLE`.

Return Values Upon successful completion, `pthread_attr_setdetachstate()` and `pthread_attr_getdetachstate()` return a value of `0`. Otherwise, an error number is returned to indicate the error.

The `pthread_attr_getdetachstate()` function stores the value of the *detachstate* attribute in *detachstate* if successful.

Errors The `pthread_attr_setdetachstate()` or `pthread_attr_getdetachstate()` functions may fail if:

`EINVAL` *attr* or *detachstate* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_attr_init(3C)`, `pthread_attr_setstackaddr(3C)`, `pthread_attr_setstacksize(3C)`, `pthread_create(3C)`, `attributes(5)`, `standards(5)`

Name pthread_attr_getguardsize, pthread_attr_setguardsize – get or set thread guardsize attribute

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
     size_t *restrict guardsize);

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

Description The *guardsize* attribute controls the size of the guard area for the created thread's stack. The *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error results (possibly in a `SIGSEGV` signal being delivered to the thread).

The *guardsize* attribute is provided to the application for two reasons:

1. Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads will never overflow their stack, can save system resources by turning off guard areas.
2. When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

The `pthread_attr_getguardsize()` function gets the *guardsize* attribute in the *attr* object. This attribute is returned in the *guardsize* parameter.

The `pthread_attr_setguardsize()` function sets the *guardsize* attribute in the *attr* object. The new value of this attribute is obtained from the *guardsize* parameter. If *guardsize* is `0`, a guard area will not be provided for threads created with *attr*. If *guardsize* is greater than `0`, a guard area of at least size *guardsize* bytes is provided for each thread created with *attr*.

A conforming implementation is permitted to round up the value contained in *guardsize* to a multiple of the configurable system variable `PAGESIZE`. If an implementation rounds up the value of *guardsize* to a multiple of `PAGESIZE`, a call to `pthread_attr_getguardsize()` specifying *attr* will store in the *guardsize* parameter the guard size specified by the previous `pthread_attr_setguardsize()` function call.

The default value of the *guardsize* attribute is `PAGESIZE` bytes. The actual value of `PAGESIZE` is implementation-dependent and may not be the same on all implementations.

If the *stackaddr* attribute has been set (that is, the caller is allocating and managing its own thread stacks), the *guardsize* attribute is ignored and no protection will be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

Return Values If successful, the `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions will fail if:

`EINVAL` The attribute *attr* is invalid.

`EINVAL` The parameter *guardsize* is invalid.

`EINVAL` The parameter *guardsize* contains an invalid value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `sysconf(3C)`, `pthread_attr_init(3C)`, `attributes(5)`, `standards(5)`

Name pthread_attr_getinheritsched, pthread_attr_setinheritsched – get or set inheritsched attribute

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
     int *restrict inheritsched);

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

Description The functions `pthread_attr_setinheritsched()` and `pthread_attr_getinheritsched()`, respectively, set and get the *inheritsched* attribute in the *attr* argument.

When the attribute objects are used by `pthread_create()`, the *inheritsched* attribute determines how the other scheduling attributes of the created thread are to be set:

`PTHREAD_INHERIT_SCHED` Specifies that the scheduling policy and associated attributes are to be inherited from the creating thread, and the scheduling attributes in this *attr* argument are to be ignored.

`PTHREAD_EXPLICIT_SCHED` Specifies that the scheduling policy and associated attributes are to be set to the corresponding values from this attribute object.

The symbols `PTHREAD_INHERIT_SCHED` and `PTHREAD_EXPLICIT_SCHED` are defined in the header `<pthread.h>`.

Return Values If successful, the `pthread_attr_setinheritsched()` and `pthread_attr_getinheritsched()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_attr_setinheritsched()` or `pthread_attr_getinheritsched()` functions may fail if:

`EINVAL` *attr* or *inheritsched* is invalid.

Usage After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also pthread_attr_getschedparam(3C), pthread_attr_init(3C), pthread_attr_setscope(3C), pthread_attr_setschedpolicy(3C), pthread_create(3C), pthread_setschedparam(3C), attributes(5), standards(5)

Name pthread_attr_getschedparam, pthread_attr_setschedparam – get or set schedparam attribute

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
     struct sched_param *restrict param);

int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
     const struct sched_param *restrict param);
```

Description The functions `pthread_attr_setschedparam()` and `pthread_attr_getschedparam()`, respectively, set and get the scheduling parameter attributes in the *attr* argument. The contents of the *param* structure are defined in `<sched.h>`. The only required member of *param* is *sched_priority*.

Return Values If successful, the `pthread_attr_setschedparam()` and `pthread_attr_getschedparam()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_attr_setschedparam()` function may fail if:

`EINVAL` *attr* is invalid.

The `pthread_attr_getschedparam()` function may fail if:

`EINVAL` *attr* or *param* is invalid.

Usage After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_attr_init(3C)`, `pthread_attr_setscope(3C)`, `pthread_attr_setinheritsched(3C)`, `pthread_attr_setschedpolicy(3C)`, `pthread_create(3C)`, `pthread_setschedparam(3C)`, `attributes(5)`, `standards(5)`

Name pthread_attr_getschedpolicy, pthread_attr_setschedpolicy – get or set schedpolicy attribute

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
     int *restrict policy);

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

Description The functions `pthread_attr_setschedpolicy()` and `pthread_attr_getschedpolicy()`, respectively, set and get the *schedpolicy* attribute in the *attr* argument.

Supported values of *policy* include `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`, which are defined by the header `<sched.h>`. When threads executing with the scheduling policy `SCHED_FIFO` or `SCHED_RR` are waiting on a mutex, they acquire the mutex in priority order when the mutex is unlocked.

See `sched.h`(3HEAD) for a description of all defined policy values. Valid policy values can also be obtained from `pthread_getschedparam`(3C) and `sched_getscheduler`(3C).

Return Values If successful, the `pthread_attr_setschedpolicy()` and `pthread_attr_getschedpolicy()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_attr_setschedpolicy()` or `pthread_attr_getschedpolicy()` function may fail if:

`EINVAL` *attr* or *policy* is invalid.

Usage After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards`(5).

See Also `pthread_attr_init`(3C), `pthread_attr_setscope`(3C), `pthread_attr_setinheritsched`(3C), `pthread_attr_setschedparam`(3C), `pthread_create`(3C), `pthread_getschedparam`(3C), `sched.h`(3HEAD), `sched_getscheduler`(3C), `attributes`(5), `standards`(5)

Name pthread_attr_getscope, pthread_attr_setscope – get or set contentionscope attribute

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_attr_getscope(const pthread_attr_t *restrict attr,
     int *restrict contentionscope);

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

Description The `pthread_attr_setscope()` and `pthread_attr_getscope()` functions are used to set and get the *contentionscope* attribute in the *attr* object.

The *contentionscope* attribute can have the value `PTHREAD_SCOPE_SYSTEM`, signifying system scheduling contention scope, or `PTHREAD_SCOPE_PROCESS`, signifying process scheduling contention scope.

The symbols `PTHREAD_SCOPE_SYSTEM` and `PTHREAD_SCOPE_PROCESS` are defined by the header `<pthread.h>`.

Return Values If successful, the `pthread_attr_setscope()` and `pthread_attr_getscope()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_attr_setscope()`, or `pthread_attr_getscope()`, function may fail if:

`EINVAL` *attr* or *contentionscope* is invalid.

Usage After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_attr_init(3C)`, `pthread_attr_setinheritsched(3C)`, `pthread_attr_setschedpolicy(3C)`, `pthread_attr_setschedparam(3C)`, `pthread_create(3C)`, `pthread_setschedparam(3C)`, `attributes(5)`, `standards(5)`

Name pthread_attr_getstack, pthread_attr_setstack – get or set stack attributes

Synopsis

```
cc -mt  [ flag... ] file... -lpthread [ library... ]
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
     void **restrict stackaddr, size_t *restrict stacksize);

int pthread_attr_setstack(pthread_attr_t * attr, void *stackaddr,
     size_t stacksize);
```

Description The `pthread_attr_getstack()` and `pthread_attr_setstack()` functions, respectively, get and set the thread creation stack attributes *stackaddr* and *stacksize* in the *attr* object.

The stack attributes specify the area of storage to be used for the created thread's stack. The base (lowest addressable byte) of the storage is *stackaddr*, and the size of the storage is *stacksize* bytes. The *stacksize* argument must be at least {`PTHREAD_STACK_MIN`}. The *stackaddr* argument must be aligned appropriately to be used as a stack; for example, `pthread_attr_setstack()` might fail with `EINVAL` if (*stackaddr* & 0x7) is not 0. All pages within the stack described by *stackaddr* and *stacksize* are both readable and writable by the thread.

Return Values Upon successful completion, these functions return a 0; otherwise, an error number is returned to indicate the error.

The `pthread_attr_getstack()` function stores the stack attribute values in stackaddr and stacksize if successful.

Errors The `pthread_attr_setstack()` function will fail if:

`EINVAL` The value of *stacksize* is less than {`PTHREAD_STACK_MIN`}.

The `pthread_attr_setstack()` function may fail if:

`EACCES` The stack page(s) described by *stackaddr* and *stacksize* are not both readable and writable by the thread.

`EINVAL` The value of *stackaddr* does not have proper alignment to be used as a stack, or (*stackaddr* + *stacksize*) lacks proper alignment.

Usage These functions are appropriate for use by applications in an environment where the stack for a thread must be placed in some particular region of memory.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also pthread_attr_init(3C), pthread_attr_setdetachstate(3C), pthread_attr_setstacksize(3C), pthread_create(3C), attributes(5)

Name pthread_attr_getstackaddr, pthread_attr_setstackaddr – get or set stackaddr attribute

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
     void **restrict stackaddr);

int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

Description The functions `pthread_attr_setstackaddr()` and `pthread_attr_getstackaddr()`, respectively, set and get the thread creation *stackaddr* attribute in the *attr* object. The *stackaddr* default is `NULL`. See `pthread_create(3C)`.

The *stackaddr* attribute specifies the location of storage to be used for the created thread's stack. The size of the storage is at least `PTHREAD_STACK_MIN`.

Return Values Upon successful completion, `pthread_attr_setstackaddr()` and `pthread_attr_getstackaddr()` return a value of `0`. Otherwise, an error number is returned to indicate the error.

If successful, the `pthread_attr_getstackaddr()` function stores the *stackaddr* attribute value in *stackaddr*.

Errors The `pthread_attr_setstackaddr()` function may fail if:

`EINVAL` *attr* is invalid.

The `pthread_attr_getstackaddr()` function may fail if:

`EINVAL` *attr* or *stackaddr* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_attr_init(3C)`, `pthread_attr_setdetachstate(3C)`, `pthread_attr_setstacksize(3C)`, `pthread_create(3C)`, `attributes(5)`, `standards(5)`

Name pthread_attr_getstacksize, pthread_attr_setstacksize – get or set stacksize attribute

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
     size_t *restrict stacksize);

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

Description The functions `pthread_attr_setstacksize()` and `pthread_attr_getstacksize()`, respectively, set and get the thread creation *stacksize* attribute in the *attr* object.

The *stacksize* attribute defines the minimum stack size (in bytes) allocated for the created threads stack. When the *stacksize* argument is `NULL`, the default stack size becomes `1` megabyte for 32-bit processes and 2 megabytes for 64-bit processes.

Return Values Upon successful completion, `pthread_attr_setstacksize()` and `pthread_attr_getstacksize()` return a value of `0`. Otherwise, an error number is returned to indicate the error. The `pthread_attr_getstacksize()` function stores the *stacksize* attribute value in *stacksize* if successful.

Errors The `pthread_attr_setstacksize()` or `pthread_attr_getstacksize()` function may fail if:

`EINVAL` *attr* or *stacksize* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_attr_init(3C)`, `pthread_attr_setstackaddr(3C)`, `pthread_attr_setdetachstate(3C)`, `pthread_create(3C)`, `attributes(5)`, `standards(5)`

Name pthread_attr_init, pthread_attr_destroy – initialize or destroy threads attribute object

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);
```

Description The function `pthread_attr_init()` initializes a thread attributes object *attr* with the default value for all of the individual attributes used by a given implementation.

The resulting attribute object (possibly modified by setting individual attribute values), when used by `pthread_create()`, defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to `pthread_create()`.

The `pthread_attr_init()` function initializes a thread attributes object (*attr*) with the default value for each attribute as follows:

Attribute	Default Value	Meaning of Default
contentionscope	`PTHREAD_SCOPE_PROCESS`	resource competition within process
detachstate	`PTHREAD_CREATE_JOINABLE`	joinable by other threads
stackaddr	`NULL`	stack allocated by system
stacksize	0	1 or 2 megabyte
priority	0	priority of the thread
policy	`SCHED_OTHER`	traditional time-sharing policy
inheritsched	`PTHREAD_INHERIT_SCHED`	scheduling policy and parameters are inherited from the creating thread
guardsize	`PAGESIZE`	size of guard area for a thread's created stack

The `pthread_attr_destroy()` function destroys a thread attributes object (*attr*), which cannot be reused until it is reinitialized. An implementation may cause `pthread_attr_destroy()` to set *attr* to an implementation-dependent invalid value. The behavior of using the attribute after it has been destroyed is undefined.

Return Values Upon successful completion, `pthread_attr_init()` and `pthread_attr_destroy()` return a value of `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_attr_init()` function will fail if:

`ENOMEM` Insufficient memory exists to initialize the thread attributes object.

The `pthread_attr_destroy()` function may fail if:

`EINVAL` *attr* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `sysconf(3C)`, `pthread_attr_getdetachstate(3C)`, `pthread_attr_getguardsize(3C)`, `pthread_attr_getinheritsched(3C)`, `pthread_attr_getschedparam(3C)`, `pthread_attr_getschedpolicy(3C)`, `pthread_attr_getscope(3C)`, `pthread_attr_getstackaddr(3C)`, `pthread_attr_getstacksize(3C)`, `pthread_attr_setdetachstate(3C)`, `pthread_attr_setguardsize(3C)`, `pthread_attr_setinheritsched(3C)`, `pthread_attr_setschedparam(3C)`, `pthread_attr_setschedpolicy(3C)`, `pthread_attr_setscope(3C)`, `pthread_attr_setstackaddr(3C)`, `pthread_attr_setstacksize(3C)`, `pthread_create(3C)`, `attributes(5)`, `standards(5)`

Name pthread_barrierattr_destroy, pthread_barrierattr_init – destroy and initialize barrier attributes object

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);

int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

Description The `pthread_barrierattr_destroy()` function destroys a barrier attributes object. A destroyed *attr* attributes object can be reinitialized using `pthread_barrierattr_init()`. The results of otherwise referencing the object after it has been destroyed are undefined. An implementation can cause `pthread_barrierattr_destroy()` to set the object referenced by *attr* to an invalid value.

The `pthread_barrierattr_init()` function initializes a barrier attributes object *attr* with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_barrierattr_init()` is called specifying an already initialized *attr* attributes object.

After a barrier attributes object has been used to initialize one or more barriers, any function affecting the attributes object (including destruction) does not affect any previously initialized barrier.

Return Values Upon successful completion, the `pthread_barrierattr_destroy()` and `pthread_barrierattr_init()` functions returns 0. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_barrierattr_init()` function will fail if:

`ENOMEM` Insufficient memory exists to initialize the barrier attributes object.

The `pthread_barrierattr_destroy()` function may fail if:

`EINVAL` The value specified by *attr* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also pthread_barrierattr_getpshared(3C), attributes(5), standards(5)

Name pthread_barrierattr_getpshared, pthread_barrierattr_setpshared – get and set process-shared attribute of barrier attributes object

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>

int pthread_barrierattr_getpshared(
     const pthread_barrierattr_t *restrict attr,
     int *restrict pshared);

int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
     int pshared);
```

Description The `pthread_barrierattr_getpshared()` function obtains the value of the *process-shared* attribute from the attributes object referenced by *attr*. The `pthread_barrierattr_setpshared()` function sets the *process-shared* attribute in an initialized attributes object referenced by *attr*.

The *process-shared* attribute is set to `PTHREAD_PROCESS_SHARED` to permit a barrier to be operated upon by any thread that has access to the memory where the barrier is allocated. If the *process-shared* attribute is `PTHREAD_PROCESS_PRIVATE`, the barrier will only be operated upon by threads created within the same process as the thread that initialized the barrier. If threads of different processes attempt to operate on such a barrier, the behavior is undefined.

The default value of the attribute is `PTHREAD_PROCESS_PRIVATE`. Both constants `PTHREAD_PROCESS_SHARED` and `PTHREAD_PROCESS_PRIVATE` are defined in <`pthread.h`>.

No barrier attributes other than the *process-shared* attribute are provided.

Return Values Upon successful completion, the `pthread_barrierattr_getpshared()` function returns 0 and stores the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the `pthread_barrierattr_setpshared()` function returns 0. Otherwise, an error number is returned to indicate the error.

Errors These functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `pthread_barrierattr_setpshared()` function may fail if:

`EINVAL` The new value specified for the *process-shared* attribute is not one of the legal values `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also pthread_barrier_init(3C), pthread_barrierattr_destroy(3C), attributes(5), standards(5)

Name pthread_barrier_destroy, pthread_barrier_init – destroy and initialize a barrier object

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>

int pthread_barrier_destroy(pthread_barrier_t *barrier);

int pthread_barrier_init(pthread_barrier_t *barrier,
     const pthread_barrierattr_t *restrict attr, unsigned count);
```

Description The `pthread_barrier_destroy()` function destroys the barrier referenced by *barrier* and releases any resources used by the barrier. The effect of subsequent use of the barrier is undefined until the barrier is reinitialized by another call to `pthread_barrier_init()`. An implementation can use this function to set barrier to an invalid value. The results are undefined if `pthread_barrier_destroy()` is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

The `pthread_barrier_init()` function allocates any resources required to use the barrier referenced by barrier and initializes the barrier with attributes referenced by *attr*. If attr is `NULL`, the default barrier attributes are used; the effect is the same as passing the address of a default barrier attributes object. The results are undefined if `pthread_barrier_init()` is called when any thread is blocked on the barrier (that is, has not returned from the `pthread_barrier_wait(3C)` call). The results are undefined if a barrier is used without first being initialized. The results are undefined if `pthread_barrier_init()` is called specifying an already initialized barrier.

The *count* argument specifies the number of threads that must call `pthread_barrier_wait()` before any of them successfully return from the call. The value specified by *count* must be greater than 0.

If the `pthread_barrier_init()` function fails, the barrier is not initialized and the contents of *barrier* are undefined.

Only the object referenced by barrier can be used for performing synchronization. The result of referring to copies of that object in calls to `pthread_barrier_destroy()` or `pthread_barrier_wait()` is undefined.

Return Values Upon successful completion, these functions returns 0. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_barrier_init()` function will fail if:

`EAGAIN` The system lacks the necessary resources to initialize another barrier.

`EINVAL` The value specified by *count* is equal to 0.

`ENOMEM` Insufficient memory exists to initialize the barrier.

The `pthread_barrier_init()` function may fail if:

`EBUSY` The implementation has detected an attempt to destroy a barrier while it is in use (for example, while being used in a `pthread_barrier_wait()` call) by another thread.

`EINVAL` The value specified by *attr* is invalid.

The `pthread_barrier_destroy()` function may fail if:

`EBUSY` The implementation has detected an attempt to destroy a barrier while it is in use (for example, while being used in a `pthread_barrier_wait()` call) by another thread.

`EINVAL` The value specified by *barrier* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_barrier_wait(3C)`, `attributes(5)`, `standards(5)`

Name pthread_barrier_wait – synchronize at a barrier

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Description The `pthread_barrier_wait()` function synchronizes participating threads at the barrier referenced by *barrier*. The calling thread blocks until the required number of threads have called `pthread_barrier_wait()` specifying the barrier.

When the required number of threads have called `pthread_barrier_wait()` specifying the barrier, the constant `PTHREAD_BARRIER_SERIAL_THREAD` is returned to one unspecified thread and 0 is returned to each of the remaining threads. At this point, the barrier is reset to the state it had as a result of the most recent `pthread_barrier_init(3C)` function that referenced it.

The constant `PTHREAD_BARRIER_SERIAL_THREAD` is defined in `<pthread.h>` and its value is distinct from any other value returned by `pthread_barrier_wait()`.

The results are undefined if this function is called with an uninitialized barrier.

If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler the thread resumes waiting at the barrier if the barrier wait has not completed (that is, if the required number of threads have not arrived at the barrier during the execution of the signal handler); otherwise, the thread continues as normal from the completed barrier wait. Until the thread in the signal handler returns from it, it is unspecified whether other threads may proceed past the barrier once they have all reached it.

A thread that has blocked on a barrier does not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution.

Eligibility for processing resources is determined by the scheduling policy.

Return Values Upon successful completion, the `pthread_barrier_wait()` function returns `PTHREAD_BARRIER_SERIAL_THREAD` for a single (arbitrary) thread synchronized at the barrier and 0 for each of the other threads. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_barrier_wait()` function will fail if:

`EINVAL` The value specified by *barrier* does not refer to an initialized barrier object.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also pthread_barrier_destroy(3C), attributes(5), standards(5)

Name pthread_cancel – cancel execution of a thread

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_cancel(pthread_t target_thread);
```

Description The `pthread_cancel()` function requests that *target_thread* be canceled.

By default, cancellation is deferred until *target_thread* reaches a cancellation point. See `cancellation(5)`.

Cancellation cleanup handlers for *target_thread* are called when the cancellation is acted on. Upon return of the last cancellation cleanup handler, the thread-specific data destructor functions are called for *target_thread*. *target_thread* is terminated when the last destructor function returns.

A thread acting on a cancellation request runs with all signals blocked. All thread termination functions, including cancellation cleanup handlers and thread-specific data destructor functions, are called with all signals blocked.

The cancellation processing in *target_thread* runs asynchronously with respect to the calling thread returning from `pthread_cancel()`.

Return Values If successful, the `pthread_cancel()` function returns `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_cancel()` function may fail if:

`ESRCH` No thread was found with an ID corresponding to that specified by the given thread ID, *target_thread*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_cleanup_pop(3C)`, `pthread_cleanup_push(3C)`, `pthread_cond_wait(3C)`, `pthread_cond_timedwait(3C)`, `pthread_exit(3C)`, `pthread_join(3C)`, `pthread_setcancelstate(3C)`, `pthread_setcanceltype(3C)`, `pthread_testcancel(3C)`, `setjmp(3C)`, `attributes(5)`, `cancellation(5)`, `condition(5)`, `standards(5)`

Notes See cancellation(5) for a discussion of cancellation concepts.

Name pthread_cleanup_pop – pop a thread cancellation cleanup handler

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

void pthread_cleanup_pop(int execute);
```

Description The `pthread_cleanup_pop()` function removes the cleanup handler routine at the top of the cancellation cleanup stack of the calling thread and executes it if *execute* is non-zero.

When the thread calls `pthread_cleanup_pop()` with a non-zero *execute* argument, the argument at the top of the stack is popped and executed. An argument of `0` pops the handler without executing it.

The `pthread_cleanup_push(3C)` and `pthread_cleanup_pop()` functions can be implemented as macros. The application must ensure that they appear as statements, and in pairs within the same lexical scope (that is, the `pthread_cleanup_push()` macro can be thought to expand to a token list whose first token is '{' with `pthread_cleanup_pop()` expanding to a token list whose last token is the corresponding '}').

The effect of the use of `return`, `break`, `continue`, and `goto` to prematurely leave a code block described by a pair of `pthread_cleanup_push()` and `pthread_cleanup_pop()` function calls is undefined.

Using `longjmp()` or `siglongjmp()` to jump into or out of a push/pop pair can result in either the matching push or the matching pop statement not getting executed.

Return Values The `pthread_cleanup_pop()` function returns no value.

Errors No errors are defined.

The `pthread_cleanup_pop()` function will not return an error code of `EINTR`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_cancel(3C)`, `pthread_cleanup_push(3C)`, `pthread_exit(3C)`, `pthread_join(3C)`, `pthread_setcancelstate(3C)`, `pthread_setcanceltype(3C)`, `pthread_testcancel(3C)`, `setjmp(3C)`, `attributes(5)`, `cancellation(5)`, `condition(5)`, `standards(5)`

Notes See cancellation(5) for a discussion of cancellation concepts.

Name pthread_cleanup_push – push a thread cancellation cleanup handler

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

void pthread_cleanup_push(void (*handler) (void *), void *arg);
```

Description The `pthread_cleanup_push()` function pushes the specified cancellation cleanup handler routine, *handler*, onto the cancellation cleanup stack of the calling thread.

When a thread exits or is canceled and its cancellation cleanup stack is not empty, the cleanup handlers are invoked with the argument *arg* in last in, first out (LIFO) order from the cancellation cleanup stack.

An exiting or cancelled thread runs with all signals blocked. All thread termination functions, including cancellation cleanup handlers, are called with all signals blocked.

The `pthread_cleanup_push()` and `pthread_cleanup_pop(3C)` functions can be implemented as macros. The application must ensure that they appear as statements, and in pairs within the same lexical scope (that is, the `pthread_cleanup_push()` macro can be thought to expand to a token list whose first token is '{' with `pthread_cleanup_pop()` expanding to a token list whose last token is the corresponding '}').

The effect of the use of `return`, `break`, `continue`, and `goto` to prematurely leave a code block described by a pair of `pthread_cleanup_push()` and `pthread_cleanup_pop()` function calls is undefined.

Using `longjmp()` or `siglongjmp()` to jump into or out of a push/pop pair can cause either the matching push or the matching pop statement not getting executed.

Return Values The `pthread_cleanup_push()` function returns no value.

Errors No errors are defined.

The `pthread_cleanup_push()` function will not return an error code of `EINTR`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also longjmp(3C), pthread_cancel(3C), pthread_cleanup_pop(3C), pthread_exit(3C), pthread_join(3C), pthread_setcancelstate(3C), pthread_setcanceltype(3C), pthread_testcancel(3C), attributes(5), cancellation(5), condition(5), standards(5)

Notes See cancellation(5) for a discussion of cancellation concepts.

Name pthread_condattr_getclock, pthread_condattr_setclock – get and set the clock selection condition variable attribute

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>

int pthread_condattr_getclock(
     const pthread_condattr_t *restrict attr,
     clockid_t *restrict clock_id);

int pthread_condattr_setclock(pthread_condattr_t *attr
      clockid_t clock_id);
```

Description The `pthread_condattr_getclock()` function obtains the value of the clock attribute from the attributes object referenced by *attr*. The `pthread_condattr_setclock()` function sets the clock attribute in an initialized attributes object referenced by *attr*. If `pthread_condattr_setclock()` is called with a *clock_id* argument that refers to a CPU-time clock, the call fails.

The clock attribute is the clock ID of the clock that is used to measure the timeout service of `pthread_cond_timedwait(3C)`. The default value of the clock attribute refers to the system clock.

Return Values Upon successful completion, the `pthread_condattr_getclock()` function returns 0 and stores the value of the clock attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the `pthread_condattr_setclock()` function returns 0. Otherwise, an error number is returned to indicate the error.

Errors These functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `pthread_condattr_setclock()` function may fail if:

`EINVAL` The value specified by *clock_id* does not refer to a known clock, or is a CPU-time clock.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also pthread_cond_init(3C), pthread_cond_timedwait(3C), pthread_condattr_destroy(3C), pthread_condattr_getpshared(3C), pthread_create(3C), pthread_mutex_init(3C), attributes(5), standards(5)

Name pthread_condattr_getpshared, pthread_condattr_setpshared – get or set process-shared condition variable attributes

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_condattr_getpshared(
     const pthread_condattr_t *restrict attr,
     int *restrict pshared);

int pthread_condattr_setpshared(pthread_condattr_t *attr,
     int pshared);
```

Description The `pthread_condattr_getpshared()` function obtains the value of the *process-shared* attribute from the attributes object referenced by *attr*. The `pthread_condattr_setpshared()` function is used to set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

The *process-shared* attribute is set to `PTHREAD_PROCESS_SHARED` to permit a condition variable to be operated upon by any thread that has access to the memory where the condition variable is allocated, even if the condition variable is allocated in memory that is shared by multiple processes. If the *process-shared* attribute is `PTHREAD_PROCESS_PRIVATE`, the condition variable will only be operated upon by threads created within the same process as the thread that initialized the condition variable; if threads of differing processes attempt to operate on such a condition variable, the behavior is undefined. The default value of the attribute is `PTHREAD_PROCESS_PRIVATE`.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-dependent.

Return Values If successful, the `pthread_condattr_setpshared()` function returns `0`. Otherwise, an error number is returned to indicate the error.

If successful, the `pthread_condattr_getpshared()` function returns `0` and stores the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_condattr_getpshared()` and `pthread_condattr_setpshared()` functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `pthread_condattr_setpshared()` function will fail if:

`EINVAL` The new value specified for the attribute is outside the range of legal values for that attribute.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also pthread_condattr_init(3C), pthread_create(3C), pthread_mutex_init(3C), pthread_cond_init(3C), attributes(5), standards(5)

Name pthread_condattr_init, pthread_condattr_destroy – initialize or destroy condition variable attributes object

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);
```

Description The `pthread_condattr_init()` function initializes a condition variable attributes object *attr* with the default value for all of the attributes defined by the implementation.

At present, the only attribute available is the scope of condition variables. The default scope of the attribute is `PTHREAD_PROCESS_PRIVATE`.

Attempts to initialize previously initialized condition variable attributes object will leave the storage allocated by the previous initialization unallocated.

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialized condition variables.

The `pthread_condattr_destroy()` function destroys a condition variable attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_condattr_destroy()` to set the object referenced by *attr* to an invalid value. A destroyed condition variable attributes object can be re-initialized using `pthread_condattr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-dependent.

Return Values If successful, the `pthread_condattr_init()` and `pthread_condattr_destroy()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_condattr_init()` function will fail if:

`ENOMEM` Insufficient memory exists to initialize the condition variable attributes object.

The `pthread_condattr_destroy()` function may fail if:

`EINVAL` The value specified by *attr* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Standard	See standards(5).

See Also pthread_condattr_getpshared(3C), pthread_condattr_setpshared(3C), pthread_cond_init(3C), pthread_create(3C), pthread_mutex_init(3C), attributes(5), standards(5)

Name pthread_cond_init, pthread_cond_destroy – initialize or destroy condition variables

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
     const pthread_condattr_t *restrict attr);

int pthread_cond_destroy(pthread_cond_t *cond

pthread_cond_t cond= PTHREAD_COND_INITIALIZER;
```

Description The function `pthread_cond_init()` initializes the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes are used; the effect is the same as passing the address of a default condition variable attributes object. See `pthread_condattr_init(3C)`. Upon successful initialization, the state of the condition variable becomes initialized.

Attempting to initialize an already initialized condition variable results in undefined behavior.

The function `pthread_cond_destroy()` destroys the given condition variable specified by *cond*; the object becomes, in effect, uninitialized. An implementation may cause `pthread_cond_destroy()` to set the object referenced by *cond* to an invalid value. A destroyed condition variable object can be re-initialized using `pthread_cond_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro `PTHREAD_COND_INITIALIZER` can be used to initialize condition variables that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_cond_init()` with parameter *attr* specified as NULL, except that no error checks are performed.

Return Values If successful, the `pthread_cond_init()` and `pthread_cond_destroy()` functions return `0`. Otherwise, an error number is returned to indicate the error. The `EBUSY` and `EINVAL` error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the condition variable specified by *cond*.

Errors The `pthread_cond_init()` function will fail if:

`EAGAIN` The system lacked the necessary resources (other than memory) to initialize another condition variable.

`ENOMEM` Insufficient memory exists to initialize the condition variable.

The `pthread_cond_init()` function may fail if:

`EBUSY` The implementation has detected an attempt to re-initialize the object referenced by *cond*, a previously initialized, but not yet destroyed, condition variable.

`EINVAL` The value specified by *attr* is invalid.

The `pthread_cond_destroy()` function may fail if:

`EBUSY` The implementation has detected an attempt to destroy the object referenced by *cond* while it is referenced (for example, while being used in a `pthread_cond_wait()` or `pthread_cond_timedwait()`) by another thread.

`EINVAL` The value specified by *cond* is invalid.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also

`pthread_cond_signal(3C)`, `pthread_cond_broadcast(3C)`, `pthread_cond_wait(3C)`, `pthread_cond_timedwait(3C)`, `pthread_condattr_init(3C)`, `attributes(5)`, `condition(5)`, `standards(5)`

Name pthread_cond_signal, pthread_cond_broadcast – signal or broadcast a condition

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

Description These two functions are used to unblock threads blocked on a condition variable.

The `pthread_cond_signal()` call unblocks at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).

The `pthread_cond_broadcast()` call unblocks all threads currently blocked on the specified condition variable *cond*.

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a `pthread_cond_signal()` or `pthread_cond_broadcast()` returns from its call to `pthread_cond_wait()` or `pthread_cond_timedwait()`, the thread owns the mutex with which it called `pthread_cond_wait()` or `pthread_cond_timedwait()`. The thread(s) that are unblocked contend for the mutex according to the scheduling policy (if applicable), and as if each had called `pthread_mutex_lock()`.

The `pthread_cond_signal()` or `pthread_cond_broadcast()` functions may be called by a thread whether or not it currently owns the mutex that threads calling `pthread_cond_wait()` or `pthread_cond_timedwait()` have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex is locked by the thread calling `pthread_cond_signal()` or `pthread_cond_broadcast()`.

The `pthread_cond_signal()` and `pthread_cond_broadcast()` functions have no effect if there are no threads currently blocked on *cond*.

Return Values If successful, the `pthread_cond_signal()` and `pthread_cond_broadcast()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_cond_signal()` and `pthread_cond_broadcast()` function may fail if:

`EINVAL` The value *cond* does not refer to an initialized condition variable.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also pthread_cond_init(3C), pthread_cond_wait(3C), pthread_cond_timedwait(3C), attributes(5), condition(5), standards(5)

Name pthread_cond_wait, pthread_cond_timedwait, pthread_cond_reltimedwait_np – wait on a condition

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

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

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

int pthread_cond_reltimedwait_np(pthread_cond_t *cond,
     pthread_mutex_t *mutex, const struct timespec *reltime);
```

Description The `pthread_cond_wait()`, `pthread_cond_timedwait()`, and `pthread_cond_reltimedwait_np()` functions are used to block on a condition variable. They are called with *mutex* locked by the calling thread or undefined behavior will result.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*. Atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable." That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to `pthread_cond_signal()` or `pthread_cond_broadcast()` in that thread behaves as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex has been locked and is owned by the calling thread. If mutex is a robust mutex where an owner terminated while holding the lock and the state is recoverable, the mutex is acquired even though the function returns an error value.

When using condition variables there is always a boolean predicate, an invariant, associated with each condition wait that must be true before the thread should proceed. Spurious wakeups from the `pthread_cond_wait()`, `pthread_cond_timedwait()`, or `pthread_cond_reltimedwait_np()` functions could occur. Since the return from `pthread_cond_wait()`, `pthread_cond_timedwait()`, or `pthread_cond_reltimedwait_np()` does not imply anything about the value of this predicate, the predicate should always be reevaluated.

The order in which blocked threads are awakened by `pthread_cond_signal()` or `pthread_cond_broadcast()` is determined by the scheduling policy. See `pthreads(5)`.

The effect of using more than one mutex for concurrent `pthread_cond_wait()`, `pthread_cond_timedwait()`, or `pthread_cond_reltimedwait_np()` operations on the same condition variable will result in undefined behavior.

A condition wait (whether timed or not) is a cancellation point. When the cancelability enable state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is reacquired before calling the first cancellation cleanup handler.

A thread that has been unblocked because it has been canceled while blocked in a call to pthread_cond_wait() or pthread_cond_timedwait() does not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The pthread_cond_timedwait() function is the same as pthread_cond_wait() except that an error is returned if the absolute time specified by *abstime* passes (that is, system time equals or exceeds *abstime*) before the condition *cond* is signaled or broadcast, or if the absolute time specified by *abstime* has already been passed at the time of the call. The *abstime* argument is of type struct timespec, defined in time.h(3HEAD). When such time-outs occur, pthread_cond_timedwait() will nonetheless release and reacquire the mutex referenced by *mutex*. The function pthread_cond_timedwait() is also a cancellation point.

The pthread_cond_reltimedwait_np() function is a non-standard extension provided by the Solaris version of POSIX threads as indicated by the "_np" (non-portable) suffix. The pthread_cond_reltimedwait_np() function is the same as pthread_cond_timedwait() except that the *reltime* argument specifies a non-negative time relative to the current system time rather than an absolute time. The *reltime* argument is of type struct timespec, defined in time.h(3HEAD). An error value is returned if the relative time passes (that is, system time equals or exceeds the starting system time plus the relative time) before the condition *cond* is signaled or broadcast. When such timeouts occur, pthread_cond_reltimedwait_np() releases and reacquires the mutex referenced by *mutex*. The pthread_cond_reltimedwait_np() function is also a cancellation point.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it returns 0 due to spurious wakeup.

Return Values Except in the case of ETIMEDOUT, EOWNERDEAD, or ENOTRECOVERABLE, all of these error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return, in effect, prior to modifying the state of the mutex specified by *mutex* or the condition variable specified by *cond*.

Upon successful completion, 0 is returned. Otherwise, an error value is returned to indicate the error.

Errors These functions will fail if:

EPERM The mutex type is PTHREAD_MUTEX_ERRORCHECK or the mutex is a robust mutex, and the current thread does not own the mutex.

The pthread_cond_timedwait() function will fail if:

`ETIMEDOUT` The absolute time specified by *abstime* to `pthread_cond_timedwait()` has passed.

The `pthread_cond_reltimedwait_np()` function will fail if:

`EINVAL` The value specified by *reltime* is invalid.

`ETIMEDOUT` The relative time specified by *reltime* to `pthread_cond_reltimedwait_np()` has passed.

These functions may fail if:

`EINVAL` The value specified by *cond*, *mutex*, *abstime*, or *reltime* is invalid.

`EINVAL` Different mutexes were supplied for concurrent operations on the same condition variable.

If the mutex specified by *mutex* is a robust mutex (initialized with the robustness attribute `PTHREAD_MUTEX_ROBUST`), the `pthread_cond_wait()`, `pthread_cond_timedwait()`, and `pthread_cond_reltimedwait_np()` functions will, under the specified conditions, return the following error values. For complete information, see the `pthread_mutex_lock(3C)` and `pthread_mutexattr_setrobust(3C)` manual pages.

`EOWNERDEAD` The last owner of this mutex died while holding the mutex, leaving the state it was protecting possibly inconsistent. The mutex is now owned by the caller.

`ENOTRECOVERABLE` The mutex was protecting state that has now been left irrecoverable. The mutex has not been acquired.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also

`pthread_cond_signal(3C)`, `pthread_cond_broadcast(3C)`, `pthread_mutex_lock(3C)`, `pthread_mutexattr_getrobust(3C)`, `time.h(3HEAD)`, `attributes(5)`, `condition(5)`, `pthreads(5)`, `standards(5)`

Name pthread_create – create a thread

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
     const pthread_attr_t *restrict attr,
     void *(*start_routine)(void*), void *restrict arg);
```

Description The `pthread_create()` function is used to create a new thread, with attributes specified by *attr*, within a process. If *attr* is `NULL`, the default attributes are used. (See `pthread_attr_init(3C)`). If the attributes specified by *attr* are modified later, the thread's attributes are not affected. Upon successful completion, `pthread_create()` stores the ID of the created thread in the location referenced by *thread*.

The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect is as if there was an implicit call to `pthread_exit()` using the return value of *start_routine* as the exit status. Note that the thread in which `main()` was originally invoked differs from this. When it returns from `main()`, the effect is as if there was an implicit call to `exit()` using the return value of `main()` as the exit status.

The signal state of the new thread is initialised as follows:

- The signal mask is inherited from the creating thread.
- The set of signals pending for the new thread is empty.

Default thread creation:

```
pthread_t tid;
void *start_func(void *), *arg;

pthread_create(&tid, NULL, start_func, arg);
```

This would have the same effect as:

```
pthread_attr_t attr;

pthread_attr_init(&attr); /* initialize attr with default */
                          /* attributes */
pthread_create(&tid, &attr, start_func, arg);
```

User-defined thread creation: To create a thread that is scheduled on a system-wide basis, use:

```
pthread_attr_init(&attr); /* initialize attr with default */
                                    /* attributes */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
                                    /* system-wide contention */
pthread_create(&tid, &attr, start_func, arg);
```

To customize the attributes for POSIX threads, see `pthread_attr_init(3C)`.

A new thread created with `pthread_create()` uses the stack specified by the *stackaddr* attribute, and the stack continues for the number of bytes specified by the *stacksize* attribute. By default, the stack size is 1 megabyte for 32-bit processes and 2 megabyte for 64-bit processes (see `pthread_attr_setstacksize(3C)`). If the default is used for both the *stackaddr* and *stacksize* attributes, `pthread_create()` creates a stack for the new thread with at least 1 megabyte for 32-bit processes and 2 megabyte for 64-bit processes. (For customizing stack sizes, see `NOTES`).

If `pthread_create()` fails, no new thread is created and the contents of the location referenced by *thread* are undefined.

Return Values If successful, the `pthread_create()` function returns `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_create()` function will fail if:

`EAGAIN` The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process `PTHREAD_THREADS_MAX` would be exceeded.

`EINVAL` The value specified by *attr* is invalid.

`EPERM` The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy.

Examples **EXAMPLE 1** Example of concurrency with multithreading

The following is an example of concurrency with multithreading. Since POSIX threads and Solaris threads are fully compatible even within the same process, this example uses `pthread_create()` if you execute `a.out 0`, or `thr_create()` if you execute `a.out 1`.

Five threads are created that simultaneously perform a time-consuming function, `sleep(10)`. If the execution of this process is timed, the results will show that all five individual calls to sleep for ten-seconds completed in about ten seconds, even on a uniprocessor. If a single-threaded process calls `sleep(10)` five times, the execution time will be about 50-seconds.

The command-line to time this process is:

POSIX threading /usr/bin/time a.out 0

Solaris threading /usr/bin/time a.out 1

```
/* cc thisfile.c -lthread -lpthread */
#define _REENTRANT    /* basic 3-lines for threads */
#include <pthread.h>
#include <thread.h>

#define NUM_THREADS 5
#define SLEEP_TIME 10
```

EXAMPLE 1 Example of concurrency with multithreading *(Continued)*

```
void *sleeping(void *);   /* thread routine */
int i;
thread_t tid[NUM_THREADS];      /* array of thread IDs */

int
main(int argc, char *argv[])
{
    if (argc == 1)  {
        printf("use 0 as arg1 to use pthread_create( )\n");
        printf("or use 1 as arg1 to use thr_create( )\n");
        return (1);
    }

    switch (*argv[1])  {
    case '0':  /* POSIX */
        for ( i = 0; i < NUM_THREADS; i++)
            pthread_create(&tid[i], NULL, sleeping,
                (void *)SLEEP_TIME);
        for ( i = 0; i < NUM_THREADS; i++)
            pthread_join(tid[i], NULL);
        break;

    case '1':  /* Solaris */
        for ( i = 0; i < NUM_THREADS; i++)
            thr_create(NULL, 0, sleeping, (void *)SLEEP_TIME, 0,
                &tid[i]);
        while (thr_join(0, NULL, NULL) == 0)
            ;
        break;
    }  /* switch */
    printf("main( ) reporting that all %d threads have
        terminated\n", i);
    return (0);
}  /* main */

void *
sleeping(void *arg)
{
    int sleep_time = (int)arg;
    printf("thread %d sleeping %d seconds ...\n", thr_self( ),
        sleep_time);
    sleep(sleep_time);
    printf("\nthread %d awakening\n", thr_self( ));
```

EXAMPLE 1 Example of concurrency with multithreading *(Continued)*

```
    return (NULL);
}
```

If `main()` had not waited for the completion of the other threads (using `pthread_join(3C)` or `thr_join(3C)`), it would have continued to process concurrently until it reached the end of its routine and the entire process would have exited prematurely. See `exit(2)`.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also

`fork(2)`, `pthread_attr_init(3C)`, `pthread_cancel(3C)`, `pthread_exit(3C)`, `pthread_join(3C)`, `sysconf(3C)`, `attributes(5)`, `standards(5)`

Notes

Multithreaded application threads execute independently of each other, so their relative behavior is unpredictable. Therefore, it is possible for the thread executing `main()` to finish before all other user application threads. The `pthread_join(3C)`function, on the other hand, must specify the terminating thread (IDs) for which it will wait.

A user-specified stack size must be greater than the value `PTHREAD_STACK_MIN`. A minimum stack size may not accommodate the stack frame for the user thread function *start_func*. If a stack size is specified, it must accommodate *start_func* requirements and the functions that it may call in turn, in addition to the minimum requirement.

It is usually very difficult to determine the runtime stack requirements for a thread. `PTHREAD_STACK_MIN` specifies how much stack storage is required to execute a `NULL` *start_func*. The total runtime requirements for stack storage are dependent on the storage required to do runtime linking, the amount of storage required by library runtimes (as `printf()`) that your thread calls. Since these storage parameters are not known before the program runs, it is best to use default stacks. If you know your runtime requirements or decide to use stacks that are larger than the default, then it makes sense to specify your own stacks.

Name pthread_detach – detach a thread

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Description The `pthread_detach()` function is used to indicate to the implementation that storage for the thread *thread* can be reclaimed when that thread terminates. In other words, `pthread_detach()` dynamically resets the *detachstate* attribute of the thread to `PTHREAD_CREATE_DETACHED`. After a successful call to this function, it would not be necessary to reclaim the thread using `pthread_join()`. See `pthread_join(3C)`. If *thread* has not terminated, `pthread_detach()` will not cause it to terminate. The effect of multiple `pthread_detach()` calls on the same target thread is unspecified.

Return Values If successful, `pthread_detach()` returns `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_detach()` function will fail if:

`EINVAL` The implementation has detected that the value specified by *thread* does not refer to a joinable thread.

`ESRCH` No thread could be found corresponding to that specified by the given thread ID.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_create(3C)`, `pthread_join(3C)`, `attributes(5)`, `standards(5)`

Name pthread_equal – compare thread IDs

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Description The `pthread_equal()` function compares the thread IDs *t1* and *t2*.

Return Values The `pthread_equal()` function returns a non-zero value if *t1* and *t2* are equal. Otherwise, `0` is returned.

If *t1* or *t2* is an invalid thread ID, the behavior is undefined.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_create(3C)`, `pthread_self(3C)`, `attributes(5)`, `standards(5)`

Notes Solaris thread IDs do not require an equivalent function because the `thread_t` structure is an unsigned int.

Name pthread_exit – terminate calling thread

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

Description The `pthread_exit()` function terminates the calling thread, in a similar way that `exit(3C)` terminates the calling process. If the thread is not detached, the exit status specified by *value_ptr* is made available to any successful join with the terminating thread. See `pthread_join(3C)`. Any cancellation cleanup handlers that have been pushed and not yet popped are popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions will be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process level cleanup actions, including, but not limited to, calling any `atexit()` routines that might exist.

An exiting thread runs with all signals blocked. All thread termination functions, including cancellation cleanup handlers and thread-specific data destructor functions, are called with all signals blocked.

An implicit call to `pthread_exit()` is made when a thread other than the thread in which `main()` was first invoked returns from the start routine that was used to create it. The function's return value serves as the thread's exit status.

The behavior of `pthread_exit()` is undefined if called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to `pthread_exit()`.

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the `pthread_exit()` *value_ptr* parameter value.

The process exits with an exit status of `0` after the last thread has been terminated. The behavior is as if the implementation called `exit()` with a `0` argument at thread termination time.

Return Values The `pthread_exit()` function cannot return to its caller.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Standard	See standards(5).

See Also exit(3C), pthread_cancel(3C), pthread_create(3C), pthread_join(3C), pthread_key_create(3C), attributes(5), standards(5)

Name pthread_getconcurrency, pthread_setconcurrency – get or set level of concurrency

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_getconcurrency(void);

int pthread_setconcurrency(int new_level);
```

Description Unbound threads in a process may or may not be required to be simultaneously active. By default, the threads implementation ensures that a sufficient number of threads are active so that the process can continue to make progress. While this conserves system resources, it may not produce the most effective level of concurrency.

The `pthread_setconcurrency()` function allows an application to inform the threads implementation of its desired concurrency level, *new_level*. The actual level of concurrency provided by the implementation as a result of this function call is unspecified.

If *new_level* is `0`, it causes the implementation to maintain the concurrency level at its discretion as if `pthread_setconcurrency()` was never called.

The `pthread_getconcurrency()` function returns the value set by a previous call to the `pthread_setconcurrency()` function. If the `pthread_setconcurrency()` function was not previously called, this function returns `0` to indicate that the implementation is maintaining the concurrency level.

When an application calls `pthread_setconcurrency()` it is informing the implementation of its desired concurrency level. The implementation uses this as a hint, not a requirement.

If an implementation does not support multiplexing of user threads on top of several kernel scheduled entities, the `pthread_setconcurrency()` and `pthread_getconcurrency()` functions will be provided for source code compatibility but they will have no effect when called. To maintain the function semantics, the *new_level* parameter will be saved when `pthread_setconcurrency()` is called so that a subsequent call to `pthread_getconcurrency()` returns the same value.

Return Values If successful, the `pthread_setconcurrency()` function returns `0`. Otherwise, an error number is returned to indicate the error.

The `pthread_getconcurrency()` function always returns the concurrency level set by a previous call to `pthread_setconcurrency()`. If the `pthread_setconcurrency()` function has never been called, `pthread_getconcurrency()` returns `0`.

Errors The `pthread_setconcurrency()` function will fail if:

`EINVAL` The value specified by *new_level* is negative.

`EAGAIN` The value specific by *new_level* would cause a system resource to be exceeded.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also pthread_create(3C), pthread_attr_init(3C), attributes(5), standards(5)

Name pthread_getschedparam, pthread_setschedparam – access dynamic thread scheduling parameters

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_getschedparam(pthread_t thread, int *restrict policy,
     struct sched_param *restrict param);

int pthread_setschedparam(pthread_t thread, int policy,
     const struct sched_param *param);
```

Description The `pthread_getschedparam()` and `pthread_setschedparam()` functions allow the scheduling policy and scheduling parameters of individual threads within a multithreaded process to be retrieved and set. Supported policies are :

`SCHED_OTHER`	traditional time-sharing scheduling class
`SCHED_FIFO`	real-time class: run to completion
`SCHED_RR`	real-time class: round-robin
`SCHED_IA`	interactive time-sharing class
`SCHED_FSS`	fair-share scheduling class
`SCHED_FX`	fixed priority scheduling class

See `pthreads(5)`. The affected scheduling parameter is the *sched_priority* member of the `sched_param` structure.

The `pthread_getschedparam()` function retrieves the scheduling policy and scheduling parameters for the thread whose thread ID is given by *thread* and stores those values in *policy* and *param*, respectively. The priority value returned from `pthread_getschedparam()` is the value specified by the most recent `pthread_setschedparam()` or `pthread_create()` call affecting the target thread, and does not reflect any temporary adjustments to its priority as a result of any priority inheritance or ceiling functions. The `pthread_setschedparam()` function sets the scheduling policy and associated scheduling parameters for the thread whose thread ID is given by *thread* to the policy and associated parameters provided in *policy* and *param*, respectively.

If the `pthread_setschedparam()` function fails, no scheduling parameters will be changed for the target thread.

Return Values If successful, the `pthread_getschedparam()` and `pthread_setschedparam()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_getschedparam()` and `pthread_gsetschedparam()` functions will fail if:

`ESRCH` The value specified by *thread* does not refer to an existing thread.

The `pthread_setschedparam()` function will fail if:

`EINVAL` The value specified by *policy* or one of the scheduling parameters associated with the scheduling policy *policy* is invalid.

`EPERM` The caller does not have the appropriate permission to set either the scheduling parameters or the scheduling policy of the specified thread.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also

`pthread_attr_init(3C)`, `sched_getparam(3C)`, `sched_get_priority_max(3C)sched_get_priority_max(3C)`, `sched_get_priority_min(3C)`, `sched_setparam(3C)`, `sched_getscheduler(3C)`, `sched_setscheduler(3C)`, `attributes(5)`, `pthreads(5)`, `standards(5)`

Name pthread_getspecific, pthread_setspecific – manage thread-specific data

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

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

void *pthread_getspecific(pthread_key_t key);
```

Description The `pthread_setspecific()` function associates a thread-specific *value* with a *key* obtained by way of a previous call to `pthread_key_create()`. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The `pthread_getspecific()` function returns the value currently bound to the specified *key* on behalf of the calling thread.

The effect of calling `pthread_setspecific()` or `pthread_getspecific()` with a *key* value not obtained from `pthread_key_create()` or after *key* has been deleted with `pthread_key_delete()` is undefined.

Both `pthread_setspecific()` and `pthread_getspecific()` may be called from a thread-specific data destructor function. However, calling `pthread_setspecific()` from a destructor may result in lost storage or infinite loops.

Return Values The `pthread_getspecific()` function returns the thread-specific data value associated with the given *key*. If no thread-specific data value is associated with *key*, then the value `NULL` is returned.

Upon successful completion, the `pthread_setspecific()` function returns `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_setspecific()` function will fail if:

`ENOMEM` Insufficient memory exists to associate the value with the key.

The `pthread_setspecific()` function may fail if:

`EINVAL` The key value is invalid.

The `pthread_getspecific()` function does not return errors.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also pthread_key_create(3C), attributes(5), standards(5)

Name pthread_join – wait for thread termination

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

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

Description The `pthread_join()` function suspends processing of the calling thread until the target *thread* completes. *thread* must be a member of the current process and it cannot be a detached thread. See `pthread_create(3C)`.

If two or more threads wait for the same thread to complete, all will suspend processing until the thread has terminated, and then one thread will return successfully and the others will return with an error of `ESRCH`. The `pthread_join()` function will not block processing of the calling thread if the target *thread* has already terminated.

If a `pthread_join()` call returns successfully with a non-null *status* argument, the value passed to `pthread_exit(3C)` by the terminating thread will be placed in the location referenced by *status*.

If the `pthread_join()` calling thread is cancelled, then the target *thread* will remain joinable by `pthread_join()`. However, the calling thread may set up a cancellation cleanup handler on *thread* prior to the join call, which may detach the target *thread* by calling `pthread_detach(3C)`. See `pthread_detach(3C)` and `pthread_cancel(3C)`.

Return Values If successful, `pthread_join()` returns `0`. Otherwise, an error number is returned to indicate the error.

Errors

`EDEADLK` A joining deadlock would occur, such as when a thread attempts to wait for itself.

`EINVAL` The thread corresponding to the given thread ID is a detached thread.

`ESRCH` No thread could be found corresponding to the given thread ID.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_cancel(3C)`, `pthread_create(3C)`, `pthread_detach(3C)`, `pthread_exit(3C)`, `wait(3C)`, `attributes(5)`, `standards(5)`

Notes The `pthread_join(3C)` function must specify the *thread* ID for whose termination it will wait.

Calling `pthread_join()` also "detaches" the thread; that is, `pthread_join()` includes the effect of the `pthread_detach()` function. If a thread were to be cancelled when blocked in `pthread_join()`, an explicit detach would have to be performed in the cancellation cleanup handler. The `pthread_detach()` function exists primarily for this purpose.

Name pthread_key_create, pthread_key_create_once_np – create thread-specific data key

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_key_create(pthread_key_t *key,
     void (*destructor)(void*));

int pthread_key_create_once_np(pthread_key_t *key,
     void (*destructor)(void*));
```

Description The `pthread_key_create()` function creates a thread-specific data key visible to all threads in the process. Key values provided by `pthread_key_create()` are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by `pthread_setspecific()` are maintained on a per-thread basis and persist for the life of the calling thread.

Upon key creation, the value `NULL` is associated with the new key in all active threads. Upon thread creation, the value `NULL` is associated with all defined keys in the new thread.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-`NULL` destructor pointer, and the thread has a non-`NULL` value associated with that key, the function pointed to is called with the current associated value as its sole argument. Destructors can be called in any order.

If, after all the destructors have been called for all keys with non-`NULL` values, there are still some keys with non-`NULL` values, the process will be repeated. If, after at least `PTHREAD_DESTRUCTOR_ITERATIONS` iterations of destructor calls for outstanding non-`NULL` values, there are still some keys with non-`NULL` values, the process is continued, even though this might result in an infinite loop.

An exiting thread runs with all signals blocked. All thread termination functions, including thread-specific data destructor functions, are called with all signals blocked.

The `pthread_key_create_once_np()` function is identical to the `pthread_key_create()` function except that the key referred to by **key* must be statically initialized with the value `PTHREAD_ONCE_KEY_NP` before calling `pthread_key_create_once_np()`, and the key is created exactly once. This function call is equivalent to using `pthread_once(3C)` to call a onetime initialization function that calls `pthread_key_create()` to create the data key.

Return Values If successful, the `pthread_key_create()` and `pthread_key_create_once_np()` functions store the newly created key value at **key* and return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_key_create()` and `pthread_key_create_once_np()` functions will fail if:

`EAGAIN` The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process `PTHREAD_KEYS_MAX` has been exceeded.

ENOMEM Insufficient memory exists to create the key.

The pthread_key_create() and pthread_key_create_once_np() functions will not return an error value of EINTR.

Examples

EXAMPLE 1 Call thread-specific data in the function from more than one thread without special initialization.

In the following example, the thread-specific data in the function can be called from more than one thread without special initialization. For each argument passed to the executable, a thread is created and privately bound to the string-value of that argument.

```
/* cc -mt thisfile.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

static void *thread_function(void *);
static void show_tsd(void);
static void cleanup(void*);

#define MAX_THREADS 20

static pthread_key_t tsd_key = PTHREAD_ONCE_KEY_NP;

int
main(int argc, char *argv[])
{
     pthread_t tid[MAX_THREADS];
     int num_threads;
     int i;

     if ((num_threads = argc - 1) > MAX_THREADS)
          num_threads = MAX_THREADS;
     for (i = 0; i < num_threads; i++)
          pthread_create(&tid[i], NULL, thread_function, argv[i+1]);
     for (i = 0; i < num_threads; i++)
          pthread_join(tid[i], NULL);
     return (0);
}

static void *
thread_function(void *arg)
{
     char *data;
```

EXAMPLE 1 Call thread-specific data in the function from more than one thread without special initialization. *(Continued)*

```
     pthread_key_create_once_np(&tsd_key, cleanup);
     data = malloc(strlen(arg) + 1);
     strcpy(data, arg);
     pthread_setspecific(tsd_key, data);
     show_tsd();
     return (NULL);
}

static void
show_tsd()
{
     void *tsd = pthread_getspecific(tsd_key);

     printf("tsd for %d = %s\n", pthread_self(), (char *)tsd);
}

/* application-specific clean-up function */
static void
cleanup(void *tsd)
{
     printf("freeing tsd for %d = %s\n", pthread_self(), (char *)tsd);
     free(tsd);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed.
MT-Level	MT-Safe
Standard	See below.

For `pthread_key_create()`, see `standards(5)`.

See Also

`pthread_once(3C)`, `pthread_getspecific(3C)`, `pthread_setspecific(3C)`, `pthread_key_delete(3C)`, `attributes(5)`, `standards(5)`

Name pthread_key_delete – delete thread-specific data key

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_key_delete(pthread_key_t key);
```

Description The `pthread_key_delete()` function deletes a thread-specific data key previously returned by `pthread_key_create()`. The thread-specific data values associated with *key* need not be `NULL` at the time `pthread_key_delete()` is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after `pthread_key_delete()` is called. Any attempt to use *key* following the call to `pthread_key_delete()` results in undefined behaviour.

The `pthread_key_delete()` function is callable from within destructor functions. No destructor functions will be invoked by `pthread_key_delete()`. Any destructor function that may have been associated with *key* will no longer be called upon thread exit.

Return Values If successful, the `pthread_key_delete()` function returns `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_key_delete()` function may fail if:

`EINVAL` The *key* value is invalid.

The `pthread_key_delete()` function will not return an error code of `EINTR`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_key_create(3C)`, `attributes(5)`, `standards(5)`

Name pthread_kill – send a signal to a thread

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <signal.h>
#include <pthread.h>

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

Description The `pthread_kill()` function sends the *sig* signal to the thread designated by thread. The *thread* argument must be a member of the same process as the calling thread. The *sig* argument must be one of the signals listed in `signal.h(3HEAD)`, with the exception of `SIGCANCEL` being reserved and off limits to `pthread_kill()`. If *sig* is 0, a validity check is performed for the existence of the target thread; no signal is sent.

Return Values Upon successful completion, the function returns a value of `0`. Otherwise the function returns an error number. If the `pthread_kill()` function fails, no signal is sent.

Errors The `pthread_kill()` function will fail if:

`ESRCH` No thread could be found corresponding to that specified by the given thread ID.

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `kill(1)`, `pthread_self(3C)`, `pthread_sigmask(3C)`, `raise(3C)`, `attributes(5)`, `standards(5)`

Name pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling – get or set prioceiling attribute of mutex attribute object

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

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

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
     int prioceiling);
```

Description The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions, respectively, get and set the priority ceiling attribute of a mutex attribute object pointed to by *attr*, which was previously created by the `pthread_mutexattr_init()` function.

The *prioceiling* attribute contains the priority ceiling of initialized mutexes. The values of *prioceiling* must be within the range of priorities defined by `SCHED_FIFO`.

The *prioceiling* attribute defines the priority ceiling of initialized mutexes, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion, the priority ceiling of the mutex must be set to a priority higher than or equal to the highest priority of all the threads that may lock that mutex.

The ceiling value should be drawn from the range of priorities for the `SCHED_FIFO` policy. When a thread acquires such a mutex, the policy of the thread at mutex acquisition should match that from which the ceiling value was derived (`SCHED_FIFO`, in this case). If a thread changes its scheduling policy while holding a ceiling mutex, the behavior of `pthread_mutex_lock()` and `pthread_mutex_unlock()` on this mutex is undefined. See `pthread_mutex_lock(3C)`.

Return Values Upon successful completion, the `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_mutexattr_setprioceiling()` function will fail if:

`EINVAL` The value specified by *attr* is `NULL` or *prioceiling* is invalid.

The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions may fail if:

`EINVAL` The value specified by *attr* or *prioceiling* is invalid.

`EPERM` The caller does not have the privilege to perform the operation.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also pthread_cond_init(3C), pthread_create(3C), pthread_mutex_init(3C), pthread_mutex_lock(3C), sched_get_priority_min(3C), attributes(5), standards(5)

Name pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol – get or set protocol attribute of mutex attribute object

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

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

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
     int protocol);
```

Description The `pthread_mutexattr_setprotocol()` and `pthread_mutexattr_getprotocol()` functions, respectively, set and get the protocol attribute of a mutex attribute object pointed to by *attr*, which was previously created by the `pthread_mutexattr_init()` function.

The *protocol* attribute defines the protocol to be followed in utilizing mutexes. The value of *protocol* may be one of `PTHREAD_PRIO_NONE`, `PTHREAD_PRIO_INHERIT`, or `PTHREAD_PRIO_PROTECT`, which are defined by the header `<pthread.h>`.

When a thread owns a mutex with the `PTHREAD_PRIO_NONE` protocol attribute, its priority and scheduling are not affected by its mutex ownership.

When a thread is blocking higher priority threads because of owning one or more mutexes with the `PTHREAD_PRIO_INHERIT` protocol attribute, it executes at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

When a thread owns one or more mutexes initialized with the `PTHREAD_PRIO_PROTECT` protocol, it executes at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes.

While a thread is holding a mutex that has been initialized with the `PRIO_INHERIT` or `PRIO_PROTECT` protocol attributes, it will not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed, such as by a call to `sched_setparam()`. Likewise, when a thread unlocks a mutex that has been initialized with the `PRIO_INHERIT` or `PRIO_PROTECT` protocol attributes, it will not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed.

If a thread simultaneously owns several mutexes initialized with different protocols, it will execute at the highest of the priorities that it would have obtained by each of these protocols.

If a thread makes a call to `pthread_mutex_lock()` for a mutex that was initialized with the protocol attribute `PTHREAD_PRIO_INHERIT`, and if the calling thread becomes blocked because the mutex is owned by another thread, then the owner thread inherits the priority level of the

calling thread for as long as it continues to own the mutex. The implementation updates its execution priority to the maximum of its assigned priority and all its inherited priorities. Furthermore, if this owner thread becomes blocked on another mutex, the same priority inheritance effect will be propagated to the other owner thread, in a recursive manner.

A thread that uses mutexes initialized with the `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT` *protocol* attribute values should have its scheduling policy equal to `SCHED_FIFO` or `SCHED_RR` (see `pthread_attr_getschedparam(3C)` and `pthread_getschedparam(3C)`).

If a thread with scheduling policy equal to `SCHED_OTHER` uses a mutex initialized with the `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT` *protocol* attribute value, the effect on the thread's scheduling and priority is unspecified.

The `_POSIX_THREAD_PRIO_INHERIT` and `_POSIX_THREAD_PRIO_PROTECT` options are designed to provide features to solve priority inversion due to mutexes. A priority inheritance or priority ceiling mutex is designed to minimize the dispatch latency of a high priority thread when a low priority thread is holding a mutex required by the high priority thread. This is a specific need for the realtime application domain.

Threads created by realtime applications need to be such that their priorities can influence their access to system resources (CPU resources, at least), in competition with all threads running on the system.

Return Values Upon successful completion, the `pthread_mutexattr_getprotocol()` and `pthread_mutexattr_setprotocol()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_mutexattr_getprotocol()` and `pthread_mutexattr_setprotocol()` functions will fail if:

`EINVAL` The value specified by *attr* is `NULL`.

`ENOSYS` Neither of the options `_POSIX_THREAD_PRIO_PROTECT` and `_POSIX_THREAD_PRIO_INHERIT` is defined and the system does not support the function.

`ENOTSUP` The value specified by *protocol* is an unsupported value.

The `pthread_mutexattr_getprotocol()` and `pthread_mutexattr_setprotocol()` functions may fail if:

`EINVAL` The value specified by *attr* or *protocol* is invalid.

`EPERM` The caller does not have the privilege to perform the operation.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also pthread_attr_getschedparam(3C), pthread_mutex_init(3C), pthread_mutexattr_init(3C), sched_setparam(3C), sched_setscheduler(3C), attributes(5), standards(5)

Name pthread_mutexattr_getpshared, pthread_mutexattr_setpshared – get or set process-shared attribute

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_mutexattr_getpshared(
     const pthread_mutexattr_t *restrict attr,
     int *restrict pshared);

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
     int pshared);
```

Description The `pthread_mutexattr_getpshared()` function obtains the value of the *process-shared* attribute from the attributes object referenced by *attr*. The `pthread_mutexattr_setpshared()` function is used to set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

The *process-shared* attribute is set to `PTHREAD_PROCESS_SHARED` to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the *process-shared* attribute is `PTHREAD_PROCESS_PRIVATE`, the mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, the behavior is undefined. The default value of the attribute is `PTHREAD_PROCESS_PRIVATE`.

Return Values Upon successful completion, `pthread_mutexattr_getpshared()` returns `0` and stores the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number is returned to indicate the error.

Upon successful completion, `pthread_mutexattr_setpshared()` returns `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_mutexattr_getpshared()` and `pthread_mutexattr_setpshared()` functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `pthread_mutexattr_setpshared()` function may fail if:

`EINVAL` The new value specified for the attribute is outside the range of legal values for that attribute.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Standard	See standards(5).

See Also pthread_create(3C), pthread_mutex_init(3C), pthread_mutexattr_init(3C), pthread_cond_init(3C), attributes(5), standards(5)

Name pthread_mutexattr_getrobust, pthread_mutexattr_setrobust – get and set the mutex robust attribute

Synopsis

```
cc –mt [ flag... ] file... [ library... ]
#include <pthread.h>

int pthread_mutexattr_getrobust(const pthread_mutexattr_t *attr,
     int *robust);

int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
     int robust);
```

Description The `pthread_mutexattr_getrobust()` and `pthread_mutexattr_setrobust()` functions, respectively, get and set the mutex robust attribute. This attribute is set in the *robust* parameter. Valid values for *robust* include:

`PTHREAD_MUTEX_STALLED`
: No special actions are taken if the owner of the mutex is terminated while holding the mutex lock. This can lead to deadlocks because no other thread can unlock the mutex. This is the default value.

`PTHREAD_MUTEX_ROBUST`
: If the owning thread of a robust mutex terminates while holding the mutex lock, or if the process containing the owning thread of a robust mutex terminates, either normally or abnormally, or if the process containing the owner of the mutex unmaps the memory containing the mutex or performs one of the `exec(2)` functions, the next thread that acquires the mutex will be notified by the return value `EOWNERDEAD` from the locking function.

  The notified thread can then attempt to recover the state protected by the mutex and, if successful, mark the state as consistent again by a call to `pthread_mutex_consistent()`. After a subsequent successful call to `pthread_mutex_unlock(3C)`, the mutex lock will be released and can be used normally by other threads. If the mutex is unlocked without a call to `pthread_mutex_consistent()`, it will be in a permanently unusable state and all attempts to lock the mutex will fail with the error `ENOTRECOVERABLE`. The only permissible operation on such a mutex is `pthread_mutex_destroy(3C)`.

  The actions required to make the state protected by the mutex consistent are solely dependent on the application. Calling `pthread_mutex_consistent(3C)` does not, by itself, make the state protected by the mutex consistent.

  The behavior is undefined if the value specified by the *attr* argument to `pthread_mutexattr_getrobust()` or `pthread_mutexattr_setrobust()` does not refer to an initialized mutex attributes object.

Return Values Upon successful completion, the `pthread_mutexattr_getrobust()` function returns 0 and stores the value of the robust attribute of *attr* into the object referenced by the robust parameter. Otherwise, an error value is returned to indicate the error.

Upon successful completion, the `pthread_mutexattr_setrobust()` function returns 0. Otherwise, an error value is returned to indicate the error.

Errors The `pthread_mutexattr_setrobust()` function will fail if:

`EINVAL` The value of *robust* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `exec(2)`, `pthread_mutex_consistent(3C)`, `pthread_mutex_destroy(3C)`, `pthread_mutex_init(3C)`, `pthread_mutex_lock(3C)`, `pthread_mutex_unlock(3C)`, `pthread_mutexattr_getpshared(3C)`, `pthread_mutexattr_init(3C)`, `attributes(5)`, `mutex(5)`, `standards(5)`

Notes The mutex memory must be zeroed before first initialization of a mutex with the `PTHREAD_MUTEX_ROBUST` attribute. Any thread in any process interested in the robust lock can call `pthread_mutex_init()` to potentially initialize it, provided that all such callers of `pthread_mutex_init()` specify the same set of attributes in their attribute structures. In this situation, if `pthread_mutex_init()` is called on a previously initialized robust mutex, it will not reinitialize the mutex and will return the error value `EBUSY`. If `pthread_mutex_init()` is called on a previously initialized robust mutex, and if the caller specifies a different set of attributes from those already in effect for the mutex, it will not reinitialize the mutex and will return the error value `EINVAL`.

Name pthread_mutexattr_gettype, pthread_mutexattr_settype – get or set mutex type

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_mutexattr_gettype(pthread_mutexattr_t *restrict attr,
     int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

Description The `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()` functions respectively get and set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The default value of the *type* attribute is `PTHREAD_MUTEX_DEFAULT`.

The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types include:

`PTHREAD_MUTEX_NORMAL` This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.

`PTHREAD_MUTEX_ERRORCHECK` This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex that another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.

`PTHREAD_MUTEX_RECURSIVE` A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock that can occur with mutexes of type `PTHREAD_MUTEX_NORMAL` cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex that another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error. This type of mutex is only supported for mutexes whose process shared attribute is `PTHREAD_PROCESS_PRIVATE`.

`PTHREAD_MUTEX_DEFAULT` Attempting to recursively lock a mutex of this type results in undefined behavior. Attempting to unlock a mutex of this type that was not locked by the calling thread results in undefined behavior. Attempting to unlock a mutex of this type that is not locked results in undefined behavior. An

implementation is allowed to map this mutex to one of the other mutex types.

Return Values Upon successful completion, the `pthread_mutexattr_settype()` function returns `0`. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the `pthread_mutexattr_gettype()` function returns `0` and stores the value of the *type* attribute of *attr* in the object referenced by the *type* parameter. Otherwise an error number is returned to indicate the error.

Errors The `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()` functions will fail if:

`EINVAL` The value `type` is invalid.

The `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()` functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_cond_timedwait(3C)`, `pthread_cond_wait(3C)`, `attributes(5)`, `standards(5)`

Notes Application should not use a `PTHREAD_MUTEX_RECURSIVE` mutex with condition variables because the implicit unlock performed for `pthread_cond_wait()` or `pthread_cond_timedwait()` will not actually release the mutex (if it had been locked multiple times). If this occurs, no other thread can satisfy the condition of the predicate.

Name pthread_mutexattr_init, pthread_mutexattr_destroy – initialize or destroy mutex attributes object

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Description The `pthread_mutexattr_init()` function initializes a mutex attributes object *attr* with the default value for all of the attributes defined by the implementation.

The effect of initializing an already initialized mutex attributes object is undefined.

After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including destruction) does not affect any previously initialized mutexes.

The `pthread_mutexattr_destroy()` function destroys a mutex attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_mutexattr_destroy()` to set the object referenced by *attr* to an invalid value. A destroyed mutex attributes object can be re-initialized using `pthread_mutexattr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

Return Values Upon successful completion, `pthread_mutexattr_init()` and `pthread_mutexattr_destroy()` return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_mutexattr_init()` function may fail if:

`ENOMEM` Insufficient memory exists to initialize the mutex attributes object.

The `pthread_mutexattr_destroy()` function may fail if:

`EINVAL` The value specified by *attr* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also pthread_cond_init(3C), pthread_create(3C), pthread_mutex_init(3C), pthread_mutexattr_settype(3C), attributes(5), standards(5)

Name pthread_mutex_consistent – mark state protected by robust mutex as consistent

Synopsis

```
cc –mt [ flag... ] file... [ library... ]
#include <pthread.h>

int pthread_mutex_consistent(pthread_mutex_t *mutex);
```

Description The following applies only to mutexes that have been initialized with the PTHREAD_MUTEX_ROBUST attribute. See pthread_mutexattr_getrobust(3C).

If mutex is a robust mutex in an inconsistent state, the pthread_mutex_consistent() function can be used to mark the state protected by the mutex referenced by mutex as consistent again.

If the owner of a robust mutex terminates while holding the mutex, or if the process containing the owner of the mutex unmaps the memory containing the mutex or performs one of the exec(2) functions, the mutex becomes inconsistent and the next thread that acquires the mutex lock is notified of the state by the return value EOWNERDEAD. In this case, the mutex does not become normally usable again until the state is marked consistent.

The pthread_mutex_consistent() function is only responsible for notifying the system that the state protected by the mutex has been recovered and that normal operations with the mutex can be resumed. It is the responsibility of the application to recover the state so it can be reused. If the application is not able to perform the recovery, it can notify the system that the situation is unrecoverable by a call to pthread_mutex_unlock(3C) without a prior call to pthread_mutex_consistent(), in which case subsequent threads that attempt to lock the mutex will fail to acquire the lock and be returned ENOTRECOVERABLE.

If the thread which acquired the mutex lock with the return value EOWNERDEAD terminates before calling either pthread_mutex_consistent() or pthread_mutex_unlock(), the next thread that acquires the mutex lock is notified about the state of the mutex by the return value EOWNERDEAD.

Return Values Upon successful completion, the pthread_mutexattr_consistent() function returns 0. Otherwise, an error value is returned to indicate the error.

Errors The pthread_mutex_consistent() function will fail if:

EINVAL The current thread does not own the mutex or the mutex is not a PTHREAD_MUTEX_ROBUST mutex having an inconsistent state (EOWNERDEAD).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Standard	See standards(5).

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also exec(2), pthread_mutex_lock(3C), pthread_mutex_unlock(3C), pthread_mutexattr_getrobust(3C), attributes(5), mutex(5), standards(5)

Name pthread_mutex_getprioceiling, pthread_mutex_setprioceiling – change priority ceiling of a mutex

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
     int *restrict prioceiling);

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

Description The `pthread_mutex_getprioceiling()` function returns the current priority ceiling of the mutex.

The `pthread_mutex_setprioceiling()` function either locks the mutex if it is unlocked, or blocks until it can successfully lock the mutex, then it changes the mutex's priority ceiling and releases the mutex. When the change is successful, the previous value of the priority ceiling is returned in *old_ceiling*. The process of locking the mutex need not adhere to the priority protect protocol.

If the `pthread_mutex_setprioceiling()` function fails, the mutex priority ceiling is not changed.

The ceiling value should be drawn from the range of priorities for the `SCHED_FIFO` policy. When a thread acquires such a mutex, the policy of the thread at mutex acquisition should match that from which the ceiling value was derived (`SCHED_FIFO`, in this case). If a thread changes its scheduling policy while holding a ceiling mutex, the behavior of `pthread_mutex_lock()` and `pthread_mutex_unlock()` on this mutex is undefined. See `pthread_mutex_lock(3C)`.

The ceiling value should not be treated as a persistent value resident in a `pthread_mutex_t` that is valid across upgrades of Solaris. The semantics of the actual ceiling value are determined by the existing priority range for the `SCHED_FIFO` policy, as returned by the `sched_get_priority_min()` and `sched_get_priority_max()` functions (see `sched_get_priority_min(3C)`) when called on the version of Solaris on which the ceiling value is being utilized.

Return Values Upon successful completion, the `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions may fail if:

`EINVAL` The value specified by *mutex* does not refer to a currently existing mutex.

The `pthread_mutex_setprioceiling()` function will fail if:

`EINVAL` The mutex was not initialized with its *protocol* attribute having the value of `PTHREAD_PRIO_PROTECT`.

`EINVAL` The priority requested by *prioceiling* is out of range.

`EPERM` The caller does not have the privilege to perform the operation.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_mutex_init(3C)`, `pthread_mutex_lock(3C)`, `sched_get_priority_min(3C)`, `attributes(5)`, `standards(5)`

Name pthread_mutex_init, pthread_mutex_destroy – initialize or destroy a mutex

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
     const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER;
```

Description The `pthread_mutex_init()` function initializes the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is `NULL`, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Except for robust mutexes, attempting to initialize an already initialized mutex results in undefined behavior.

The `pthread_mutex_destroy()` function destroys the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be re-initialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize mutexes that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_mutex_init()` with parameter *attr* specified as `NULL`, except that no error checks are performed.

Return Values If successful, the `pthread_mutex_init()` and `pthread_mutex_destroy()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_mutex_init()` function will fail if:

`EAGAIN` The system lacked the necessary resources (other than memory) to initialize another mutex.

`EBUSY` An attempt was detected to re-initialize a robust mutex previously initialized but not yet destroyed. See `pthread_mutexattr_setrobust(3C)`.

`EINVAL` An attempt was detected to re-initialize a robust mutex previously initialized with a different set of attributes. See `pthread_mutexattr_setrobust(3C)`.

`ENOMEM` Insufficient memory exists to initialize the mutex.

`EPERM` The caller does not have the privilege to perform the operation.

The `pthread_mutex_init()` function may fail if:

`EBUSY` An attempt was detected to re-initialize the object referenced by *mutex*, a mutex previously initialized but not yet destroyed.

`EINVAL` The value specified by *attr* or *mutex* is invalid.

The `pthread_mutex_destroy()` function may fail if:

`EBUSY` An attempt was detected to destroy the object referenced by *mutex* while it is locked or referenced (for example, while being used in a `pthread_cond_wait`(3C) or `pthread_cond_timedwait`(3C)) by another thread.

`EINVAL` The value specified by *mutex* is invalid.

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards`(5).

See Also

`pthread_cond_wait`(3C), `pthread_mutex_lock`(3C), `pthread_mutexattr_setprioceiling`(3C), `pthread_mutexattr_setprotocol`(3C), `pthread_mutexattr_setpshared`(3C), `pthread_mutexattr_setrobust`(3C), `pthread_mutexattr_settype`(3C), `attributes`(5), `mutex`(5), `standards`(5)

Name pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – lock or unlock a mutex

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Description The mutex object referenced by mutex is locked by calling `pthread_mutex_lock()`. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

If the mutex type is `PTHREAD_MUTEX_NORMAL`, deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, undefined behavior results.

If the mutex type is `PTHREAD_MUTEX_ERRORCHECK`, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_RECURSIVE`, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to 1. Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches `0`, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_DEFAULT`, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The `pthread_mutex_trylock()` function is identical to `pthread_mutex_lock()` except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call fails immediately with `EBUSY`.

The `pthread_mutex_unlock()` function releases the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by *mutex* when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread will acquire the mutex. (In the case of `PTHREAD_MUTEX_RECURSIVE` mutexes, the mutex becomes available when the count reaches `0` and the calling thread no longer has any locks on this mutex.)

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

Return Values If successful, the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions return `0`. Otherwise, an error number is returned to indicate the error.

The `pthread_mutex_trylock()` function returns `0` if a lock on the mutex object referenced by *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions will fail if:

`EAGAIN` The mutex could not be acquired because the maximum number of recursive locks for mutex has been exceeded.

`EINVAL` The *mutex* was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT` and the calling thread's priority is higher than the mutex's current priority ceiling.

`EPERM` The mutex was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT` and the calling thread is not in the real-time class (`SCHED_RR` or `SCHED_FIFO` scheduling class).

The `pthread_mutex_trylock()` function will fail if:

`EBUSY` The *mutex* could not be acquired because it was already locked.

The `pthread_mutex_lock()`, `pthread_mutex_trylock()` and `pthread_mutex_unlock()` functions may fail if:

`EINVAL` The value specified by *mutex* does not refer to an initialized mutex object.

The `pthread_mutex_lock()` function may fail if:

`EDEADLK` The current thread already owns the mutex.

`ENOMEM` The limit on the number of simultaneously held mutexes has been exceeded.

The `pthread_mutex_unlock()` function will fail if:

`EPERM` The mutex type is PTHREAD_MUTEX_ERRORCHECK or the mutex is a robust mutex, and the current thread does not own the mutex.

When a thread makes a call to `pthread_mutex_lock()` or `pthread_mutex_trylock()`, if the mutex is initialized with the robustness attribute having the value `PTHREAD_MUTEX_ROBUST` (see `pthread_mutexattr_getrobust(3C)`), the call will return these error values if:

`EOWNERDEAD` The last owner of this mutex died while holding the mutex, or the process containing the owner of the mutex unmapped the memory containing the mutex or performed one of the `exec(2)` functions. This mutex is now owned by the caller. The caller must now attempt to make the state protected by the mutex consistent. If it is able to clean up

the state, then it should call `pthread_mutex_consistent()` for the mutex and unlock the mutex. Subsequent calls to `pthread_mutex_lock()` and `pthread_mutex_trylock()` will behave normally, as before. If the caller is not able to clean up the state, `pthread_mutex_consistent()` should not be called for the mutex, but the mutex should be unlocked. Subsequent calls to `pthread_mutex_lock()` and `pthread_mutex_trylock()` will fail to acquire the mutex with the error value `ENOTRECOVERABLE`. If the owner who acquired the lock with `EOWNERDEAD` dies, the next owner will acquire the lock with `EOWNERDEAD`.

`ENOTRECOVERABLE` The mutex trying to be acquired was protecting the state that has been left irrecoverable by the mutex's last owner. The mutex has not been acquired. This condition can occur when the lock was previously acquired with `EOWNERDEAD`, and the owner was not able to clean up the state and unlocked the mutex without calling `pthread_mutex_consistent()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_mutex_consistent(3C)`, `pthread_mutex_init(3C)`, `pthread_mutexattr_setprotocol(3C)`, `pthread_mutexattr_setrobust(3C)`, `pthread_mutexattr_settype(3C)`, `attributes(5)`, `standards(5)`

Notes In the current implementation of threads, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `mutex_lock()`, `mutex_unlock()`, `pthread_mutex_trylock()`, and `mutex_trylock()` do not validate the mutex type. Therefore, an uninitialized mutex or a mutex with an invalid type does not return `EINVAL`. Interfaces for mutexes with an invalid type have unspecified behavior.

Uninitialized mutexes that are allocated locally may contain junk data. Such mutexes need to be initialized using `pthread_mutex_init()` or `mutex_init()`.

Name pthread_mutex_timedlock, pthread_mutex_reltimedlock_np – lock a mutex

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
     const struct timespec *restrict abs_timeout);

int pthread_mutex_reltimedlock_np(pthread_mutex_t *restrict mutex,
     const struct timespec *restrict rel_timeout);
```

Description The `pthread_mutex_timedlock()` function locks the mutex object referenced by *mutex*. If the mutex is already locked, the calling thread blocks until the mutex becomes available as in the `pthread_mutex_lock(3C)`. If the mutex cannot be locked without waiting for another thread to unlock the mutex, this wait is terminated when the specified timeout expires.

The `pthread_mutex_reltimedlock_np()` function is identical to the `pthread_mutex_timedlock()` function, except that the timeout is specified as a relative time interval.

For `pthread_mutex_timedlock()`, the timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

For `pthread_mutex_reltimedlock_np()`, the timeout expires when the time interval specified by *rel_timeout* passes, as measured by the `CLOCK_REALTIME` clock, or if the time interval specified by *rel_timeout* is negative at the time of the call.

The resolution of the timeout is the resolution of the `CLOCK_REALTIME` clock. The `timespec` data type is defined in the `<time.h>`header.

Under no circumstance will either function fail with a timeout if the mutex can be locked immediately. The validity of the *timeout* parameter is not checked if the mutex can be locked immediately.

As a consequence of the priority inheritance rules (for mutexes initialized with the `PRIO_INHERIT` protocol), if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex is adjusted as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

Return Values Upon successful completion, the `pthread_mutex_timedlock()` and `pthread_mutex_reltimedlock_np()` functions return 0. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_mutex_timedlock()` and `pthread_mutex_reltimedlock_np()` functions will fail for the same reasons as `pthread_mutex_lock(3C)`. In addition, they will fail if:

`EINVAL` The caller would have blocked and the *timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1,000 million.

`ETIMEDOUT` The mutex could not be locked before the specified *timeout* expired.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	For `pthread_mutex_timedlock()`, see `standards(5)`.

See Also `time(2)`, `pthread_mutex_destroy(3C)`, `pthread_mutex_lock(3C)`, `pthread_mutex_trylock(3C)`, `attributes(5)`, `standards(5)`

Name pthread_once – initialize dynamic package

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]

#include <pthread.h>
pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control,
     void (*init_routine)(void));
```

Description If any thread in a process with a *once_control* parameter makes a call to `pthread_once()`, the first call will summon the `init_routine()`, but subsequent calls will not. The *once_control* parameter determines whether the associated initialization routine has been called. The `init_routine()` is complete upon return of `pthread_once()`.

`pthread_once()` is not a cancellation point; however, if the function `init_routine()` is a cancellation point and is canceled, the effect on *once_control* is the same as if `pthread_once()` had never been called.

The constant `PTHREAD_ONCE_INIT` is defined in the `<pthread.h>` header.

If *once_control* has automatic storage duration or is not initialized by `PTHREAD_ONCE_INIT`, the behavior of `pthread_once()` is undefined.

Return Values Upon successful completion, `pthread_once()` returns `0`. Otherwise, an error number is returned to indicate the error.

Errors `EINVAL` *once_control* or *init_routine* is `NULL`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `attributes(5)`, `standards(5)`

Notes Solaris threads do not offer this functionality.

Name pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared – get or set process-shared attribute of read-write lock attributes object

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_rwlockattr_getpshared(
     const pthread_rwlockattr_t *restrict attr,
     int *restrict pshared);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
     int pshared);
```

Description The *process-shared* attribute is set to `PTHREAD_PROCESS_SHARED` to permit a read-write lock to be operated upon by any thread that has access to the memory where the read-write lock is allocated, even if the read-write lock is allocated in memory that is shared by multiple processes. If the *process-shared* attribute is `PTHREAD_PROCESS_PRIVATE`, the read-write lock will only be operated upon by threads created within the same process as the thread that initialised the read-write lock; if threads of differing processes attempt to operate on such a read-write lock, the behaviour is undefined. The default value of the *process-shared* attribute is `PTHREAD_PROCESS_PRIVATE`.

The `pthread_rwlockattr_getpshared()` function obtains the value of the *process-shared* attribute from the initialised attributes object referenced by *attr*. The `pthread_rwlockattr_setpshared()` function is used to set the *process-shared* attribute in an initialised attributes object referenced by attr.

Return Values If successful, the `pthread_rwlockattr_setpshared()` function returns `0`. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the `pthread_rwlockattr_getpshared()` returns `0` and stores the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise an error number is returned to indicate the error.

Errors The `pthread_rwlockattr_getpshared()` and `pthread_rwlockattr_setpshared()` functions will fail if:

`EINVAL` The value specified by *attr* or *pshared* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also pthread_rwlock_init(3C), pthread_rwlock_rdlock(3C), pthread_rwlock_unlock(3C), pthread_rwlock_wrlock(3C), pthread_rwlockattr_init(3C), attributes(5), standards(5)

Name pthread_rwlockattr_init, pthread_rwlockattr_destroy – initialize or destroy read-write lock attributes object

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Description The `pthread_rwlockattr_init()` function initializes a read-write lock attributes object *attr* with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_rwlockattr_init()` is called specifying an already initialized read-write lock attributes object.

After a read-write lock attributes object has been used to initialize one or more read-write locks, any function affecting the attributes object (including destruction) does not affect any previously initialized read-write locks.

The `pthread_rwlockattr_destroy()` function destroys a read-write lock attributes object. The effect of subsequent use of the object is undefined until the object is re-initialized by another call to `pthread_rwlockattr_init()`. An implementation can cause `pthread_rwlockattr_destroy()` to set the object referenced by *attr* to an invalid value.

Return Values If successful, the `pthread_rwlockattr_init()` and `pthread_rwlockattr_destroy()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_rwlockattr_init()` function will fail if:

`ENOMEM` Insufficient memory exists to initialize the read-write lock attributes object.

The `pthread_rwlockattr_destroy()` function may fail if:

`EINVAL` The value specified by *attr* is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_rwlock_init(3C)`, `pthread_rwlock_rdlock(3C)`, `pthread_rwlock_unlock(3C)`, `pthread_rwlock_wrlock(3C)`, `pthread_rwlockattr_getpshared(3C)`, `attributes(5)`, `standards(5)`

Name pthread_rwlock_init, pthread_rwlock_destroy – initialize or destroy read-write lock object

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
     const pthread_rwlockattr_t *restrict attr);

int pthread_rwlock_destroy(pthread_rwlock_t **rwlock);

pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;
```

Description The `pthread_rwlock_init()` function initializes the read-write lock referenced by *rwlock* with the attributes referenced by *attr*. If *attr* is NULL, the default read-write lock attributes are used; the effect is the same as passing the address of a default read-write lock attributes object. Once initialized, the lock can be used any number of times without being re-initialized. Upon successful initialization, the state of the read-write lock becomes initialized and unlocked. Results are undefined if `pthread_rwlock_init()` is called specifying an already initialized read-write lock. Results are undefined if a read-write lock is used without first being initialized.

If the `pthread_rwlock_init()` function fails, *rwlock* is not initialized and the contents of *rwlock* are undefined.

The `pthread_rwlock_destroy()` function destroys the read-write lock object referenced by *rwlock* and releases any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is re-initialized by another call to `pthread_rwlock_init()`. An implementation may cause `pthread_rwlock_destroy()` to set the object referenced by *rwlock* to an invalid value. Results are undefined if `pthread_rwlock_destroy()` is called when any thread holds *rwlock*. Attempting to destroy an uninitialized read-write lock results in undefined behaviour. A destroyed read-write lock object can be re-initialized using `pthread_rwlock_init()`; the results of otherwise referencing the read-write lock object after it has been destroyed are undefined.

In cases where default read-write lock attributes are appropriate, the macro `PTHREAD_RWLOCK_INITIALIZER` can be used to initialize read-write locks that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_rwlock_init()` with the parameter *attr* specified as NULL, except that no error checks are performed.

Return Values If successful, the `pthread_rwlock_init()` and `pthread_rwlock_destroy()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_rwlock_init()` and `pthread_rwlock_destroy()` functions will fail if:

`EINVAL` The value specified by *attr* is invalid.

`EINVAL` The value specified by *rwlock* is invalid.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also pthread_rwlock_rdlock(3C), pthread_rwlock_unlock(3C), pthread_rwlock_wrlock(3C), pthread_rwlockattr_init(3C), attributes(5), standards(5)

Name pthread_rwlock_rdlock, pthread_rwlock_tryrdlock – lock or attempt to lock read-write lock object for reading

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

Description The `pthread_rwlock_rdlock()` function applies a read lock to the read-write lock referenced by *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock.

The calling thread does not acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock; otherwise, the calling thread acquires the lock. If the read lock is not acquired, the calling thread blocks until it can acquire the lock.

A thread can hold multiple concurrent read locks on *rwlock* (that is, successfully call the `pthread_rwlock_rdlock()` function *n* times). If so, the thread must perform matching unlocks (that is, it must call the `pthread_rwlock_unlock()` function *n* times).

The maximum number of concurrent read locks that a thread can hold on one read-write lock is currently set at 100,000, though this number could change in a future release. There is no imposed limit on the number of different threads that can apply a read lock to one read-write lock.

The `pthread_rwlock_tryrdlock()` function applies a read lock like the `pthread_rwlock_rdlock()` function, with the exception that the function fails if the equivalent `pthread_rwlock_rdlock()` call would have blocked the calling thread. In no case will the `pthread_rwlock_tryrdlock()` function ever bloc. It always either acquires the lock or fails and returns immediately.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the signal handler the thread resumes waiting for the read-write lock for reading as if it was not interrupted.

Return Values If successful, the `pthread_rwlock_rdlock()` function returns `0`. Otherwise, an error number is returned to indicate the error.

The `pthread_rwlock_tryrdlock()` function returns `0` if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise an error number is returned to indicate the error.

Errors The `pthread_rwlock_rdlock()` and `pthread_rwlock_tryrdlock()` functions will fail if:

`EAGAIN` The read lock could not be acquired because the maximum number of read locks by the current thread for *rwlock* has been exceeded.

The `pthread_rwlock_rdlock()` function will fail if:

`EDEADLK` The current thread already owns the read-write lock for writing.

The `pthread_rwlock_tryrdlock()` function will fail if:

`EBUSY` The read-write lock could not be acquired for reading because a writer holds the lock or a writer with the appropriate priority was blocked on it.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_rwlock_init(3C)`, `pthread_rwlock_wrlock(3C)`, `pthread_rwlockattr_init(3C)`, `pthread_rwlock_unlock(3C)`, `attributes(5)`, `standards(5)`

Name pthread_rwlock_timedrdlock, pthread_rwlock_reltimedrdlock_np – lock a read-write lock for reading

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
     const struct timespec *restrict abs_timeout);

int pthread_rwlock_reltimedrdlock_np(pthread_rwlock_t *restrict rwlock,
     const struct timespec *restrict rel_timeout);
```

Description The `pthread_rwlock_timedrdlock()` function applies a read lock to the read-write lock referenced by *rwlock* as in the `pthread_rwlock_rdlock(3C)` function. If the lock cannot be acquired without waiting for other threads to unlock the lock, this wait will be terminated when the specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the `CLOCK_REALTIME` `clock` (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

The `pthread_rwlock_reltimedrdlock_np()` function is identical to the `pthread_rwlock_timedrdlock()` function, except that the timeout is specified as a relative time interval. The timeout expires when the time interval specified by *rel_timeout* passes, as measured by the `CLOCK_REALTIME` clock, or if the time interval specified by *rel_timeout* is negative at the time of the call.

The resolution of the timeout is the resolution of the `CLOCK_REALTIME` clock. The `timespec` data type is defined in the `<time.h>` header. Under no circumstances does either function fail with a timeout if the lock can be acquired immediately. The validity of the timeout parameter need not be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock with a call to `pthread_rwlock_timedrdlock()` or `pthread_rwlock_reltimedrdlock_np()`, upon return from the signal handler the thread resumes waiting for the lock as if it was not interrupted.

The calling thread might deadlock if at the time the call is made it holds a write lock on rwlock.

The results are undefined if this function is called with an uninitialized read-write lock.

Return Values The `pthread_rwlock_timedrdlock()` and `pthread_rwlock_reltimedrdlock_np()` functions return 0 if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_rwlock_timedrdlock()` and and `pthread_rwlock_reltimedrdlock_np()` functions will fail if:

`ETIMEDOUT` The lock could not be acquired before the specified timeout expired.

The `pthread_rwlock_timedrdlock()` and `pthread_rwlock_reltimedrdlock_np()` functions may fail if:

`EAGAIN` The read lock could not be acquired because the maximum number of read locks for lock would be exceeded.

`EDEADLK` The calling thread already holds a write lock on *rwlock*.

`EINVAL` The value specified by *rwlock* does not refer to an initialized read-write lock object, or the timeout nanosecond value is less than zero or greater than or equal to 1 000 million.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	For `pthread_rwlock_timedrdlock()`, see `standards(5)`.

See Also `pthread_rwlock_destroy(3C)`, `pthread_rwlock_rdlock(3C)`, `pthread_rwlock_timedwrlock(3C)`, `pthread_rwlock_trywrlock(3C)`, `pthread_rwlock_unlock(3C)`, `pthread_rwlock_wrlock(3C)`, `attributes(5)`, `standards(5)`

Name pthread_rwlock_timedwrlock, pthread_rwlock_reltimedwrlock_np – lock a read-write lock for writing

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
     const struct timespec *restrict abs_timeout);

int pthread_rwlock_reltimedwrlock_np(pthread_rwlock_t *restrict rwlock,
     const struct timespec *restrict rel_timeout);
```

Description The `pthread_rwlock_timedwrlock()` function applies a write lock to the read-write lock referenced by *rwlock* as in the `pthread_rwlock_wrlock(3C)` function. If the lock cannot be acquired without waiting for other threads to unlock the lock, this wait will be terminated when the specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the `CLOCK_REALTIME` clock (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

The `pthread_rwlock_reltimedwrlock_np()` function is identical to the `pthread_rwlock_timedwrlock()` function, except that the timeout is specified as a relative time interval. The timeout expires when the time interval specified by *rel_timeout* passes, as measured by the `CLOCK_REALTIME` clock, or if the time interval specified by *rel_timeout* is negative at the time of the call.

The resolution of the timeout is the resolution of the `CLOCK_REALTIME` clock. The `timespec` data type is defined in the `<time.h>` header. Under no circumstances does either function fail with a timeout if the lock can be acquired immediately. The validity of the a*bs_timeout* parameter need not be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock with a call to `pthread_rwlock_timedwrlock()` or `pthread_rwlock_reltimedwrlock_np()`, upon return from the signal handler the thread resumes waiting for the lock as if it was not interrupted.

The calling thread can deadlock if at the time the call is made it holds the read-write lock. The results are undefined if this function is called with an uninitialized read-write lock.

Return Values The `pthread_rwlock_timedwrlock()` and `pthread_rwlock_reltimedwrlock_np()` functions return 0 if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_rwlock_timedwrlock()` and `pthread_rwlock_reltimedwrlock_np()` functions will fail if:

`ETIMEDOUT` The lock could not be acquired before the specified timeout expired.

The `pthread_rwlock_timedwrlock()` and `pthread_rwlock_reltimedwrlock_np()` functions may fail if:

`EDEADLK` The calling thread already holds the rwlock.

`EINVAL` The value specified by *rwlock* does not refer to an initialized read-write lock object, or the timeout nanosecond value is less than zero or greater than or equal to 1,000 million.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	For `pthread_rwlock_timedwrlock()`, see `standards(5)`.

See Also `pthread_rwlock_destroy(3C)`, `pthread_rwlock_rdlock(3C)`, `pthread_rwlock_timedrdlock(3C)`, `pthread_rwlock_trywrlock(3C)`, `pthread_rwlock_unlock(3C)`, `pthread_rwlock_wrlock(3C)`, `attributes(5)`, `standards(5)`

Name pthread_rwlock_unlock – unlock read-write lock object

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Description The `pthread_rwlock_unlock()` function is called to release a lock held on the read-write lock object referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the calling thread.

If this function is called to release a read lock from the read-write lock object and there are other read locks currently held on this read-write lock object, the read-write lock object remains in the read locked state. If this function releases the calling thread's last read lock on this read-write lock object, then the calling thread is no longer one of the owners of the object. If this function releases the last read lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If this function is called to release a write lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If the call to the `pthread_rwlock_unlock()` function results in the read-write lock object becoming unlocked and there are multiple threads waiting to acquire the read-write lock object for writing, the scheduling policy is used to determine which thread acquires the read-write lock object for writing. If there are multiple threads waiting to acquire the read-write lock object for reading, the scheduling policy is used to determine the order in which the waiting threads acquire the read-write lock object for reading. If there are multiple threads blocked on *rwlock* for both read locks and write locks, it is unspecified whether the readers acquire the lock first or whether a writer acquires the lock first.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

Return Values If successful, the `pthread_rwlock_unlock()` function returns `0`. Otherwise, an error number is returned to indicate the error.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also pthread_rwlock_init(3C), pthread_rwlock_rdlock(3C), pthread_rwlock_wrlock(3C), pthread_rwlockattr_init(3C), attributes(5), standards(5)

Name pthread_rwlock_wrlock, pthread_rwlock_trywrlock – lock or attempt to lock read-write lock object for writing

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

Description The `pthread_rwlock_wrlock()` function applies a write lock to the read-write lock referenced by *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock *rwlock*. Otherwise, the thread blocks until it can acquire the lock.

The `pthread_rwlock_trywrlock()` function applies a write lock like the `pthread_rwlock_wrlock()` function, with the exception that the function fails if any thread currently holds *rwlock* (for reading or writing).

Writers are favored over readers of the same priority to avoid writer starvation. See `pthread_rwlock_rdlock(3C)`.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the signal handler the thread resumes waiting for the read-write lock for writing as if it was not interrupted.

Return Values If successful, the `pthread_rwlock_wrlock()` function returns `0`. Otherwise, an error number is returned to indicate the error.

The `pthread_rwlock_trywrlock()` function returns `0` if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise an error number is returned to indicate the error.

Errors The `pthread_rwlock_wrlock()` function will fail if:

`EDEADLK` The current thread already owns the read-write lock for writing or reading.

The `pthread_rwlock_trywrlock()` function will fail if:

`EBUSY` The read-write lock could not be acquired for writing because it was already locked for reading or writing.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also pthread_rwlock_init(3C), pthread_rwlock_unlock(3C), pthread_rwlockattr_init(3C), pthread_rwlock_rdlock(3C), attributes(5), standards(5)

Name pthread_self – get calling thread's ID

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]

#include <pthread.h>

pthread_t pthread_self(void);
```

Description The `pthread_self()` function returns the thread ID of the calling thread.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_create(3C)`, `pthread_equal(3C)`, `attributes(5)`, `standards(5)`

Name pthread_setcancelstate – enable or disable cancellation

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

Description The `pthread_setcancelstate()` function atomically sets the calling thread's cancellation state to the specified *state* and if *oldstate* is not `NULL`, stores the previous cancellation *state* in *oldstate*.

The *state* can be either of the following:

`PTHREAD_CANCEL_ENABLE`
: This is the default. When cancellation is deferred (deferred cancellation is also the default), cancellation occurs when the target thread reaches a cancellation point and a cancel is pending. When cancellation is asynchronous, receipt of a `pthread_cancel(3C)` call causes immediate cancellation.

`PTHREAD_CANCEL_DISABLE`
: When cancellation is deferred, all cancellation requests to the target thread are held pending. When cancellation is asynchronous, all cancellation requests to the target thread are held pending; as soon as cancellation is re-enabled, pending cancellations are executed immediately.

See `cancellation(5)` for the definition of a cancellation point and a discussion of cancellation concepts. See `pthread_setcanceltype(3C)` for explanations of deferred and asynchronous cancellation.

Return Values Upon successful completion, `pthread_setcancelstate()`, returns `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_setcancelstate()` function will fail if:

`EINVAL` The specified *state* is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_cancel(3C)`, `pthread_cleanup_pop(3C)`, `pthread_cleanup_push(3C)`, `pthread_exit(3C)`, `pthread_join(3C)`, `pthread_setcanceltype(3C)`, `pthread_testcancel(3C)`, `setjmp(3C)`, `attributes(5)`, `cancellation(5)`, `condition(5)`, `standards(5)`

Name pthread_setcanceltype – set cancellation type of a thread

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

Description The `pthread_setcanceltype()` function atomically sets the calling thread's cancellation type to the specified `type` and, if *oldtype* is not `NULL`, stores the previous cancellation `type` in *oldtype*. The `type` can be either of the following:

`PTHREAD_CANCEL_DEFERRED` This is the default. When cancellation is enabled (enabled cancellation is also the default), cancellation occurs when the target thread reaches a cancellation point and a cancel is pending. When cancellation is disabled, all cancellation requests to the target thread are held pending.

`PTHREAD_CANCEL_ASYNCHRONOUS` When cancellation is enabled, receipt of a `pthread_cancel(3C)` call causes immediate cancellation. When cancellation is disabled, all cancellation requests to the target thread are held pending; as soon as cancellation is re-enabled, pending cancellations are executed immediately.

See `cancellation(5)` for the definition of a cancellation point and a discussion of cancellation concepts. See `pthread_setcancelstate(3C)` for explanations of enabling and disabling cancellation.

The `pthread_setcanceltype()` function is a cancellation point if `type` is called with `PTHREAD_CANCEL_ASYNCHRONOUS` and the cancellation state is `PTHREAD_CANCEL_ENABLE`.

Return Values Upon successful completion, the `pthread_setcanceltype()` function returns `0`. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_setcanceltype()` function will fail if:

`EINVAL` The specified `type` is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also pthread_cancel(3C), pthread_cleanup_pop(3C), pthread_cleanup_push(3C), pthread_exit(3C), pthread_join(3C), pthread_setcancelstate(3C), pthread_testcancel(3C), setjmp(3C), attributes(5), cancellation(5), condition(5), standards(5)

Name pthread_setschedprio – dynamic thread scheduling parameters access

Synopsis

```
cc -mt [ flag... ] file... -lpthread [ library... ]
#include <pthread.h>

int pthread_setschedprio(pthread_t thread, int prio);
```

Description The `pthread_setschedprio()` function sets the scheduling priority for the thread whose thread ID is given by *thread* to the value given by *prio*.

If the `pthread_setschedprio()` function fails, the scheduling priority of the target thread is not changed.

Return Values If successful, the `pthread_setschedprio()` function returns 0; otherwise, an error number is returned to indicate the error.

Errors The `pthread_setschedprio()` function will fail if:

`EINVAL` The value of *prio* is invalid for the scheduling policy of the specified thread.

`EPERM` The caller does not have the appropriate permission to set the priority to the value specified.

`ESRCH` The value specified by thread does not refer to an existing thread.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_getschedparam(3C)`, `attributes(5)`, `standards(5)`

Name pthread_sigmask – change or examine calling thread's signal mask

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

Description The `pthread_sigmask()` function changes or examines a calling thread's signal mask. Each thread has its own signal mask. A new thread inherits the calling thread's signal mask and priority; however, pending signals are not inherited. Signals pending for a new thread will be empty.

If the value of the argument *set* is not `NULL`, *set* points to a set of signals that can modify the currently blocked set. If the value of *set* is `NULL`, the value of *how* is insignificant and the thread's signal mask is unmodified; thus, `pthread_sigmask()` can be used to inquire about the currently blocked signals.

The value of the argument *how* specifies the method in which the set is changed and takes one of the following values:

`SIG_BLOCK` *set* corresponds to a set of signals to block. They are added to the current signal mask.

`SIG_UNBLOCK` *set* corresponds to a set of signals to unblock. These signals are deleted from the current signal mask.

`SIG_SETMASK` *set* corresponds to the new signal mask. The current signal mask is replaced by `set`.

If the value of *oset* is not `NULL`, it points to the location where the previous signal mask is stored.

Return Values Upon successful completion, the `pthread_sigmask()` function returns `0`. Otherwise, it returns a non-zero value.

Errors The `pthread_sigmask()` function will fail if:

`EINVAL` The value of *how* is not defined and *oset* is `NULL`.

Examples **EXAMPLE 1** Create a default thread that can serve as a signal catcher/handler with its own signal mask.

The following example shows how to create a default thread that can serve as a signal catcher/handler with its own signal mask. `new` will have a different value from the creator's signal mask.

As POSIX threads and Solaris threads are fully compatible even within the same process, this example uses `pthread_create(3C)` if you execute `a.out 0`, or `thr_create(3C)` if you execute `a.out 1`.

EXAMPLE 1 Create a default thread that can serve as a signal catcher/handler with its own signal mask. *(Continued)*

In this example:

- The `sigemptyset(3C)` function initializes a null signal set, `new`. The `sigaddset(3C)` function packs the signal, `SIGINT`, into that new set.
- Either `pthread_sigmask()` or `thr_sigsetmask()` is used to mask the signal, `SIGINT` (CTRL-C), from the calling thread, which is `main()`. The signal is masked to guarantee that only the new thread will receive this signal.
- `pthread_create()` or `thr_create()` creates the signal-handling thread.
- Using `pthread_join(3C)` or `thr_join(3C)`, `main()` then waits for the termination of that signal-handling thread, whose ID number is `user_threadID`; `main()` will then `sleep(3C)` for 2 seconds, after which the program terminates.
- The signal-handling thread, `handler`:
  - Assigns the handler `interrupt()` to handle the signal `SIGINT`, by the call to `sigaction(2)`.
  - Resets its own signal set to *not block* the signal, `SIGINT`.
  - Sleeps for 8 seconds to allow time for the user to deliver the signal, `SIGINT`, by pressing the CTRL-C.

```
/* cc thisfile.c -lthread -lpthread */
#define _REENTRANT    /* basic first 3-lines for threads */
#include <pthread.h>
#include <thread.h>
thread_t user_threadID;
sigset_t new;
void *handler( ), interrupt( );

int
main( int argc, char *argv[ ] )  {
    test_argv(argv[1]);

    sigemptyset(&new);
    sigaddset(&new, SIGINT);
    switch(*argv[1])  {

        case '0':   /* POSIX */
            pthread_sigmask(SIG_BLOCK, &new, NULL);
            pthread_create(&user_threadID, NULL, handler,
            argv[1]);
            pthread_join(user_threadID, NULL);
            break;
```

EXAMPLE 1 Create a default thread that can serve as a signal catcher/handler with its own signal mask. *(Continued)*

```
        case '1':   /* Solaris */
            thr_sigsetmask(SIG_BLOCK, &new, NULL);
            thr_create(NULL, 0, handler, argv[1], 0,
            &user_threadID);
            thr_join(user_threadID, NULL, NULL);
            break;
        }  /* switch */

    printf("thread handler, # %d, has exited\n",user_threadID);
    sleep(2);
    printf("main thread, # %d is done\n", thr_self( ));
 return (0)
} /* end main */

struct sigaction act;

void *
handler(char *argv1)
{
    act.sa_handler = interrupt;
    sigaction(SIGINT, &act, NULL);
    switch(*argv1)    {
        case '0':     /* POSIX */
            pthread_sigmask(SIG_UNBLOCK, &new, NULL);
            break;
        case '1':   /* Solaris */
            thr_sigsetmask(SIG_UNBLOCK, &new, NULL);
            break;
    }
    printf("\n Press CTRL-C to deliver SIGINT signal to the
     process\n");
    sleep(8);  /* give user time to hit CTRL-C */
 return (NULL)
}

void
interrupt(int sig)
{
 printf("thread %d caught signal %d\n", thr_self( ), sig);
}

void test_argv(char argv1[ ])    {
    if(argv1 == NULL)  {
        printf("use 0 as arg1 to use thr_create( );\n \
```

EXAMPLE 1 Create a default thread that can serve as a signal catcher/handler with its own signal mask. *(Continued)*

```
        or use 1 as arg1 to use pthread_create( )\n");
        exit(NULL);
    }
}
```

In the last example, the `handler` thread served as a signal-handler while also taking care of activity of its own (in this case, sleeping, although it could have been some other activity). A thread could be completely dedicated to signal-handling simply by waiting for the delivery of a selected signal by blocking with `sigwait(2)`. The two subroutines in the previous example, `handler()` and `interrupt()`, could have been replaced with the following routine:

```
void *
handler(void *unused)
{
    int signal;
    printf("thread %d is waiting for you to press the CTRL-C keys\n",
            thr_self( ));
    sigwait(&new, &signal);
    printf("thread %d has received the signal %d \n", thr_self( ),
        signal);
    return (NULL);
}
/* pthread_create( ) and thr_create( ) would use NULL instead
   of argv[1] for the arg passed to handler( ) */
```

In this routine, one thread is dedicated to catching and handling the signal specified by the set `new`, which allows `main()` and all of its other sub-threads, created *after* `pthread_sigmask()` or `thr_sigsetmask()` masked that signal, to continue uninterrupted. Any use of `sigwait(2)` should be such that all threads block the signals passed to `sigwait(2)` at all times. Only the thread that calls `sigwait()` will get the signals. The call to `sigwait(2)` takes two arguments.

For this type of background dedicated signal-handling routine, a Solaris daemon thread can be used by passing the argument `THR_DAEMON` to `thr_create(3C)`.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe and Async-Signal-Safe
Standard	See `standards(5)`.

See Also sigaction(2), sigprocmask(2), sigwait(2), cond_wait(3C), pthread_cancel(3C), pthread_create(3C), pthread_join(3C), pthread_self(3C), sigaddset(3C), sigemptyset(3C), sigsetops(3C), sleep(3C), attributes(5), cancellation(5), standards(5)

Notes It is not possible to block signals that cannot be caught or ignored (see sigaction(2)). It is also not possible to block or unblock SIGCANCEL, as SIGCANCEL is reserved for the implementation of POSIX thread cancellation (see pthread_cancel(3C) and cancellation(5)). This restriction is quietly enforced by the standard C library.

Using sigwait(2) in a dedicated thread allows asynchronously generated signals to be managed synchronously; however, sigwait(2) should never be used to manage synchronously generated signals.

Synchronously generated signals are exceptions that are generated by a thread and are directed at the thread causing the exception. Since sigwait() blocks waiting for signals, the blocking thread cannot receive a synchronously generated signal.

The sigprocmask(2) function behaves the same as if pthread_sigmask() has been called. POSIX leaves the semantics of the call to sigprocmask(2) unspecified in a multi-threaded process, so programs that care about POSIX portability should not depend on this semantic.

If a signal is delivered while a thread is waiting on a condition variable, the cond_wait(3C) function will be interrupted and the handler will be executed. The state of the lock protecting the condition variable is undefined while the thread is executing the signal handler.

Although pthread_sigmask() is Async-Signal-Safe with respect to the Solaris environment, this safeness is not guaranteed to be portable to other POSIX domains.

Signals that are generated synchronously should not be masked. If such a signal is blocked and delivered, the receiving process is killed.

Name pthread_spin_destroy, pthread_spin_init – destroy or initialize a spin lock object

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>

int pthread_spin_destroy(pthread_spinlock_t *lock);

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

Description The `pthread_spin_destroy()` function destroys the spin lock referenced by *lock* and release any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is reinitialized by another call to `pthread_spin_init()`. The results are undefined if `pthread_spin_destroy()` is called when a thread holds the lock, or if this function is called with an uninitialized thread spin lock.

The `pthread_spin_init()` function allocates any resources required to use the spin lock referenced by *lock* and initialize the lock to an unlocked state.

If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is `PTHREAD_PROCESS_SHARED`, the spin lock can be operated upon by any thread that has access to the memory where the spin lock is allocated, even if it is allocated in memory that is shared by multiple processes.

If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is `PTHREAD_PROCESS_PRIVATE`, or if the option is not supported, the spin lock can only be operated upon by threads created within the same process as the thread that initialized the spin lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is undefined.

The results are undefined if `pthread_spin_init()` is called specifying an already initialized spin lock. The results are undefined if a spin lock is used without first being initialized.

If the `pthread_spin_init()` function fails, the lock is not initialized and the contents of *lock* are undefined.

Only the object referenced by *lock* can be used for performing synchronization.

The result of referring to copies of that object in calls to `pthread_spin_destroy()`, `pthread_spin_lock(3C)`, `pthread_spin_trylock(3C)`, or `pthread_spin_unlock(3C)` is undefined.

Return Values Upon successful completion, these functions returns 0. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_spin_init()` function will fail if:

`EAGAIN` The system lacks the necessary resources to initialize another spin lock.

These functions may fail if:

EBUSY The system has detected an attempt to initialize or destroy a spin lock while it is in use (for example, while being used in a `pthread_spin_lock()` call) by another thread.

EINVAL The value specified by *lock* is invalid.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also

`pthread_spin_lock(3C)`, `pthread_spin_unlock(3C)`, `attributes(5)`, `standards(5)`

Name pthread_spin_lock, pthread_spin_trylock – lock a spin lock object

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);

#include <pthread.h>

int pthread_spin_trylock(pthread_spinlock_t *lock);
```

Description The `pthread_spin_lock()` function locks the spin lock referenced by *lock*. The calling thread acquires the lock if it is not held by another thread. Otherwise, the thread spins (that is, does not return from the `pthread_spin_lock call()`) until the lock becomes available. The results are undefined if the calling thread holds the lock at the time the call is made.

The `pthread_spin_trylock()` function locks the spin lock referenced by *lock* if it is not held by any thread. Otherwise, the function fails.

The results are undefined if either of these functions is called with an uninitialized spin lock.

Return Values Upon successful completion, these functions returns 0. Otherwise, an error number is returned to indicate the error.

Errors The `pthread_spin_trylock()` function will fail if:

`EBUSY` A thread currently holds the lock.

These functions may fail if:

`EINVAL` The value specified by *lock* does not refer to an initialized spin lock object.

The `pthread_spin_lock()` function may fail if:

`EDEADLK` The calling thread already holds the lock.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_spin_destroy(3C)`, `pthread_spin_unlock(3C)`, `attributes(5)`, `standards(5)`

Name pthread_spin_unlock – unlock a spin lock object

Synopsis

```
cc -mt [ flag... ] file... [ library... ]
#include <pthread.h>

int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Description The `pthread_spin_unlock()` function releases the spin lock referenced by *lock* which was locked with the `pthread_spin_lock(3C)` or `pthread_spin_trylock(3C)` functions. The results are undefined if the lock is not held by the calling thread. If there are threads spinning on the lock when `pthread_spin_unlock()` is called, the lock becomes available and an unspecified spinning thread acquires the lock.

The results are undefined if this function is called with an uninitialized thread spin lock.

Return Values Upon successful completion, the `pthread_spin_unlock()` function returns 0. Otherwise, an error number shall be returned to indicate the error.

Errors The `pthread_spin_unlock()` function will fail if:

`EINVAL` An invalid argument was specified.

`EPERM` The calling thread does not hold the lock.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_spin_destroy(3C)`, `pthread_spin_lock(3C)`, `attributes(5)`, `standards(5)`

Name pthread_testcancel – create cancellation point in the calling thread

Synopsis

```
cc –mt [ flag... ] file... –lpthread [ library... ]
#include <pthread.h>

void pthread_testcancel(void);
```

Description The `pthread_testcancel()` function forces testing for cancellation. This is useful when you need to execute code that runs for long periods without encountering cancellation points; such as a library routine that executes long-running computations without cancellation points. This type of code can block cancellation for unacceptable long periods of time. One strategy for avoiding blocking cancellation for long periods, is to insert calls to `pthread_testcancel()` in the long-running computation code and to setup a cancellation handler in the library code, if required.

Return Values The `pthread_testcancel()` function returns `void`.

Errors The `pthread_testcancel()` function does not return errors.

Examples See `cancellation(5)` for an example of using `pthread_testcancel()` to force testing for cancellation and a discussion of cancellation concepts.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `Intro(3)`, `pthread_cleanup_pop(3C)`, `pthread_cleanup_push(3C)`, `pthread_exit(3C)`, `pthread_join(3C)`, `pthread_setcancelstate(3C)`, `pthread_setcanceltype(3C)`, `setjmp(3C)`, `attributes(5)`, `cancellation(5)`, `condition(5)`, `standards(5)`

Notes The `pthread_testcancel()` function has no effect if cancellation is disabled.

Use `pthread_testcancel()` with `pthread_setcanceltype()` called with its *canceltype* set to `PTHREAD_CANCEL_DEFERRED`. The `pthread_testcancel()` function operation is undefined if `pthread_setcanceltype()` was called with its *canceltype* argument set to `PTHREAD_CANCEL_ASYNCHRONOUS`.

It is possible to kill a thread when it is holding a resource, such as lock or allocated memory. If that thread has not setup a cancellation cleanup handler to release the held resource, the application is "cancel-unsafe". See `attributes(5)` for a discussion of Cancel-Safety, Deferred-Cancel-Safety, and Asynchronous-Cancel-Safety.

Name ptrace – allows a parent process to control the execution of a child process

Synopsis

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

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

Description The `ptrace()` function allows a parent process to control the execution of a child process. Its primary use is for the implementation of breakpoint debugging. The child process behaves normally until it encounters a signal (see `signal.h(3HEAD)`), at which time it enters a stopped state and its parent is notified by the `wait(3C)` function. When the child is in the stopped state, its parent can examine and modify its "core image" using `ptrace()`. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

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

`0` This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state on receipt of a signal rather than the state specified by *func* (see `signal(3C)`). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results ensue if the parent does not expect to trace the child.

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

`1, 2` With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If instruction and data space are separated, request `1` returns a word from instruction space, and request `2` returns a word from data space. If instruction and data space are not separated, either request `1` or request `2` may be used with equal results. The *data* argument is ignored. These two requests fail if *addr* is not the start address of a word, in which case `−1` is returned to the parent process and the parent's `errno` is set to `EIO`.

`3` With this request, the word at location *addr* in the child's user area in the system's address space (see `<sys/user.h>`) is returned to the parent process. The *data* argument is ignored. This request fails if *addr* is not the start address of a word or is outside the user area, in which case `−1` is returned to the parent process and the parent's `errno` is set to `EIO`.

`4, 5` With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If instruction and data space are separated, request `4` writes a word into instruction space, and request `5` writes a word into data space. If instruction and data space are not separated, either request `4` or request `5` may be used with equal results. On success, the value written into the address space of the

child is returned to the parent. These two requests fail if *addr* is not the start address of a word. On failure −1 is returned to the parent process and the parent's `errno` is set to `EIO`.

`6` With this request, a few entries in the child's user area can be written. *data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are the general registers and the condition codes of the Processor Status Word.

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

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

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

To forestall possible fraud, `ptrace()` inhibits the set-user-ID facility on subsequent calls to one of the `exec` family of functions (see `exec(2)`). If a traced process calls one of these functions, it stops before executing the first instruction of the new image showing signal `SIGTRAP`.

Errors The `ptrace()` function will fail if:

`EIO` The *request* argument is an illegal number.

`EPERM` The calling process does not have appropriate privileges to control the calling process. See `proc(4)`.

`ESRCH` The *pid* argument identifies a child that does not exist or has not executed a `ptrace()` call with request `0`.

Usage The `ptrace()` function is available only with the 32-bit version of `libc(3LIB)`. It is not available with the 64-bit version of this library.

The `/proc` debugging interfaces should be used instead of `ptrace()`, which provides quite limited debugger support and is itself implemented using the `/proc` interfaces. There is no actual `ptrace()` system call in the kernel. See `proc(4)` for descriptions of the `/proc` debugging interfaces.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also exec(2), exit(2), libc(3LIB), signal(3C), signal.h(3HEAD), wait(3C), proc(4), attributes(5)

Name ptsname – get name of the slave pseudo-terminal device

Synopsis

```
#include <stdlib.h>

char *ptsname(int fildes);
```

Description The `ptsname()` function returns the name of the slave pseudo-terminal device associated with a master pseudo-terminal device. *fildes* is a file descriptor returned from a successful open of the master device. `ptsname()` returns a pointer to a string containing the null-terminated path name of the slave device of the form `/dev/pts/N`, where `N` is a non-negative integer.

Return Values Upon successful completion, the function `ptsname()` returns a pointer to a string which is the name of the pseudo-terminal slave device. This value points to a static data area that is overwritten by each call to `ptsname()`. Upon failure, `ptsname()` returns `NULL`. This could occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `open(2)`, `grantpt(3C)`, `ttyname(3C)`, `unlockpt(3C)`, `attributes(5)`, `standards(5)`

STREAMS Programming Guide

Name putenv – change or add value to environment

Synopsis

```
#include <stdlib.h>

int putenv(char *string);
```

Description The `putenv()` function makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment.

The *string* argument points to a string of the form *name*=*value*. The space used by *string* is no longer used once a new string-defining *name* is passed to `putenv()`.

The `putenv()` function uses `malloc(3C)` to enlarge the environment.

After `putenv()` is called, environment variables are not in alphabetical order.

Return Values Upon successful completion, `putenv()` returns 0. Otherwise, it returns a non-zero value and sets `errno` to indicate the error.

Errors The `putenv()` function may fail if:

`ENOMEM` Insufficient memory was available.

Usage The `putenv()` function can be safely called from multithreaded programs. Caution must be exercised when using this function and `getenv(3C)` in multithreaded programs. These functions examine and modify the environment list, which is shared by all threads in a program. The system prevents the list from being accessed simultaneously by two different threads. It does not, however, prevent two threads from successively accessing the environment list using `putenv()` or `getenv()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `exec(2)`, `getenv(3C)`, `malloc(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Warnings The *string* argument should not be an automatic variable. It should be declared static if it is declared within a function because it cannot be automatically declared. A potential error is to call `putenv()` with a pointer to an automatic variable as the argument and to then exit the calling function while *string* is still part of the environment.

Name putpwent – write password file entry

Synopsis

```
#include <pwd.h>

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

Description The `putpwent()` function is the inverse of `getpwent()`. See `getpwnam(3C)`. Given a pointer to a `passwd` structure created by `getpwent()`, `getpwuid()`, or `getpwnam()`, `putpwent()` writes a line on the stream *f* that matches the format of `/etc/passwd`.

Return Values The `putpwent()` function returns a non-zero value if an error was detected during its operation. Otherwise, it returns `0`.

Usage The `putpwent()` function is of limited utility, since most password files are maintained as Network Information Service (NIS) files that cannot be updated with this function. For this reason, the use of this function is discouraged. If used at all, it should be used with `putspent(3C)` to update the shadow file.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `getpwnam(3C)`, `putspent(3C)`, `attributes(5)`

Name puts, fputs – put a string on a stream

Synopsis

```
#include <stdio.h>

int puts(const char *s);

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

Description The `puts()` function writes the string pointed to by *s*, followed by a `NEWLINE` character, to the standard output stream `stdout` (see `Intro(3)`). The terminating null byte is not written.

The `fputs()` function writes the null-terminated string pointed to by *s* to the named output *stream*. The terminating null byte is not written.

The `st_ctime` and `st_mtime` fields of the file will be marked for update between the successful execution of `fputs()` and the next successful completion of a call to `fflush(3C)` or `fclose(3C)` on the same stream or a call to `exit(2)` or `abort(3C)`.

Return Values On successful completion, both functions return the number of bytes written; otherwise they return `EOF` and set `errno` to indicate the error.

Errors Refer to `fputc(3C)`.

Usage Unlike `puts()`, the `fputs()` function does not write a NEWLINE character at the end of the string.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `exit(2)`, `write(2)`, `Intro(3)`, `abort(3C)`, `fclose(3C)`, `ferror(3C)`, `fflush(3C)`, `fopen(3C)`, `fputc(3C)`, `printf(3C)`, `stdio(3C)`, `attributes(5)`, `standards(5)`

Name putspent – write shadow password file entry

Synopsis

```
#include <shadow.h>

int putspent(const struct spwd *p, FILE *fp);
```

Description The `putspent()` function is the inverse of `getspent()`. See `getspnam(3C)`. Given a pointer to a `spwd` structure created by `getspent()` or `getspnam()`, `putspent()` writes a line on the stream *fp* that matches the format of `/etc/shadow`.

The `spwd` structure contains the following members:

```
char           *sp_namp;
char           *sp_pwdp;
int            sp_lstchg;
int            sp_min;
int            sp_max;
int            sp_warn;
int            sp_inact;
int            sp_expire;
unsigned int   sp_flag;
```

If the `sp_min`, `sp_max`, `sp_lstchg`, `sp_warn`, `sp_inact`, or `sp_expire` member of the `spwd` structure is −1, or if `sp_flag` is 0, the corresponding `/etc/shadow` field is cleared.

Return Values The `putspent()` function returns a non-zero value if an error was detected during its operation. Otherwise, it returns `0`.

Usage Since this function is for internal use only, compatibility is not guaranteed. For this reason, its use is discouraged. If used at all, if should be used with `putpwent(3C)` to update the password file.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `getpwnam(3C)`, `getspnam(3C)`, `putpwent(3C)`, `attributes(5)`

Name putws – convert a string of Process Code characters to EUC characters

Synopsis

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

int putws(wchar_t *s);
```

Description The `putws()` function converts the Process Code string (terminated by a `(wchar_t)NULL`) pointed to by *s*, to an Extended Unix Code (EUC) string followed by a NEWLINE character, and writes it to the standard output stream `stdout`. It does not write the terminal null character.

Return Values The `putws()` function returns the number of Process Code characters transformed and written. It returns `EOF` if it attempts to write to a file that has not been opened for writing.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `ferror(3C)`, `fopen(3C)`, `fread(3C)`, `getws(3C)`, `printf(3C)`, `putwc(3C)`, `attributes(5)`

Name qsort – quick sort

Synopsis

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
     int (*compar)(const void *, const void *));
```

Description The `qsort()` function is an implementation of the quick-sort algorithm. It sorts a table of data in place. The contents of the table are sorted in ascending order according to the user-supplied comparison function.

The *base* argument points to the element at the base of the table. The *nel* argument is the number of elements in the table. The *width* argument specifies the size of each element in bytes. The *compar* argument is the name of the comparison function, which is called with two arguments that point to the elements being compared.

The function must return an integer less than, equal to, or greater than zero to indicate if the first argument is to be considered less than, equal to, or greater than the second argument.

The contents of the table are sorted in ascending order according to the user supplied comparison function.

Usage The `qsort()` function safely allows concurrent access by multiple threads to disjoint data, such as overlapping subtrees or tables.

Examples **EXAMPLE 1** Program sorts.

The following program sorts a simple array:

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

static int
intcompare(const void *p1, const void *p2)
{
    int i = *((int *)p1);
    int j = *((int *)p2);

    if (i > j)
        return (1);
    if (i < j)
        return (-1);
    return (0);
}

int
main()
{
    int i;
```

EXAMPLE 1 Program sorts. *(Continued)*

```
    int a[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
    size_t nelems = sizeof (a) / sizeof (int);

    qsort((void *)a, nelems, sizeof (int), intcompare);

    for (i = 0; i < nelems; i++) {
        (void) printf("%d ", a[i]);
    }

    (void) printf("\n");
    return (0);
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also

sort(1), bsearch(3C), lsearch(3C), string(3C), attributes(5), standards(5)

Notes

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The relative order in the output of two items that compare as equal is unpredictable.

Name raise – send a signal to the executing thread

Synopsis

```
#include <signal.h>

int raise(int sig);
```

Description The `raise()` function sends the signal *sig* to the executing thread. If a signal handler is called, the raise function does not return until after the signal handler returns.

The effect of the raise function is equivalent to calling:

```
pthread_kill(pthread_self(), sig);
```

See the `pthread_kill(3C)` manual page for a detailed list of failure conditions and the `signal.h(3HEAD)` manual page for a list of signals.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `pthread_kill(3C)`, `pthread_self(3C)`, `signal.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name rand, srand, rand_r – simple random-number generator

Synopsis

```
#include <stdlib.h>

int rand(void);

void srand(unsigned int seed);

int rand_r(unsigned int *seed);
```

Description The `rand()` function uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range of 0 to `RAND_MAX` (defined in `<stdlib.h>`).

The `srand()` function uses the argument *seed* as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand()`. If `srand()` is then called with the same *seed* value, the sequence of pseudo-random numbers will be repeated. If `rand()` is called before any calls to `srand()` have been made, the same sequence will be generated as when `srand()` is first called with a *seed* value of 1.

The `rand_r()` function has the same functionality as `rand()` except that a pointer to a seed *seed* must be supplied by the caller. If `rand_r()` is called with the same initial value for the object pointed to by seed and that object is not modified between successive calls to `rand_r()`, the same sequence as that produced by calls to `rand()` will be generated.

The `rand()` and `srand()` functions provide per-process pseudo-random streams shared by all threads. The same effect can be achieved if all threads call `rand_r()` with a pointer to the same seed object. The `rand_r()` function allows a thread to generate a private pseudo-random stream by having the seed object be private to the thread.

Usage The spectral properties of `rand()` are limited. The `drand48(3C)` function provides a better, more elaborate random-number generator.

When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should be used only in multithreaded applications.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `drand48(3C)`, `attributes(5)`, `standards(5)`

Name random, srandom, initstate, setstate – pseudorandom number functions

Synopsis

```
#include <stdlib.h>

long random(void);

void srandom(unsigned int seed);

char *initstate(unsigned int seed, char *state, size_t size);

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

Description The `random()` function uses a nonlinear additive feedback random-number generator employing a default state array size of 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random-number generator is approximately 16 x $(2^{31}-1)$. The size of the state array determines the period of the random-number generator. Increasing the state array size increases the period.

The `srandom()` function initializes the current state array using the value of *seed*.

The `random()` and `srandom()` functions have (almost) the same calling sequence and initialization properties as `rand()` and `srand()` (see rand(3C)). The difference is that rand(3C) produces a much less random sequence—in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by `random()` are usable.

The algorithm from `rand()` is used by `srandom()` to generate the 31 state integers. Because of this, different `srandom()` seeds often produce, within an offset, the same sequence of low order bits from `random()`. If low order bits are used directly, `random()` should be initialized with `setstate()` using high quality random values.

Unlike `srand()`, `srandom()` does not return the old seed because the amount of state information used is much more than a single word. Two other routines are provided to deal with restarting/changing random number generators. With 256 bytes of state information, the period of the random-number generator is greater than 2^{69}, which should be sufficient for most purposes.

Like rand(3C), `random()` produces by default a sequence of numbers that can be duplicated by calling `srandom()` with 1 as the seed.

The `initstate()` and `setstate()` functions handle restarting and changing random-number generators. The `initstate()` function allows a state array, pointed to by the *state* argument, to be initialized for future use. The `size` argument, which specifies the size in bytes of the state array, is used by `initstate()` to decide what type of random-number generator to use; the larger the state array, the more random the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one of these values. For values smaller than 8, `random()` uses a simple linear congruential random number generator. The *seed* argument specifies a starting point for the random-number sequence and provides for restarting at the same point. The `initstate()` function returns a pointer to the previous state information array.

If `initstate()` has not been called, then `random()` behaves as though `initstate()` had been called with *seed* = 1 and *size* = 128.

If `initstate()` is called with *size* < 8, then `random()` uses a simple linear congruential random number generator.

Once a state has been initialized, `setstate()` allows switching between state arrays. The array defined by the *state* argument is used for further random-number generation until `initstate()` is called or `setstate()` is called again. The `setstate()` function returns a pointer to the previous state array.

Return Values The `random()` function returns the generated pseudo-random number.

The `srandom()` function returns no value.

Upon successful completion, `initstate()` and `setstate()` return a pointer to the previous state array. Otherwise, a null pointer is returned.

Errors No errors are defined.

Usage After initialization, a state array can be restarted at a different point in one of two ways:

- The `initstate()` function can be used, with the desired seed, state array, and size of the array.
- The `setstate()` function, with the desired state, can be used, followed by `srandom()` with the desired seed. The advantage of using both of these functions is that the size of the state array does not have to be saved once it is initialized.

Examples **EXAMPLE 1** Initialize an array.

The following example demonstrates the use of `initstate()` to intialize an array. It also demonstrates how to initialize an array and pass it to `setstate()`.

```
# include <stdlib.h>
static unsigned int state0[32];
static unsigned int state1[32] = {
     3,
     0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
     0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd,
     0x8c2e680f, 0xeb3d799f, 0xb11ee0b7, 0x2d436b86,
     0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
     0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc,
     0xde3b81e0, 0xdf0a6fb5, 0xf103bc02, 0x48f340fb,
     0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
     0xf5ad9d0e, 0x8999220b, 0x27fb47b9
     };
main() {
     unsigned seed;
     int n;
```

EXAMPLE 1 Initialize an array. *(Continued)*

```
    seed = 1;
    n = 128;
    (void)initstate(seed, (char *)state0, n);
    printf("random() = %d0\n", random());
    (void)setstate((char *)state1);
    printf("random() = %d0\n", random());
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See NOTES below.
Standard	See standards(5).

See Also

drand48(3C), rand(3C), attributes(5), standards(5)

Notes

The random() and srandom() functions are unsafe in multithreaded applications.

Use of these functions in multithreaded applications is unsupported.

For initstate() and setstate(), the *state* argument must be aligned on an int boundary.

Newer and better performing random number generators such as addrans() and lcrans() are available with the SUNWspro package.

Name rctlblk_set_value, rctlblk_get_firing_time, rctlblk_get_global_action, rctlblk_get_global_flags, rctlblk_get_local_action, rctlblk_get_local_flags, rctlblk_get_privilege, rctlblk_get_recipient_pid, rctlblk_get_value, rctlblk_get_enforced_value, rctlblk_set_local_action, rctlblk_set_local_flags, rctlblk_set_privilege, rctlblk_set_recipient_pid, rctlblk_size – manipulate resource control blocks

Synopsis

```
#include <rctl.h>

hrtime_t  rctlblk_get_firing_time(rctlblk_t *rblk);

int rctlblk_get_global_action(rctlblk_t *rblk);

int rctlblk_get_global_flags(rctlblk_t *rblk);

int rctlblk_get_local_action(rctlblk_t *rblk, int *signalp);

int rctlblk_get_local_flags(rctlblk_t *rblk);

rctl_priv_t  rctlblk_get_privilege(rctlblk_t *rblk);

id_t rctlblk_get_recipient_pid(rctlblk_t *rblk);

rctl_qty_t  rctlblk_get_value(rctlblk_t *rblk);

rctl_qty_t  rctlblk_get_enforced_value(rctlblk_t *rblk);

void rctlblk_set_local_action(rctlblk_t *rblk, rctl_action_t action,
     int signal);

void rctlblk_set_local_flags(rctlblk_t *rblk, int flags);

void rctlblk_set_privilege(rctlblk_t *rblk, rctl_priv_t privilege);

void rctlblk_set_value(rctlblk_t *rblk, rctl_qty_t value);

void  rctlblk_set_recipient_pid(id_tpid);

size_t rctlblk_size(void);
```

Description The resource control block routines allow the establishment or retrieval of values from a resource control block used to transfer information using the `getrctl(2)` and `setrctl(2)` functions. Each of the routines accesses or sets the resource control block member corresponding to its name. Certain of these members are read-only and do not possess set routines.

The firing time of a resource control block is 0 if the resource control action-value has not been exceeded for its lifetime on the process. Otherwise the firing time is the value of `gethrtime(3C)` at the moment the action on the resource control value was taken.

The global actions and flags are the action and flags set by `rctladm(1M)`. These values cannot be set with `setrctl(2)`. Valid global actions are listed in the table below. Global flags are generally a published property of the control and are not modifiable.

RCTL_GLOBAL_DENY_ALWAYS	The action taken when a control value is exceeded on this control will always include denial of the resource.
RCTL_GLOBAL_DENY_NEVER	The action taken when a control value is exceeded on this control will always exclude denial of the resource; the resource will always be granted, although other actions can also be taken.
RCTL_GLOBAL_SIGNAL_NEVER	No signal actions are permitted on this control.
RCTL_GLOBAL_CPU_TIME	The valid signals available as local actions include the SIGXCPU signal.
RCTL_GLOBAL_FILE_SIZE	The valid signals available as local actions include the SIGXFSZ signal.
RCTL_GLOBAL_INFINITE	This resource control supports the concept of an unlimited value; generally true only of accumulation-oriented resources, such as CPU time.
RCTL_GLOBAL_LOWERABLE	Non-privileged callers are able to lower the value of privileged resource control values on this control.
RCTL_GLOBAL_NOACTION	No global action will be taken when a resource control value is exceeded on this control.
RCTL_GLOBAL_NOBASIC	No values with the RCPRIV_BASIC privilege are permitted on this control.
RCTL_GLOBAL_SYSLOG	A standard message will be logged by the syslog(3C) facility when any resource control value on a sequence associated with this control is exceeded.
RCTL_GLOBAL_SYSLOG_NEVER	The resource control does not support the syslog() global action. Exceeding a resource control value on this control will not result in a message logged by the syslog() facility.
RCTL_GLOBAL_UNOBSERVABLE	The resource control (generally on a task- or project-related control) does not support observational control values. An RCPRIV_BASIC privileged control value placed by a process on the task or process will generate an action only if the value is exceeded by that process.
RCTL_GLOBAL_BYTES	This resource control represents a number of bytes.
RCTL_GLOBAL_SECONDS	This resource control represents a quantity of time in seconds.
RCTL_GLOBAL_COUNT	This resource control represents an integer count.

The local action and flags are those on the current resource control value represented by this resource control block. Valid actions and flags are listed in the table below. In the case of `RCTL_LOCAL_SIGNAL`, the second argument to `rctlblk_set_local_action()` contains the signal to be sent. Similarly, the signal to be sent is copied into the integer location specified by the second argument to `rctlblk_get_local_action()`. A restricted set of signals is made available for normal use by the resource control facility: `SIGBART`, `SIGXRES`, `SIGHUP`, `SIGSTOP`, `SIGTERM`, and `SIGKILL`. Other signals are permitted due to global properties of a specific control. Calls to `setrctl()` with illegal signals will fail.

`RCTL_LOCAL_DENY`	When this resource control value is encountered, the request for the resource will be denied. Set on all values if `RCTL_GLOBAL_DENY_ALWAYS` is set for this control; cleared on all values if `RCTL_GLOBAL_DENY_NEVER` is set for this control.
`RCTL_LOCAL_MAXIMAL`	This resource control value represents a request for the maximum amount of resource for this control. If `RCTL_GLOBAL_INFINITE` is set for this resource control, `RCTL_LOCAL_MAXIMAL` indicates an unlimited resource control value, one that will never be exceeded.
`RCTL_LOCAL_NOACTION`	No local action will be taken when this resource control value is exceeded.
`RCTL_LOCAL_SIGNAL`	The specified signal, sent by `rctlblk_set_local_action()`, will be sent to the process that placed this resource control value in the value sequence. This behavior is also true for signal actions on project and task resource controls. The specified signal is sent only to the recipient process, not all processes within the project or task.

The `rctlblk_get_recipient_pid()` function returns the value of the process ID that placed the resource control value for basic rctls. For privileged or system rctls, `rctlblk_get_recipient_pid()` returns -1.

The `rctlblk_set_recipient_pid()` function sets the recipient *pid* for a basic rctl. When `setrctl(2)` is called with the flag `RCTL_USE_RECIPIENT_PID`, this *pid* is used. Otherwise, the PID of the calling process is used. Only privileged users can set the recipient PID to one other than the PID of the calling process. Process-scoped rctls must have a recipient PID that matches the PID of the calling process.

The `rctlblk_get_privilege()` function returns the privilege of the resource control block. Valid privileges are `RCPRIV_BASIC`, `RCPRIV_PRIVILEGED`, and `RCPRIV_SYSTEM`. System resource controls are read-only. Privileged resource controls require the {`PRIV_SYS_RESOURCE`} privilege to write, unless the `RCTL_GLOBAL_LOWERABLE` global flag is set, in which case unprivileged applications can lower the value of a privileged control.

The `rctlblk_get_value()` and `rctlblk_set_value()` functions return or establish the enforced value associated with the resource control. In cases where the process, task, or project associated with the control possesses fewer capabilities than allowable by the current value, the value returned by `rctlblk_get_enforced_value()` will differ from that returned by `rctlblk_get_value()`. This capability difference arises with processes using an address space model smaller than the maximum address space model supported by the system.

The `rctlblk_size()` function returns the size of a resource control block for use in memory allocation. The `rctlblk_t *` type is an opaque pointer whose size is not connected with that of the resource control block itself. Use of `rctlblk_size()` is illustrated in the example below.

Return Values The various set routines have no return values. Incorrectly composed resource control blocks will generate errors when used with `setrctl(2)` or `getrctl(2)`.

Errors No error values are returned. Incorrectly constructed resource control blocks will be rejected by the system calls.

Examples **EXAMPLE 1** Display the contents of a fetched resource control block.

The following example displays the contents of a fetched resource control block.

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

rctlblk_t *rblk;
int rsignal;
int raction;

if ((rblk = malloc(rctlblk_size())) == NULL) {
       (void) perror("rblk malloc");
       exit(1);
}

if (getrctl("process.max-cpu-time", NULL, rblk, RCTL_FIRST) == -1) {
       (void) perror("getrctl");
       exit(1);
}

main()
{
    raction = rctlblk_get_local_action(rblk, &rsignal),
    (void) printf("Resource control for %s\n",
        "process.max-cpu-time");
    (void) printf("Process ID:     %d\n",
        rctlblk_get_recipient_pid(rblk));
    (void) printf("Privilege:      %x\n"
        rctlblk_get_privilege(rblk));
```

EXAMPLE 1 Display the contents of a fetched resource control block. *(Continued)*

```
    (void) printf("Global flags:   %x\n"
        rctlblk_get_global_flags(rblk));
    (void) printf("Global actions: %x\n"
        rctlblk_get_global_action(rblk));
    (void) printf("Local flags:    %x\n"
        rctlblk_get_local_flags(rblk));
    (void) printf("Local action:   %x (%d)\n"
        raction, raction == RCTL_LOCAL_SIGNAL ? rsignal : 0);
    (void) printf("Value:          %llu\n",
        rctlblk_get_value(rblk));
    (void) printf("\\tEnforced value: %llu\n",
        rctlblk_get_enforced_value(rblk));
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

rctladm(1M), getrctl(2), setrctl(2), gethrtime(3C), attributes(5)

Name rctl_walk – visit registered rctls on current system

Synopsis

```
#include <rctl.h>

int rctl_walk(int (*callback)(const char *rctlname, void *walk_data),
     void *init_data);
```

Description The `rctl_walk()` function provides a mechanism for the application author to examine all active resource controls (rctls) on the current system. The *callback* function provided by the application is given the name of an rctl at each invocation and can use the *walk_data* to record its own state. The callback function should return non-zero if it encounters an error condition or attempts to terminate the walk prematurely; otherwise the callback function should return 0.

Return Values Upon successful completion, `rctl_walk()` returns 0. It returns −1 if the *callback* function returned a non-zero value or if the walk encountered an error, in which case `errno` is set to indicate the error.

Errors The `rctl_walk()` function will fail if:

`ENOMEM` There is insufficient memory available to set up the initial data for the walk.

Other returned error values are presumably caused by the *callback* function.

Examples **EXAMPLE 1** Count the number of rctls available on the system.

The following example counts the number of resource controls on the system.

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

typedef struct wdata {
     uint_t count;
} wdata_t;

wdata_t total_count;

int
simple_callback(const char *name, void *pvt)
{
     wdata_t *w = (wdata_t *)pvt;
     w->count++;
     return (0);
}

...

total_count.count = 0;
```

EXAMPLE 1 Count the number of rctls available on the system. *(Continued)*

```
errno = 0;
if (rctl_walk(simple_callback, &total_count)) == 0)
     (void) printf("count = %u\n", total_count.count);
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

setrctl(2), attributes(5)

Name readdir, readdir_r – read directory

Synopsis

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

struct dirent *readdir(DIR *dirp);

struct dirent *readdir_r(DIR *dirp, struct dirent *entry);
```

Standard conforming

```
cc [ flag... ] file... -D_POSIX_PTHREAD_SEMANTICS [ library... ]

int readdir_r(DIR *restrict dirp, struct dirent *restrict entry,
     struct dirent **restrict result);
```

Description The type DIR, which is defined in the header <dirent.h>, represents a *directory stream*, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files. Files can be removed from a directory or added to a directory asynchronously to the operation of readdir() and readdir_r().

readdir() The readdir() function returns a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument *dirp*, and positions the directory stream at the next entry. It returns a null pointer upon reaching the end of the directory stream. The structure dirent defined by the <dirent.h> header describes a directory entry.

The readdir() function will not return directory entries containing empty names. If entries for . (dot) or .. (dot-dot) exist, one entry will be returned for dot and one entry will be returned for dot-dot; otherwise they will not be returned.

The pointer returned by readdir() points to data that can be overwritten by another call to readdir() on the same directory stream. These data are not overwritten by another call to readdir() on a different directory stream.

If a file is removed from or added to the directory after the most recent call to opendir(3C) or rewinddir(3C), whether a subsequent call to readdir() returns an entry for that file is unspecified.

The readdir() function can buffer several directory entries per actual read operation. It marks for update the st_atime field of the directory each time the directory is actually read.

After a call to fork(2), either the parent or child (but not both) can continue processing the directory stream using readdir(), rewinddir() or seekdir(3C). If both the parent and child processes use these functions, the result is undefined.

If the entry names a symbolic link, the value of the d_ino member is unspecified.

`readdir_r()` Unless the end of the directory stream has been reached or an error occurred, the `readdir_r()` function initializes the `dirent` structure referenced by `entry` to represent the directory entry at the current position in the directory stream referred to by *dirp*, and positions the directory stream at the next entry.

The caller must allocate storage pointed to by *entry* to be large enough for a `dirent` structure with an array of `char d_name` member containing at least `NAME_MAX` (that is, `pathconf(directory, _PC_NAME_MAX)`) plus one elements. (`_PC_NAME_MAX` is defined in `<unistd.h>`.)

The `readdir_r()` function will not return directory entries containing empty names. It is unspecified whether entries are returned for . (dot) or .. (dot-dot).

If a file is removed from or added to the directory after the most recent call to `opendir()` or `rewinddir()`, whether a subsequent call to `readdir_r()` returns an entry for that file is unspecified.

The `readdir_r()` function can buffer several directory entries per actual read operation. It marks for update the `st_atime` field of the directory each time the directory is actually read.

The standard-conforming version (see `standards(5)`) of the `readdir_r()` function performs all of the actions described above and sets the pointer pointed to by *result*. If a directory entry is returned, the pointer will be set to the same value as the *entry* argument; otherwise, it will be set to `NULL`.

Return Values Upon successful completion, `readdir()` and the default `readdir_r()` return a pointer to an object of type `struct dirent`. When an error is encountered, a null pointer is returned and `errno` is set to indicate the error. When the end of the directory is encountered, a null pointer is returned and `errno` is not changed.

The standard-conforming `readdir_r()` returns `0` if the end of the directory is encountered or a directory entry is stored in the structure referenced by *entry*. Otherwise, an error number is returned to indicate the failure.

Errors The `readdir()` and `readdir_r()` functions will fail if:

`EOVERFLOW` One of the values in the structure to be returned cannot be represented correctly.

The `readdir()` and `readdir_r()` functions may fail if:

`EBADF` The *dirp* argument does not refer to an open directory stream.

`ENOENT` The current position of the directory stream is invalid.

Usage The `readdir()` and `readdir_r()` functions should be used in conjunction with `opendir()`, `closedir()`, and `rewinddir()` to examine the contents of the directory. Since `readdir()` and the default `readdir_r()` return a null pointer both at the end of the directory and on error, an

application wanting to check for error situations should set `errno` to 0 before calling either of these functions. If `errno` is set to non-zero on return, an error occurred.

It is safe to use `readdir()` in a threaded application, so long as only one thread reads from the directory stream at any given time. The `readdir()` function is generally preferred over the `readdir_r()` function.

The standard-conforming `readdir_r()` returns the error number if an error occurred. It returns 0 on success (including reaching the end of the directory stream).

The `readdir()` and `readdir_r()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

Examples

EXAMPLE 1 Search the current directory for the entry *name*.

The following sample program will search the current directory for each of the arguments supplied on the command line:

```
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <strings.h>

static void lookup(const char *arg)
{
        DIR *dirp;
        struct dirent *dp;

        if ((dirp = opendir(".")) == NULL) {
                perror("couldn't open '.'");
                return;
        }

        do {
                errno = 0;
                if ((dp = readdir(dirp)) != NULL) {
                        if (strcmp(dp->d_name, arg) != 0)
                                continue;

                        (void) printf("found %s\n", arg);
                        (void) closedir(dirp);
                        return;
                }
        } while (dp != NULL);

        if (errno != 0)
                perror("error reading directory");
```

EXAMPLE 1 Search the current directory for the entry *name.* *(Continued)*

```
        else
                (void) printf("failed to find %s\n", arg);
        (void) closedir(dirp);
        return;
}

int main(int argc, char *argv[])
{
        int i;
        for (i = 1; i < argc; i++)
                lookup(argv[i]);
        return (0);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See below.
Standard	See `standards(5)`.

The `readdir()` function is Unsafe. The `readdir_r()` function is Safe.

See Also

`fork(2)`, `lstat(2)`, `symlink(2)`, `Intro(3)`, `closedir(3C)`, `opendir(3C)`, `rewinddir(3C)`, `scandir(3C)`, `seekdir(3C)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Notes

When compiling multithreaded programs, see the `MULTITHREADED APPLICATIONS` section of `Intro(3)`.

Solaris 2.4 and earlier releases provided a `readdir_r()` interface as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface as described above. Support for the Draft 6 interface is provided for compatibility only and might not be supported in future releases. New applications and libraries should use the standard-conforming interface.

For POSIX.1c-conforming applications, the `_POSIX_PTHREAD_SEMANTICS` and `_REENTRANT` flags are automatically turned on by defining the `_POSIX_C_SOURCE` flag with a value >= 199506L.

Name realpath, canonicalize_file_name – resolve pathname

Synopsis

```
#include <stdlib.h>

char *realpath(const char *restrict file_name,
     char *restrict resolved_name);

char *canonicalize_file_name (const char *path);
```

Description The `realpath()` function derives, from the pathname pointed to by *file_name*, an absolute pathname that resolves to the same directory entry, whose resolution does not involve " . ", " . . ", or symbolic links. If *resolved_name* is not null, the generated pathname is stored as a null-terminated string, up to a maximum of {`PATH_MAX`} (defined in `limits.h`(3HEAD)) bytes in the buffer pointed to by *resolved_name*. If *resolved_name* is null, the generated pathname is stored as a null-terminated string in a buffer that is allocated as if `malloc`(3C) were called.

The call `canonicalize_file_name`(*path*) is equivalent to the call `realpath`(*path*, `NULL`).

Return Values On successful completion, `realpath()` returns a pointer to the resolved name. Otherwise, `realpath()` returns a null pointer and sets `errno` to indicate the error, and the contents of the buffer pointed to by *resolved_name* are left in an indeterminate state.

Errors The `realpath()` function will fail if:

`EACCES`	Read or search permission was denied for a component of *file_name*.
`EINVAL`	Either the *file_name* or *resolved_name* argument is a null pointer.
`EIO`	An error occurred while reading from the file system.
`ELOOP`	Too many symbolic links were encountered in resolving *file_name*.
`ELOOP`	A loop exists in symbolic links encountered during resolution of the *file_name* argument.
`ENAMETOOLONG`	The *file_name* argument is longer than {`PATH_MAX`} or a pathname component is longer than {`NAME_MAX`}.
`ENOENT`	A component of *file_name* does not name an existing file or *file_name* points to an empty string.
`ENOTDIR`	A component of the path prefix is not a directory.

The `realpath()` function may fail if:

`ENAMETOOLONG`	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {`PATH_MAX`}.
`ENOMEM`	Insufficient storage space is available.

Usage The `realpath()` function operates on null-terminated strings.

Execute permission is required for all the directories in the given and the resolved path.

The `realpath()` function might fail to return to the current directory if an error occurs.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `getcwd(3C)`, `limits.h(3HEAD)`, `malloc(3C)`, `sysconf(3C)`, `attributes(5)`, `standards(5)`

Name reboot – reboot system or halt processor

Synopsis

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

int reboot(int howto, char *bootargs);
```

Description The `reboot()` function reboots the system. The *howto* argument specifies the behavior of the system while rebooting and is a mask constructed by a bitwise-inclusive-OR of flags from the following list:

`RB_AUTOBOOT` The machine is rebooted from the root filesystem on the default boot device. This is the default behavior. See `boot(1M)` and `kernel(1M)`.

`RB_HALT` The processor is simply halted; no reboot takes place. This option should be used with caution.

`RB_ASKNAME` Interpreted by the bootstrap program and kernel, causing the user to be asked for pathnames during the bootstrap.

`RB_DUMP` The system is forced to panic immediately without any further processing and a crash dump is written to the dump device (see `dumpadm(1M)`) before rebooting.

Any other *howto* argument causes the kernel file to boot.

The interpretation of the *bootargs* argument is platform-dependent.

Return Values Upon successful completion, `reboot()` never returns. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `reboot()` function will fail if:

`EPERM` The {`PRIV_SYS_CONFIG`} privilege is not asserted in the effective set of the calling process.

See Also `Intro(1M)`, `boot(1M)`, `dumpadm(1M)`, `halt(1M)`, `init(1M)`, `kernel(1M)`, `reboot(1M)`, `uadmin(2)`

Name re_comp, re_exec – compile and execute regular expressions

Synopsis

```
#include <re_comp.h>

char *re_comp(const char *string);

int re_exec(const char *string);
```

Description The `re_comp()` function converts a regular expression string (RE) into an internal form suitable for pattern matching. The `re_exec()` function compares the string pointed to by the *string* argument with the last regular expression passed to `re_comp()`.

If `re_comp()` is called with a null pointer argument, the current regular expression remains unchanged.

Strings passed to both `re_comp()` and `re_exec()` must be terminated by a null byte, and may include NEWLINE characters.

The `re_comp()` and `re_exec()` functions support *simple regular expressions*, which are defined on the regexp(5) manual page. The regular expressions of the form `\{m\}`, `\{m,\}`, or `\{m,n\}` are not supported.

Return Values The `re_comp()` function returns a null pointer when the string pointed to by the *string* argument is successfully converted. Otherwise, a pointer to one of the following error message strings is returned:

```
No previous regular expression
Regular expression too long
unmatched \ (
missing ]
too many \ ( \ ) pairs
unmatched \ )
```

Upon successful completion, `re_exec()` returns `1` if *string* matches the last compiled regular expression. Otherwise, `re_exec()` returns `0` if *string* fails to match the last compiled regular expression, and `−1` if the compiled regular expression is invalid (indicating an internal error).

Errors No errors are defined.

Usage For portability to implementations conforming to X/Open standards prior to SUS, regcomp(3C) and regexec(3C) are preferred to these functions. See standards(5).

See Also grep(1), regcmp(1), regcmp(3C), regcomp(3C), regexec(3C), regexpr(3GEN), regexp(5), standards(5)

Name regcmp, regex – compile and execute regular expression

Synopsis

```
#include <libgen.h>

char *regcmp(const char *string1, /* char *string2 */ ...,
     int /*(char*)0*/);

char *regex(const char *re, const char *subject,
     /* char *ret0 */ ...);

extern char *__loc1;
```

Description The `regcmp()` function compiles a regular expression (consisting of the concatenated arguments) and returns a pointer to the compiled form. The `malloc(3C)` function is used to create space for the compiled form. It is the user's responsibility to free unneeded space so allocated. A `NULL` return from `regcmp()` indicates an incorrect argument. `regcmp(1)` has been written to generally preclude the need for this routine at execution time.

The `regex()` function executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. The `regex()` function returns `NULL` on failure or a pointer to the next unmatched character on success. A global character pointer `__loc1` points to where the match began. The `regcmp()` and `regex()` functions were mostly borrowed from the editor `ed(1)`; however, the syntax and semantics have been changed slightly. The following are the valid symbols and associated meanings.

`[ ] * .^` This group of symbols retains its meaning as described on the `regexp(5)` manual page.

`$` Matches the end of the string; `\n` matches a newline.

– Within brackets the minus means *through*. For example, `[a–z]` is equivalent to `[abcd . . .xyz]`. The – can appear as itself only if used as the first or last character. For example, the character class expression `[]–]` matches the characters `]` and –.

`+` A regular expression followed by `+` means *one or more times*. For example, `[0–9]+` is equivalent to `[0–9][0–9]*`.

{*m*} {*m*,} {*m*,*u*} Integer values enclosed in `{ }` indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (that is, {*m*}), it indicates the exact number of times the regular expression is to be applied. The value {*m*`,`} is analogous to {*m*,*infinity*}. The plus (+) and star (*) operations are equivalent to `{1,}` and `{0,}` respectively.

`( ... )$`*n* The value of the enclosed regular expression is to be returned. The value will be stored in the (*n*+1)th argument following the subject argument. At

most, ten enclosed regular expressions are allowed. The `regex()` function makes its assignments unconditionally.

`( ... )` Parentheses are used for grouping. An operator, for example, `*`, `+`, `{ }`, can work on a single character or a regular expression enclosed in parentheses. For example, `(a*(cb+)*)$0`. By necessity, all the above defined symbols are special. They must, therefore, be escaped with a `\` (backslash) to be used as themselves.

Examples

EXAMPLE 1 Example matching a leading newline in the subject string.

The following example matches a leading newline in the subject string pointed at by cursor.

```
char *cursor, *newcursor, *ptr;
    . . .
newcursor = regex((ptr = regcmp("^\n", (char *)0)), cursor);
free(ptr);
```

The following example matches through the string `Testing3` and returns the address of the character after the last matched character (the "`4`"). The string `Testing3` is copied to the character array `ret0`.

```
char ret0[9];
char *newcursor, *name;
    . . .
name = regcmp("([A–Za–z][A–za–z0–9]{0,7})$0", (char *)0);
newcursor = regex(name, "012Testing345", ret0);
```

The following example applies a precompiled regular expression in `file.i` (see `regcmp(1)`) against *string*.

```
#include "file.i"
char *string, *newcursor;
    . . .
newcursor = regex(name, string);
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also

`ed(1)`, `regcmp(1)`, `malloc(3C)`, `attributes(5)`, `regexp(5)`

Notes

The user program may run out of memory if `regcmp()` is called iteratively without freeing the vectors no longer required.

When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.

Name regcomp, regexec, regerror, regfree – regular expression matching

Synopsis

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

int regcomp(regex_t *restrict preg, const char *restrict pattern,
     int cflags);

int regexec(const regex_t *restrict preg,
     const char *restrict string, size_t nmatch,
     regmatch_t pmatch[restrict], int eflags);

size_t regerror(int errcode, const regex_t *restrict preg,
     char *restrict errbuf, size_t errbuf_size);

void regfree(regex_t *preg);
```

Description These functions interpret *basic* and *extended* regular expressions (described on the regex(5) manual page).

The structure type `regex_t` contains at least the following member:

`size_t re_nsub` Number of parenthesised subexpressions.

The structure type `regmatch_t` contains at least the following members:

`regoff_t rm_so` Byte offset from start of *string* to start of substring.

`regoff_t rm_eo` Byte offset from start of *string* of the first character after the end of substring.

`regcomp()` The `regcomp()` function will compile the regular expression contained in the string pointed to by the *pattern* argument and place the results in the structure pointed to by *preg*. The *cflags* argument is the bitwise inclusive OR of zero or more of the following flags, which are defined in the header `<regex.h>`:

`REG_EXTENDED` Use Extended Regular Expressions.

`REG_ICASE` Ignore case in match.

`REG_NOSUB` Report only success/fail in `regexec()`.

`REG_NEWLINE` Change the handling of NEWLINE characters, as described in the text.

The default regular expression type for *pattern* is a Basic Regular Expression. The application can specify Extended Regular Expressions using the `REG_EXTENDED` *cflags* flag.

If the `REG_NOSUB` flag was not set in *cflags*, then `regcomp()` will set *re_nsub* to the number of parenthesised subexpressions (delimited by \(\) in basic regular expressions or () in extended regular expressions) found in *pattern*.

`regexec()` The `regexec()` function compares the null-terminated string specified by *string* with the compiled regular expression *preg* initialized by a previous call to `regcomp()`. The *eflags* argument is the bitwise inclusive OR of zero or more of the following flags, which are defined in the header <`regex.h`>:

`REG_NOTBOL` The first character of the string pointed to by *string* is not the beginning of the line. Therefore, the circumflex character (`^`), when taken as a special character, will not match the beginning of *string*.

`REG_NOTEOL` The last character of the string pointed to by *string* is not the end of the line. Therefore, the dollar sign (*$*), when taken as a special character, will not match the end of *string*.

If *nmatch* is zero or `REG_NOSUB` was set in the *cflags* argument to `regcomp()`, then `regexec()` will ignore the *pmatch* argument. Otherwise, the *pmatch* argument must point to an array with at least *nmatch* elements, and `regexec()` will fill in the elements of that array with offsets of the substrings of *string* that correspond to the parenthesised subexpressions of *pattern*: *pmatch*`[`*i*`]`.*rm_so* will be the byte offset of the beginning and *pmatch*`[`*i*`]`.*rm_eo* will be one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th matched open parenthesis, counting from 1.) Offsets in *pmatch*`[0]` identify the substring that corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch*`[`*nmatch*`−1]` will be filled with `−1`. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a subexpression), then `regexec()` will still do the match, but will record only the first *nmatch* substrings.

When matching a basic or extended regular expression, any given parenthesised subexpression of *pattern* might participate in the match of several different substrings of *string*, or it might not match any substring even though the pattern as a whole did match. The following rules are used to determine which substrings to report in *pmatch* when matching regular expressions:

1. If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in *pmatch*`[`*i*`]` will delimit the last such match.
2. If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in *pmatch*`[`*i*`]` will be `−1`. A subexpression does not participate in the match when:

   `*` or `\{\}` appears immediately after the subexpression in a basic regular expression, or `*`, `?`, or `{ }` appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched zero times)

   or

   `|` is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.

3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained within any other subexpression that is contained within *j*, and a match of subexpression *j* is reported in *pmatch*[*j*], then the match or non-match of subexpression *i* reported in *pmatch*[*i*] will be as described in 1. and 2. above, but within the substring reported in *pmatch*[*j*] rather than the whole string.
4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch*[*j*] are −1, then the pointers in *pmatch*[*i*] also will be −1.
5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch*[*i*] will be the byte offset of the character or NULL terminator immediately following the zero-length string.

If, when `regexec()` is called, the locale is different from when the regular expression was compiled, the result is undefined.

If `REG_NEWLINE` is not set in *cflags*, then a NEWLINE character in *pattern* or *string* will be treated as an ordinary character. If `REG_NEWLINE` is set, then newline will be treated as an ordinary character except as follows:

1. A NEWLINE character in *string* will not be matched by a period outside a bracket expression or by any form of a non-matching list.
2. A circumflex (^) in *pattern*, when used to specify expression anchoring will match the zero-length string immediately after a newline in *string*, regardless of the setting of `REG_NOTBOL`.
3. A dollar-sign ($) in *pattern*, when used to specify expression anchoring, will match the zero-length string immediately before a newline in *string*, regardless of the setting of `REG_NOTEOL`.

`regfree()` The `regfree()` function frees any memory allocated by `regcomp()` associated with *preg*.

The following constants are defined as error return values:

`REG_NOMATCH`	The `regexec()` function failed to match.
`REG_BADPAT`	Invalid regular expression.
`REG_ECOLLATE`	Invalid collating element referenced.
`REG_ECTYPE`	Invalid character class type referenced.
`REG_EESCAPE`	Trailing \ in pattern.
`REG_ESUBREG`	Number in *digit* invalid or in error.
`REG_EBRACK`	`[ ]` imbalance.
`REG_ENOSYS`	The function is not supported.

REG_EPAREN	\(\) or () imbalance.
REG_EBRACE	\{ \} imbalance.
REG_BADBR	Content of \{ \} invalid: not a number, number too large, more than two numbers, first larger than second.
REG_ERANGE	Invalid endpoint in range expression.
REG_ESPACE	Out of memory.
REG_BADRPT	?, * or + not preceded by valid regular expression.

`regerror()` The `regerror()` function provides a mapping from error codes returned by `regcomp()` and `regexec()` to unspecified printable strings. It generates a string corresponding to the value of the *errcode* argument, which must be the last non-zero value returned by `regcomp()` or `regexec()` with the given value of *preg*. If *errcode* is not such a value, an error message indicating that the error code is invalid is returned.

If *preg* is a `NULL` pointer, but *errcode* is a value returned by a previous call to `regexec()` or `regcomp()`, the `regerror()` still generates an error string corresponding to the value of *errcode*.

If the *errbuf_size* argument is not zero, `regerror()` will place the generated string into the buffer of size *errbuf_size* bytes pointed to by *errbuf*. If the string (including the terminating `NULL`) cannot fit in the buffer, `regerror()` will truncate the string and null-terminate the result.

If *errbuf_size* is zero, `regerror()` ignores the *errbuf* argument, and returns the size of the buffer needed to hold the generated string.

If the *preg* argument to `regexec()` or `regfree()` is not a compiled regular expression returned by `regcomp()`, the result is undefined. A *preg* is no longer treated as a compiled regular expression after it is given to `regfree()`.

See `regex(5)` for BRE (Basic Regular Expression) Anchoring.

Return Values On successful completion, the `regcomp()` function returns `0`. Otherwise, it returns an integer value indicating an error as described in <`regex.h`>, and the content of *preg* is undefined.

On successful completion, the `regexec()` function returns `0`. Otherwise it returns `REG_NOMATCH` to indicate no match, or `REG_ENOSYS` to indicate that the function is not supported.

Upon successful completion, the `regerror()` function returns the number of bytes needed to hold the entire generated string. Otherwise, it returns `0` to indicate that the function is not implemented.

The `regfree()` function returns no value.

Errors No errors are defined.

Usage An application could use:

```
regerror(code,preg,(char *)NULL,(size_t)0)
```

to find out how big a buffer is needed for the generated string, `malloc` a buffer to hold the string, and then call `regerror()` again to get the string (see `malloc(3C)`). Alternately, it could allocate a fixed, static buffer that is big enough to hold most strings, and then use `malloc()` to allocate a larger buffer if it finds that this is too small.

Examples **EXAMPLE 1** Example to match string against the extended regular expression in pattern.

```
#include <regex.h>
/*
* Match string against the extended regular expression in
* pattern, treating errors as no match.
*
* return 1 for match, 0 for no match
*/

int
match(const char *string, char *pattern)
{
      int status;
      regex_t re;
      if (regcomp(&re, pattern, REG_EXTENDED | REG_NOSUB) != 0) {
           return(0);      /* report error */
      }
      status = regexec(&re, string, (size_t) 0, NULL, 0);
      regfree(&re);
      if (status != 0) {
            return(0);      /* report error */
      }
      return(1);
}
```

The following demonstrates how the `REG_NOTBOL` flag could be used with `regexec()` to find all substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very little error checking is done.)

```
(void) regcomp (&re, pattern, 0);
/* this call to regexec( ) finds the first match on the line */
error = regexec (&re, &buffer[0], 1, &pm, 0);
while (error == 0) {    /* while matches found */
        /* substring found between pm.rm_so and pm.rm_eo */
        /* This call to regexec( ) finds the next match */
```

EXAMPLE 1 Example to match string against the extended regular expression in pattern. *(Continued)*

```
        error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also

`fnmatch(3C)`, `glob(3C)`, `malloc(3C)`, `setlocale(3C)`, `attributes(5)`, `standards(5)`, `regex(5)`

Notes

The `regcomp()` function can be used safely in a multithreaded application as long as `setlocale(3C)` is not being called to change the locale.

Name remove – remove file

Synopsis

```
#include <stdio.h>

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

Description The `remove()` function causes the file or empty directory whose name is the string pointed to by *path* to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless the file is created anew.

For files, `remove()` is identical to `unlink()`. For directories, `remove()` is identical to `rmdir()`.

See `rmdir(2)` and `unlink(2)` for a detailed list of failure conditions.

Return Values Upon successful completion, `remove()` returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate an error.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `rmdir(2)`, `unlink(2)`, `attributes(5)`, `standards(5)`

Name rewind – reset file position indicator in a stream

Synopsis

```
#include <stdio.h>

void rewind(FILE *stream);
```

Description The call:

```
rewind(stream)
```

is equivalent to:

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

except that `rewind()` also clears the error indicator.

Return Values The `rewind()` function returns no value.

Errors Refer to `fseek(3C)` with the exception of `EINVAL` which does not apply.

Usage Because `rewind()` does not return a value, an application wishing to detect errors should clear `errno`, then call `rewind()`, and if `errno` is non-zero, assume an error has occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `fseek(3C)`, `attributes(5)`, `standards(5)`

Name rewinddir – reset position of directory stream to the beginning of a directory

Synopsis

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

void rewinddir(DIR *dirp);
```

Description The `rewinddir()` function resets the position of the directory stream to which *dirp* refers to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to `opendir`(3C) would have done. If *dirp* does not refer to a directory stream, the effect is undefined.

After a call to the `fork`(2) function, either the parent or child (but not both) may continue processing the directory stream using `readdir`(3C), `rewinddir()` or `seekdir`(3C). If both the parent and child processes use these functions, the result is undefined.

Return Values The `rewinddir()` function does not return a value.

Errors No errors are defined.

Usage The `rewinddir()` function should be used in conjunction with `opendir()`, `readdir()`, and `closedir`(3C) to examine the contents of the directory. This method is recommended for portability.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards`(5).

See Also `fork`(2), `closedir`(3C), `opendir`(3C), `readdir`(3C), `seekdir`(3C), `attributes`(5), `standards`(5)

Name rwlock, rwlock_init, rwlock_destroy, rw_rdlock, rw_wrlock, rw_tryrdlock, rw_trywrlock, rw_unlock – multiple readers, single writer locks

Synopsis

```
cc –mt [ flag... ] file...[ library... ]

#include <synch.h>

int rwlock_init(rwlock_t *rwlp, int type, void * arg);

int rwlock_destroy(rwlock_t *rwlp);

int rw_rdlock(rwlock_t *rwlp);

int rw_wrlock(rwlock_t *rwlp);

int rw_unlock(rwlock_t *rwlp);

int rw_tryrdlock(rwlock_t *rwlp);

int rw_trywrlock(rwlock_t *rwlp);
```

Description Many threads can have simultaneous read-only access to data, while only one thread can have write access at any given time. Multiple read access with single write access is controlled by locks, which are generally used to protect data that is frequently searched.

Readers/writer locks can synchronize threads in this process and other processes if they are allocated in writable memory and shared among cooperating processes (see `mmap(2)`), and are initialized for this purpose.

Additionally, readers/writer locks must be initialized prior to use. `rwlock_init()` The readers/writer lock pointed to by *rwlp* is initialized by `rwlock_init()`. A readers/writer lock is capable of having several types of behavior, which is specified by `type`. *arg* is currently not used, although a future type may define new behavior parameters by way of *arg*.

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

`USYNC_PROCESS` The readers/writer lock can synchronize threads in this process and other processes. The readers/writer lock should be initialized by only one process. *arg* is ignored. A readers/writer lock initialized with this type, must be allocated in memory shared between processses, i.e. either in Sys V shared memory (see `shmop(2)`) or in memory mapped to a file (see `mmap(2)`). It is illegal to initialize the object this way and to not allocate it in such shared memory.

`USYNC_THREAD` The readers/writer lock can synchronize threads in this process, only. *arg* is ignored.

Additionally, readers/writer locks can be initialized by allocation in zeroed memory. A `type` of `USYNC_THREAD` is assumed in this case. Multiple threads must not simultaneously initialize the same readers/writer lock. And a readers/writer lock must not be re-initialized while in use by other threads.

The following are default readers/writer lock initialization (intra-process):

```
rwlock_t rwlp;
rwlock_init(&rwlp, NULL, NULL);
```

or

```
rwlock_init(&rwlp, USYNC_THREAD, NULL);
```

or

```
rwlock_t  rwlp  =  DEFAULTRWLOCK;
```

The following is a customized readers/writer lock initialization (inter-process):

```
rwlock_init(&rwlp, USYNC_PROCESS, NULL);
```

Any state associated with the readers/writer lock pointed to by *rwlp* are destroyed by `rwlock_destroy()` and the readers/writer lock storage space is not released.

`rw_rdlock()` gets a read lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is currently locked for writing, the calling thread blocks until the write lock is freed. Multiple threads may simultaneously hold a read lock on a readers/writer lock.

`rw_tryrdlock()` trys to get a read lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is locked for writing, it returns an error; otherwise, the read lock is acquired.

`rw_wrlock()` gets a write lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is currently locked for reading or writing, the calling thread blocks until all the read and write locks are freed. At any given time, only one thread may have a write lock on a readers/writer lock.

`rw_trywrlock()` trys to get a write lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is currently locked for reading or writing, it returns an error.

`rw_unlock()` unlocks a readers/writer lock pointed to by *rwlp*, if the readers/writer lock is locked and the calling thread holds the lock for either reading or writing. One of the other threads that is waiting for the readers/writer lock to be freed will be unblocked, provided there is other waiting threads. If the calling thread does not hold the lock for either reading or writing, no error status is returned, and the program's behavior is unknown.

Return Values If successful, these functions return `0`. Otherwise, a non-zero value is returned to indicate the error.

Errors The `rwlock_init()` function will fail if:

`EINVAL` `type` is invalid.

The `rw_tryrdlock()` or `rw_trywrlock()` functions will fail if:

`EBUSY` The reader or writer lock pointed to by *rwlp* was already locked.

These functions may fail if:

`EFAULT` *rwlp* or *arg* points to an illegal address.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `mmap(2)`, `attributes(5)`

Notes These interfaces also available by way of:

```
#include <thread.h>
```

If multiple threads are waiting for a readers/writer lock, the acquisition order is random by default. However, some implementations may bias acquisition order to avoid depriving writers. The current implementation favors writers over readers.

Name scandir, alphasort – scan a directory

Synopsis

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

int scandir(const char *dirname, struct dirent *(*namelist[]),
     int (*select)(const struct dirent *),
     int (*dcomp)(const struct dirent  **,
     const struct dirent **));

int alphasort(const struct dirent **d1,
     const struct dirent **d2);
```

Description The `scandir()` function reads the directory *dirname* using `readdir(3C)` and builds an array of pointers to directory entries using `malloc(3C)`. The *namelist* argument is a pointer to an array of structure pointers. The *select* argument is a pointer to a routine that is called with a pointer to a directory entry and returns a non-zero value if the directory entry is included in the array. If this pointer is `NULL`, then all the directory entries are included. The *dcomp* argument is a pointer to a routine that is passed to `qsort(3C)`, which sorts the completed array. If this pointer is `NULL`, the array is not sorted.

The `alphasort()` function can be used as the *dcomp*() function parameter for the `scandir()` function to sort the directory entries into alphabetical order, as if by the `strcoll(3C)` function. Its arguments are the two directory entries to compare.

Return Values The `scandir()` function returns the number of entries in the array and a pointer to the array through the *namelist* argument. When an error is encountered, `scandir()` returns -1 and `errno` is set to indicate the error.

The `alphasort()` function returns an integer greater than, equal to, or less than 0 if the directory entry name pointed to by *d1* is greater than, equal to, or less than the directory entry name pointed to by *d2* when both are interpreted as appropriate to the current locale. There is no return value reserved to indicate an error.

Errors The `scandir()` function will fail if:

`EOVERFLOW` The number of directory entries exceeds the number that can be represented by an `int`.

Usage The `scandir()` and `alphasort()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See below.

The `scandir()` function is Unsafe. The `alphasort()` function is Safe.

See Also `malloc`(3C), `qsort`(3C), `readdir`(3C), `strcoll`(3C), `attributes`(5), `lf64`(5)

Name scanf, fscanf, sscanf, vscanf, vfscanf, vsscanf – convert formatted input

Synopsis

```
#include <stdio.h>

int scanf(const char *restrict format...);

int fscanf(FILE *restrict stream, const char *restrict format...);

int sscanf(const char *restrict s, const char *restrict format...);

#include <stdarg.h>
#include <stdio.h>

int vscanf(const char *format, va_list arg);

int vfscanf(FILE *stream, const char *format, va_list arg);

int vsscanf(const char *s, const char *format, va_list arg);
```

Description The `scanf()` function reads from the standard input stream `stdin`.

The `fscanf()` function reads from the named input *stream*.

The `sscanf()` function reads from the string *s*.

The `vscanf()`, `vfscanf()`, and `vsscanf()` functions are equivalent to the `scanf()`, `fscanf()`, and `sscanf()` functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by the <`stdarg.h`> header . These functions do not invoke the `va_end()` macro. Applications using these functions should call `va_end`(*ap*) afterwards to clean up.

Each function reads bytes, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the *nth* argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion character `%` (see below) is replaced by the sequence `%`*n*`$`, where *n* is a decimal integer in the range [1, `NL_ARGMAX`]. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages. In format strings containing the `%`*n*`$` form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format string more than once.

The *format* can contain either form of a conversion specification, that is, `%` or `%`*n*`$`, but the two forms cannot normally be mixed within a single *format* string. The only exception to this is that `%%` or `%*` can be mixed with the `%`*n*`$` form.

The `scanf()` function in all its forms allows for detection of a language-dependent radix character in the input string. The radix character is defined in the program's locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

The format is a character string, beginning and ending in its initial shift state, if any, composed of zero or more directives. Each directive is composed of one of the following:

- one or more *white-space characters* (space, tab, newline, vertical-tab or form-feed characters);
- an *ordinary character* (neither `%` nor a white-space character); or
- a *conversion specification*.

Conversion Specifications

Each conversion specification is introduced by the character `%` or the character sequence `%`*n*`$`, after which the following appear in sequence:

- An optional assignment-suppressing character *.
- An optional non-zero decimal integer that specifies the maximum field width.
- An option length modifier that specifies the size of the receiving object.
- A conversion specifier character that specifies the type of conversion to be applied. The valid conversion characters are described below.

The `scanf()` functions execute each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space characters is executed by reading input until no more valid input can be read, or up to the first byte which is not a white-space character which remains unread.

A directive that is an ordinary character is executed as follows. The next byte is read from the input and compared with the byte that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent bytes remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification is executed in the following steps:

Input white-space characters (as specified by `isspace(3C)`) are skipped, unless the conversion specification includes a `[`, `c`, `C`, or `n` conversion character.

An item is read from the input unless the conversion specification includes an `n` conversion character. The length of the item read is limited to any specified maximum field width, which is interpreted in either characters or bytes depending on the conversion character. In Solaris default mode, the input item is defined as the longest sequence of input bytes that forms a

matching sequence. In some cases, `scanf()` might need to read several extra characters beyond the end of the input item to find the end of a matching sequence. In C99/SUSv3 mode, the input item is defined as the longest sequence of input bytes that is, or is a prefix of, a matching sequence. With this definition, `scanf()` need only read at most one character beyond the end of the input item. Therefore, in C99/SUSv3 mode, some sequences that are acceptable to `strtod`(3C), `strtol`(3C), and similar functions are unacceptable to `scanf()`. In either mode, `scanf()` attempts to push back any excess bytes read using `ungetc`(3C). Assuming all such attempts succeed, the first byte, if any, after the input item remains unread. If the length of the input item is 0, the conversion fails. This condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` conversion character, the input item (or, in the case of a `%`*n* conversion specification, the count of input bytes) is converted to a type appropriate to the conversion character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result if the conversion specification is introduced by `%`, or in the *n*th argument if introduced by the character sequence `%`*n*`$`. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

Length Modifiers

The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `signed char` or `unsigned char`.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `short` or `unsigned short`.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `long` or `unsigned long`; that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to `double`; or that a following `c`, `s`, or `[` conversion specifier applies to an argument with type pointer to `wchar_t`.

`ll` (ell-ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `long long` or `unsigned long long`.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `intmax_t` or `uintmax_t`.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `size_t` or the corresponding signed integer type.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `ptrdiff_t` or the corresponding `unsigned` type.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to `long double`.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

Conversion Characters

The following conversion characters are valid:

`d` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `strtol(3C)` with the value 10 for the *base* argument. In the absence of a size modifier, the corresponding argument must be a pointer to `int`.

`i` Matches an optionally signed integer, whose format is the same as expected for the subject sequence of `strtol()` with 0 for the *base* argument. In the absence of a size modifier, the corresponding argument must be a pointer to `int`.

`o` Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of `strtoul(3C)` with the value 8 for the *base* argument. In the absence of a size modifier, the corresponding argument must be a pointer to `unsigned int`.

`u` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 10 for the *base* argument. In the absence of a size modifier, the corresponding argument must be a pointer to `unsigned int`.

`x` Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 16 for the *base* argument. In the absence of a size modifier, the corresponding argument must be a pointer to `unsigned int`.

`a,e,f,g` Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of `strtod(3C)`. In the absence of a size modifier, the corresponding argument must be a pointer to `float`. The `e`, `f`, and `g` specifiers match hexadecimal floating point values only in C99/SUSv3 (see `standards(5)`) mode, but the `a` specifier always matches hexadecimal floating point values.

These conversion specifiers match any subject sequence accepted by `strtod(3C)`, including the INF, INFINITY, NAN, and NAN(*n-char-sequence*) forms. The result of the conversion is the same as that of calling `strtod()` (or `strtof()` or `strtold()`) with the matching sequence, including the raising of floating point exceptions and the setting of `errno` to `ERANGE`, if applicable.

`s` Matches a sequence of bytes that are not white-space characters. The corresponding argument must be a pointer to the initial byte of an array of `char`,

`signed char`, or `unsigned char` large enough to accept the sequence and a terminating null character code, which will be added automatically.

If an `l` (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character is converted to a wide-character as if by a call to the `mbrtowc(3C)` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted. The corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

`[` Matches a non-empty sequence of characters from a set of expected characters (the *scanset*). The normal skip over white-space characters is suppressed in this case. The corresponding argument must be a pointer to the initial byte of an array of `char`, `signed char`, or `unsigned char` large enough to accept the sequence and a terminating null byte, which will be added automatically.

If an `l` (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide-character as if by a call to the `mbrtowc()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted. The corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

The conversion specification includes all subsequent characters in the *format* string up to and including the matching right square bracket (`]`). The characters between the square brackets (the *scanlist*) comprise the scanset, unless the character after the left square bracket is a circumflex (`^`), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with `[ ]` or `[^]`, the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise the first right square bracket is the one that ends the conversion specification. If a – is in the scanlist and is not the first character, nor the second where the first character is a `^`, nor the last character, it indicates a range of characters to be matched.

`c` Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the conversion specification). The corresponding argument must be a pointer to the initial byte of an array of `char`, `signed char`, or `unsigned char` large enough to accept the sequence. No null byte is added. The normal skip over white-space characters is suppressed in this case.

If an `l` (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide-character as if by a call to the `mbrtowc()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted. The corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the resulting sequence of wide-characters. No null wide-character is added.

`p` Matches the set of sequences that is the same as the set of sequences that is produced by the `%p` conversion of the corresponding `printf(3C)` functions. The corresponding argument must be a pointer to a pointer to `void`. If the input item is a value converted earlier during the same program execution, the pointer that results will compare equal to that value; otherwise the behavior of the `%p` conversion is undefined.

`n` No input is consumed. The corresponding argument must be a pointer to the integer into which is to be written the number of bytes read from the input so far by this call to the `scanf()` functions. Execution of a `%n` conversion specification does not increment the assignment count returned at the completion of execution of the function.

`C` Same as `lc`.

`S` Same as `ls`.

`%` Matches a single `%`; no conversion or assignment occurs. The complete conversion specification must be `%%`.

If a conversion specification is invalid, the behavior is undefined.

The conversion characters `A`, `E`, `F`, `G`, and `X` are also valid and behave the same as, respectively, `a`, `e`, `f`, `g`, and `x`.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any bytes matching the current conversion specification (except for `%n`) have been read (other than leading white-space characters, where permitted), execution of the current conversion specification terminates with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) is terminated with an input failure.

Reaching the end of the string in `sscanf()` is equivalent to encountering end-of-file for `fscanf()`.

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white space (including newline characters) is left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the `%n` conversion specification.

The `fscanf()` and `scanf()` functions may mark the `st_atime` field of the file associated with *stream* for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc`(3C), `fgets`(3C), `fread`(3C), `fscanf()`, `getc`(3C), `getdelim`(3C), `getline`(3C), `getchar`(3C), `gets`(3C), or `scanf()` using *stream* that returns data not supplied by a prior call to `ungetc`(3C).

Return Values Upon successful completion, these functions return the number of successfully matched and assigned input items; this number can be 0 in the event of an early matching failure. If the input ends before the first matching failure or conversion, `EOF` is returned. If a read error occurs the error indicator for the stream is set, `EOF` is returned, and `errno` is set to indicate the error.

Errors For the conditions under which the `scanf()` functions will fail and may fail, refer to `fgetc`(3C) or `fgetwc`(3C).

In addition, `fscanf()` may fail if:

`EILSEQ` Input byte sequence does not form a valid character.

`EINVAL` There are insufficient arguments.

Usage If the application calling the `scanf()` functions has any objects of type `wint_t` or `wchar_t`, it must also include the header `<wchar.h>` to have these objects defined.

Examples **EXAMPLE 1** The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name)
```

with the input line:

```
25 54.32E–1 Hamster
```

will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain the string Hamster.

EXAMPLE 2 The call:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip 0123, and place the string 56\0 in *name*. The next call to `getchar`(3C) will return the character `a`.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also fgetc(3C), fgets(3C), fgetwc(3C), fread(3C), getdelim(3C), getline(3C), isspace(3C), printf(3C), setlocale(3C), strtod(3C), strtol(3C), strtoul(3C), wcrtomb(3C), ungetc(3C), attributes(5), standards(5)

Notes The behavior of the conversion specifier "%%" has changed for all of the functions described on this manual page. Previously the "%%" specifier accepted a "%" character from input only if there were no preceding whitespace characters. The new behavior accepts "%" even if there are preceding whitespace characters. This new behavior now aligns with the description on this manual page and in various standards. If the old behavior is desired, the conversion specification "%*[%]" can be used.

Name schedctl_init, schedctl_lookup, schedctl_exit, schedctl_start, schedctl_stop – preemption control

Synopsis

```
cc [ flag... ] file...  [ library... ]
#include <schedctl.h>

schedctl_t *schedctl_init(void);

schedctl_t *schedctl_lookup(void);

void schedctl_exit(void);

void schedctl_start(schedctl_t *ptr);

void schedctl_stop(schedctl_t *ptr);
```

Description These functions provide limited control over the scheduling of a thread (see `threads(5)`). They allow a running thread to give a hint to the kernel that preemptions of that thread should be avoided. The most likely use for these functions is to block preemption while holding a spinlock. Improper use of this facility, including attempts to block preemption for sustained periods of time, may result in reduced performance.

The `schedctl_init()` function initializes preemption control for the calling thread and returns a pointer used to refer to the data. If `schedctl_init()` is called more than once by the same thread, the most recently returned pointer is the only valid one.

The `schedctl_lookup()` function returns the currently allocated preemption control data associated with the calling thread that was previously returned by `schedctl_init()`. This can be useful in programs where it is difficult to maintain local state for each thread.

The `schedctl_exit()` function removes the preemption control data associated with the calling thread.

The `schedctl_start()` macro gives a hint to the kernel scheduler that preemption should be avoided on the current thread. The pointer passed to the macro must be the same as the pointer returned by the call to `schedctl_init()` by the current thread. The behavior of the program when other values are passed is undefined.

The `schedctl_stop()` macro removes the hint that was set by `schedctl_start()`. As with `schedctl_start()`, the pointer passed to the macro must be the same as the pointer returned by the call to `schedctl_init()` by the current thread.

The `schedctl_start()` and `schedctl_stop()` macros are intended to be used to bracket short critical sections, such as the time spent holding a spinlock. Other uses, including the failure to call `schedctl_stop()` soon after calling `schedctl_start()`, might result in poor performance.

Return Values The `schedctl_init()` function returns a pointer to a `schedctl_t` structure if the initialization was successful, or `NULL` otherwise. The `schedctl_lookup()` function returns a pointer to a `schedctl_t` structure if the data for that thread was found, or `NULL` otherwise.

Errors No errors are returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `priocntl(1)`, `exec(2)`, `fork(2)`, `priocntl(2)`, `attributes(5)`, `threads(5)`

Notes Preemption control is intended for use by threads belonging to the time-sharing (TS), interactive (IA), fair-share (FSS), and fixed-priority (FX) scheduling classes. If used by threads in other scheduling classes, such as real-time (RT), no errors will be returned but `schedctl_start()` and `schedctl_stop()` will not have any effect.

The data used for preemption control are not copied in the child of a `fork(2)`. Thus, if a process containing threads using preemption control calls `fork` and the child does not immediately call `exec(2)`, each thread in the child must call `schedctl_init()` again prior to any future uses of `schedctl_start()` and `schedctl_stop()`. Failure to do so will result in undefined behavior.

Name sched_getparam – get scheduling parameters

Synopsis

```
#include <sched.h>

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

Description The `sched_getparam()` function returns the scheduling parameters of a process specified by *pid* in the `sched_param` structure pointed to by *param*. The only required member of *param* is *sched_priority*.

If a process specified by *pid* exists and if the calling process has permission, the scheduling parameters for the process whose process ID is equal to *pid* will be returned.

If *pid* is 0, the scheduling parameters for the calling process will be returned. The behavior of the `sched_getparam()` function is unspecified if the value of *pid* is negative.

Return Values Upon successful completion, the `sched_getparam()` function returns `0`. If the call to `sched_getparam()` is unsuccessful, the function returns `−1` and sets `errno` to indicate the error.

Errors The `sched_getparam()` function will fail if:

`EPERM` The requesting process does not have permission to obtain the scheduling parameters of the specified process.

`ESRCH` No process can be found corresponding to that specified by *pid*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `librt(3LIB)`, `sched.h(3HEAD)`, `sched_getscheduler(3C)`, `sched_setparam(3C)`, `sched_setscheduler(3C)`, `attributes(5)`, `standards(5)`

Name sched_get_priority_max, sched_get_priority_min – get scheduling parameter limits

Synopsis

```
#include <sched.h>

int sched_get_priority_max(int policy);

int sched_get_priority_min(int policy);
```

Description The `sched_get_priority_max()` and `sched_get_priority_min()` functions return the appropriate maximum or minimum, respectfully, for the scheduling policy specified by *policy*.

The value of *policy* is one of the scheduling policy values defined in `<sched.h>`.

Return Values If successful, the `sched_get_priority_max()` and `sched_get_priority_min()` functions return the appropriate maximum or minimum priority values, respectively. If unsuccessful, they return `−1` and set `errno` to indicate the error.

Errors The `sched_get_priority_max()` and `sched_get_priority_min()` functions will fail if:

`EINVAL` The value of the *policy* parameter does not represent a defined scheduling policy.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `librt(3LIB)`, `sched.h(3HEAD)`, `sched_getparam(3C)`, `sched_setparam(3C)`, `sched_getscheduler(3C)`, `sched_rr_get_interval(3C)`, `sched_setscheduler(3C)`, `time.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name sched_getscheduler – get scheduling policy

Synopsis

```
#include <sched.h>

int sched_getscheduler(pid_t pid);
```

Description The `sched_getscheduler()` function returns the scheduling policy of the process specified by *pid*. If the value of *pid* is negative, the behavior of the `sched_getscheduler()` function is unspecified.

The values that can be returned by `sched_getscheduler()` are defined in the header `<sched.h>` and described on the `sched_setscheduler(3C)` manual page.

If a process specified by *pid* exists and if the calling process has permission, the scheduling policy will be returned for the process whose process ID is equal to *pid*.

If *pid* is 0, the scheduling policy will be returned for the calling process.

Return Values Upon successful completion, the `sched_getscheduler()` function returns the scheduling policy of the specified process. If unsuccessful, the function returns `−1` and sets `errno` to indicate the error.

Errors The `sched_getscheduler()` function will fail if:

`EPERM` The requesting process does not have permission to determine the scheduling policy of the specified process.

`ESRCH` No process can be found corresponding to that specified by *pid*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `librt(3LIB)`, `sched.h(3HEAD)`, `sched_getparam(3C)`, `sched_setparam(3C)`, `sched_setscheduler(3C)`, `attributes(5)`, `standards(5)`

Name sched_rr_get_interval – get execution time limits

Synopsis

```
#include <sched.h>

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

Description The `sched_rr_get_interval()` function updates the `timespec` structure referenced by the *interval* argument to contain the current execution time limit (that is, time quantum) for the process specified by *pid*. If *pid* is 0, the current execution time limit for the calling process will be returned.

Return Values If successful, the `sched_rr_get_interval()` function returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `sched_rr_get_interval()` function will fail if:

`ESRCH` No process can be found corresponding to that specified by *pid*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `librt(3LIB)`, `sched.h(3HEAD)`, `sched_getparam(3C)`, `sched_setparam(3C)`, `sched_get_priority_max(3C)`, `sched_getscheduler(3C)`, `sched_setscheduler(3C)`, `attributes(5)`, `standards(5)`

Name sched_setparam – set scheduling parameters

Synopsis

```
#include <sched.h>

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

Description The `sched_setparam()` function sets the scheduling parameters of the process specified by *pid* to the values specified by the `sched_param` structure pointed to by *param*. The value of the *sched_priority* member in the `sched_param` structure is any integer within the inclusive priority range for the current scheduling policy of the process specified by *pid*. Higher numerical values for the priority represent higher priorities. If the value of *pid* is negative, the behavior of the `sched_setparam()` function is unspecified.

If a process specified by *pid* exists and if the calling process has permission, the scheduling parameters will be set for the process whose process ID is equal to *pid*. The real or effective user ID of the calling process must match the real or saved (from `exec(2)`) user ID of the target process unless the effective user ID of the calling process is `0`. See `Intro(2)`.

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

The target process, whether it is running or not running, resumes execution after all other runnable processes of equal or greater priority have been scheduled to run.

If the priority of the process specified by the *pid* argument is set higher than that of the lowest priority running process and if the specified process is ready to run, the process specified by the *pid* argument preempts a lowest priority running process. Similarly, if the process calling `sched_setparam()` sets its own priority lower than that of one or more other non-empty process lists, then the process that is the head of the highest priority list also preempts the calling process. Thus, in either case, the originating process might not receive notification of the completion of the requested priority change until the higher priority process has executed.

Return Values If successful, the `sched_setparam()` function returns `0`.

If the call to `sched_setparam()` is unsuccessful, the priority remains unchanged, and the function returns `−1` and sets `errno` to indicate the error.

Errors The `sched_setparam()` function will fail if:

`EINVAL` One or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified *pid*.

`EPERM` The requesting process does not have permission to set the scheduling parameters for the specified process, or does not have the appropriate privilege to invoke `sched_setparam()`.

`ESRCH` No process can be found corresponding to that specified by *pid*.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also Intro(2), exec(2), librt(3LIB), sched.h(3HEAD), sched_getparam(3C), sched_getscheduler(3C), sched_setscheduler(3C), attributes(5), standards(5)

Name sched_setscheduler – set scheduling policy and scheduling parameters

Synopsis

```
#include <sched.h>

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

Description The `sched_setscheduler()` function sets the scheduling policy and scheduling parameters of the process specified by *pid* to *policy* and the parameters specified in the `sched_param` structure pointed to by *param*, respectively. The value of the `sched_priority` member in the `sched_param` structure is any integer within the inclusive priority range for the scheduling policy specified by *policy*. The `sched_setscheduler()` function ignores the other members of the `sched_param` structure. If the value of *pid* is negative, the behavior of the `sched_setscheduler()` function is unspecified.

The possible values for the *policy* parameter are defined in the header <`sched.h`> (see `sched.h(3HEAD)`):

If a process specified by *pid* exists and if the calling process has permission, the scheduling policy and scheduling parameters are set for the process whose process ID is equal to *pid.* The real or effective user ID of the calling process must match the real or saved (from `exec(2)`) user ID of the target process unless the effective user ID of the calling process is 0. See `Intro(2)`.

If *pid* is 0, the scheduling policy and scheduling parameters are set for the calling process.

To change the *policy* of any process to either of the real time policies `SCHED_FIFO` or `SCHED_RR`, the calling process must either have the `SCHED_FIFO` or `SCHED_RR` policy or have an effective user ID of `0`.

The `sched_setscheduler()` function is considered successful if it succeeds in setting the scheduling policy and scheduling parameters of the process specified by *pid* to the values specified by *policy* and the structure pointed to by *param*, respectively.

Return Values Upon successful completion, the function returns the former scheduling policy of the specified process. If the `sched_setscheduler()` function fails to complete successfully, the policy and scheduling paramenters remain unchanged, and the function returns −1 and sets `errno` to indicate the error.

Errors The `sched_setscheduler()` function will fail if:

`EINVAL` The value of *policy* is invalid, or one or more of the parameters contained in *param* is outside the valid range for the specified scheduling policy.

`EPERM` The requesting process does not have permission to set either or both of the scheduling parameters or the scheduling policy of the specified process.

`ESRCH` No process can be found corresponding to that specified by *pid.*

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also priocntl(1), Intro(2), exec(2), priocntl(2), librt(3LIB), sched.h(3HEAD), sched_get_priority_max(3C), sched_getparam(3C), sched_getscheduler(3C), sched_setparam(3C), attributes(5), standards(5)

Name sched_yield – yield processor

Synopsis

```
#include <sched.h>

int sched_yield(void);
```

Description The `sched_yield()` function forces the running thread to relinquish the processor until the process again becomes the head of its process list. It takes no arguments.

Return Values If successful, `sched_yield()` returns `0`, otherwise, it returns `−1`, and sets `errno` to indicate the error condition.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `librt(3LIB)`, `sched.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name seekdir – set position of directory stream

Synopsis

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

void seekdir(DIR *dirp, long int loc);
```

Description The `seekdir()` function sets the position of the next `readdir(3C)` operation on the directory stream specified by *dirp* to the position specified by *loc*. The value of *loc* should have been returned from an earlier call to `telldir(3C)`. The new position reverts to the one associated with the directory stream when `telldir()` was performed.

If the value of *loc* was not obtained from an earlier call to `telldir()` or if a call to `rewinddir(3C)` occurred between the call to `telldir ()` and the call to `seekdir()`, the results of subsequent calls to `readdir()` are unspecified.

Return Values The `seekdir()` function returns no value.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `opendir(3C)`, `readdir(3C)`, `rewinddir(3C)`, `telldir(3C)`, `attributes(5)`, `standards(5)`

Name select, pselect, FD_SET, FD_CLR, FD_ISSET, FD_ZERO – synchronous I/O multiplexing

Synopsis

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

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

int pselect(int nfds,
     fd_set *restrict readfds, fd_set *restrict writefds,
     fd_set *restrict errorfds,
     const struct timespec *restrict timeout,
     const sigset_t *restrict sigmask);

void FD_SET(int fd, fd_set *fdset);

void FD_CLR(int fd, fd_set *fdset);

int FD_ISSET(int fd, fd_set *fdset);

void FD_ZERO(fd_set *fdset);
```

Description The `pselect()` function examines the file descriptor sets whose addresses are passed in the *readfds*, *writefds*, and *errorfds* parameters to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively.

The `select()` function is equivalent to the `pselect()` function, except as follows:

- For the `select()` function, the timeout period is given in seconds and microseconds in an argument of type `struct timeval`, whereas for the `pselect()` function the timeout period is given in seconds and nanoseconds in an argument of type `struct timespec`.
- The `select()` function has no *sigmask* argument. It behaves as `pselect()` does when *sigmask* is a null pointer.
- Upon successful completion, the `select()` function might modify the object pointed to by the *timeout* argument.

The `select()` and `pselect()` functions support regular files, terminal and pseudo-terminal devices, STREAMS-based files, FIFOs, pipes, and sockets. The behavior of `select()` and `pselect()` on file descriptors that refer to other types of file is unspecified.

The *nfds* argument specifies the range of file descriptors to be tested. The first *nfds* descriptors are checked in each set; that is, the descriptors from zero through *nfds*-1 in the descriptor sets are examined.

If the *readfs* argument is not a null pointer, it points to an object of type `fd_set` that on input specifies the file descriptors to be checked for being ready to read, and on output indicates which file descriptors are ready to read.

If the *writefs* argument is not a null pointer, it points to an object of type `fd_set` that on input specifies the file descriptors to be checked for being ready to write, and on output indicates which file descriptors are ready to write.

If the *errorfds* argument is not a null pointer, it points to an object of type `fd_set` that on input specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending.

Upon successful completion, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments are modified to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively, and return the total number of ready descriptors in all the output sets. For each file descriptor less than *nfds*, the corresponding bit will be set on successful completion if it was set on input and the associated condition is true for that file descriptor.

If none of the selected descriptors are ready for the requested operation, the `select()` or `pselect()` function blocks until at least one of the requested operations becomes ready, until the timeout occurs, or until interrupted by a signal. The *timeout* parameter controls how long the `select()` or `pselect()` function takes before timing out. If the *timeout* parameter is not a null pointer, it specifies a maximum interval to wait for the selection to complete. If the specified time interval expires without any requested operation becoming ready, the function returns. If the *timeout* parameter is a null pointer, then the call to `select()` or `pselect()` blocks indefinitely until at least one descriptor meets the specified criteria. To effect a poll, the *timeout* parameter should not be a null pointer, and should point to a zero-valued `timespec` structure.

The use of a *timeout* does not affect any pending timers set up by `alarm(2)`, `ualarm(3C)`, or `setitimer(2)`.

If *sigmask* is not a null pointer, then the `pselect()` function replaces the signal mask of the process by the set of signals pointed to by *sigmask* before examining the descriptors, and restores the signal mask of the process before returning.

A descriptor is considered ready for reading when a call to an input function with `O_NONBLOCK` clear would not block, whether or not the function would transfer data successfully. (The function might return data, an end-of-file indication, or an error other than one indicating that it is blocked, and in each of these cases the descriptor will be considered ready for reading.)

A descriptor is considered ready for writing when a call to an output function with `O_NONBLOCK` clear would not block, whether or not the function would transfer data successfully.

If a socket has a pending error, it is considered to have an exceptional condition pending. Otherwise, what constitutes an exceptional condition is file type-specific. For a file descriptor for use with a socket, it is protocol-specific except as noted below. For other file types, if the

operation is meaningless for a particular file type, `select()` or `pselect()` indicates that the descriptor is ready for read or write operations and indicates that the descriptor has no exceptional condition pending.

If a descriptor refers to a socket, the implied input function is the `recvmsg(3XNET)` function with parameters requesting normal and ancillary data, such that the presence of either type causes the socket to be marked as readable. The presence of out-of-band data is checked if the socket option `SO_OOBINLINE` has been enabled, as out-of-band data is enqueued with normal data. If the socket is currently listening, then it is marked as readable if an incoming connection request has been received, and a call to the accept function completes without blocking.

If a descriptor refers to a socket, the implied output function is the `sendmsg(3XNET)` function supplying an amount of normal data equal to the current value of the `SO_SNDLOWAT` option for the socket. If a non-blocking call to the connect function has been made for a socket, and the connection attempt has either succeeded or failed leaving a pending error, the socket is marked as writable.

A socket is considered to have an exceptional condition pending if a receive operation with `O_NONBLOCK` clear for the open file description and with the `MSG_OOB` flag set would return out-of-band data without blocking. (It is protocol-specific whether the `MSG_OOB` flag would be used to read out-of-band data.) A socket will also be considered to have an exceptional condition pending if an out-of-band data mark is present in the receive queue.

A file descriptor for a socket that is listening for connections will indicate that it is ready for reading, when connections are available. A file descriptor for a socket that is connecting asynchronously will indicate that it is ready for writing, when a connection has been established.

Selecting true for reading on a socket descriptor upon which a `listen(3XNET)` call has been performed indicates that a subsequent `accept(3XNET)` call on that descriptor will not block.

If the *timeout* argument is not a null pointer, it points to an object of type `struct timeval` that specifies a maximum interval to wait for the selection to complete. If the *timeout* argument points to an object of type `struct timeval` whose members are 0, `select()` does not block. If the *timeout* argument is a null pointer, `select()` blocks until an event causes one of the masks to be returned with a valid (non-zero) value. If the time limit expires before any event occurs that would cause one of the masks to be set to a non-zero value, `select()` completes successfully and returns 0.

If the *readfs*, *writefs*, and *errorfds* arguments are all null pointers and the *timeout* argument is not a null pointer, `select()` or `pselect()` blocks for the time specified, or until interrupted by a signal. If the *readfs*, *writefs*, and *errorfds* arguments are all null pointers and the *timeout* argument is a null pointer, `select()` blocks until interrupted by a signal.

File descriptors associated with regular files always select true for ready to read, ready to write, and error conditions.

On failure, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments are not modified. If the timeout interval expires without the specified condition being true for any of the specified file descriptors, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments have all bits set to 0.

File descriptor masks of type `fd_set` can be initialized and tested with the macros `FD_CLR()`, `FD_ISSET()`, `FD_SET()`, and `FD_ZERO()`.

`FD_CLR`(*fd*, &*fdset*)	Clears the bit for the file descriptor *fd* in the file descriptor set *fdset*.
`FD_ISSET`(*fd*, &*fdset*)	Returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.
`FD_SET`(*fd*, &*fdset*)	Sets the bit for the file descriptor *fd* in the file descriptor set *fdset*.
`FD_ZERO`(&*fdset*)	Initializes the file descriptor set *fdset* to have zero bits for all file descriptors.

The behavior of these macros is undefined if the *fd* argument is less than 0 or greater than or equal to `FD_SETSIZE`, or if *fd* is not a valid file descriptor, or if any of the arguments are expressions with side effects.

Return Values On successful completion, `select()` and `pselect()` return the total number of bits set in the bit masks. Otherwise, `−1` is returned and `errno` is set to indicate the error.

The `FD_CLR()`, `FD_SET()`, and `FD_ZERO()` macros return no value. The `FD_ISSET()` macro returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and `0` otherwise.

Errors The `select()` and `pselect()` functions will fail if:

`EBADF`	One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.
`EINTR`	The function was interrupted before any of the selected events occurred and before the timeout interval expired. If `SA_RESTART` has been set for the interrupting signal, it is implementation-dependent whether `select()` restarts or returns with `EINTR`.
`EINVAL`	An invalid timeout interval was specified.
`EINVAL`	The *nfds* argument is less than 0 or greater than `FD_SETSIZE`.
`EINVAL`	One of the specified file descriptors refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.
`EINVAL`	A component of the pointed-to time limit is outside the acceptable range: `t_sec` must be between `0` and 10^8, inclusive. `t_usec` must be greater than or equal to `0`, and less than 10^6.

Usage The poll(2) function is preferred over this function. It must be used when the number of file descriptors exceeds FD_SETSIZE.

The use of a timeout does not affect any pending timers set up by alarm(2), ualarm(3C) or setitimer(2).

On successful completion, the object pointed to by the *timeout* argument may be modified.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also alarm(2), fcntl(2), poll(2), read(2), setitimer(2), write(2), accept(3SOCKET), listen(3SOCKET), ualarm(3C), attributes(5), standards(5)

Notes The default value for FD_SETSIZE (currently 1024) is larger than the default limit on the number of open files. To accommodate 32-bit applications that wish to use a larger number of open files with select(), it is possible to increase this size at compile time by providing a larger definition of FD_SETSIZE before the inclusion of any system-supplied header. The maximum supported size for FD_SETSIZE is 65536. The default value is already 65536 for 64-bit applications.

Name semaphore, sema_init, sema_destroy, sema_wait, sema_trywait, sema_post – semaphores

Synopsis

```
cc [ flag... ] file... -lthread  –lc [ library... ]
#include <synch.h>

int sema_init(sema_t *sp, unsigned int count, int type,
     void * arg);

int sema_destroy(sema_t *sp);

int sema_wait(sema_t *sp);

int sema_trywait(sema_t *sp);

int sema_post(sema_t *sp);
```

Description A semaphore is a non-negative integer count and is generally used to coordinate access to resources. The initial semaphore count is set to the number of free resources, then threads slowly increment and decrement the count as resources are added and removed. If the semaphore count drops to 0, which means no available resources, threads attempting to decrement the semaphore will block until the count is greater than 0.

Semaphores can synchronize threads in this process and other processes if they are allocated in writable memory and shared among the cooperating processes (see `mmap(2)`), and have been initialized for this purpose.

Semaphores must be initialized before use; semaphores pointed to by *sp* to *count* are initialized by `sema_init()`. The *type* argument can assign several different types of behavior to a semaphore. No current type uses *arg*, although it may be used in the future.

The *type* argument may be one of the following:

`USYNC_PROCESS` The semaphore can synchronize threads in this process and other processes. Initializing the semaphore should be done by only one process. A semaphore initialized with this type must be allocated in memory shared between processes, either in Sys V shared memory (see `shmop(2)`), or in memory mapped to a file (see `mmap(2)`). It is illegal to initialize the object this way and not allocate it in such shared memory. *arg* is ignored.

`USYNC_THREAD` The semaphore can synchronize threads only in this process. The *arg* argument is ignored. `USYNC_THREAD` does not support multiple mappings to the same logical synch object. If you need to `mmap()` a synch object to different locations within the same address space, then the synch object should be initialized as a shared object `USYNC_PROCESS` for Solaris threads and `PTHREAD_PROCESS_PRIVATE` for POSIX threads.

A semaphore must not be simultaneously initialized by multiple threads, nor re-initialized while in use by other threads.

Default semaphore initialization (intra-process):

```
sema_t sp;
int count  =  1;
sema_init(&sp, count, NULL, NULL);
```

or

```
sema_init(&sp, count, USYNC_THREAD, NULL);
```

Customized semaphore initialization (inter-process):

```
sema_t sp;
int count  =  1;
sema_init(&sp, count, USYNC_PROCESS, NULL);
```

The `sema_destroy()` function destroys any state related to the semaphore pointed to by *sp*. The semaphore storage space is not released.

The `sema_wait()` function blocks the calling thread until the semaphore count pointed to by *sp* is greater than 0, and then it atomically decrements the count.

The `sema_trywait()` function atomically decrements the semaphore count pointed to by *sp*, if the count is greater than 0; otherwise, it returns an error.

The `sema_post()` function atomically increments the semaphore count pointed to by *sp*. If there are any threads blocked on the semaphore, one will be unblocked.

The semaphore functionality described on this man page is for the Solaris threads implementation. For the POSIX-conforming semaphore interface documentation, see `sem_close`(3C), `sem_destroy`(3C), `sem_getvalue`(3C), `sem_init`(3C), `sem_open`(3C), `sem_post`(3C), `sem_unlink`(3C), and `sem_wait`(3C).

Return Values Upon successful completion, `0` is returned; otherwise, a non-zero value indicates an error.

Errors These functions will fail if:

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

`EFAULT` Either the *sp* or *arg* argument points to an illegal address.

The `sema_wait()` function will fail if:

`EINTR` The wait was interrupted by a signal or `fork()`.

The `sema_trywait()` function will fail if:

`EBUSY` The semaphore pointed to by *sp* has a 0 count.

The `sema_post()` function will fail if:

`EOVERFLOW` The semaphore value pointed to by *sp* exceeds `SEM_VALUE_MAX`.

Examples

EXAMPLE 1 The customer waiting-line in a bank is analogous to the synchronization scheme of a semaphore using `sema_wait()` and `sema_trywait()`:

```
/* cc [ flag  . . . ] file  . . . –lthread [ library  . . . ] */
#include <errno.h>
#define TELLERS 10
sema_t     tellers;     /* semaphore */
int banking_hours(), deposit_withdrawal;
void*customer(), do_business(), skip_banking_today();
. . .

sema_init(&tellers, TELLERS, USYNC_THREAD, NULL);
    /* 10 tellers available */
while(banking_hours())
    pthread_create(NULL, NULL, customer, deposit_withdrawal);
. . .

void *
customer(int deposit_withdrawal)
{
     int this_customer, in_a_hurry = 50;
     this_customer = rand() % 100;

     if (this_customer == in_a_hurry)  {
         if (sema_trywait(&tellers) != 0)
             if (errno == EBUSY){ /* no teller available */
                  skip_banking_today(this_customer);
                  return;
         } /* else go immediately to available teller and
                                 decrement tellers */
      }
      else
         sema_wait(&tellers); /* wait for next teller, then
                                 proceed, and decrement tellers */

      do_business(deposit_withdrawal);
      sema_post(&tellers); /* increment tellers; this_customer's
                              teller is now available */
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also mmap(2), shmop(2), sem_close(3C), sem_destroy(3C), sem_getvalue(3C), sem_init(3C), sem_open(3C), sem_post(3C), sem_unlink(3C), sem_wait(3C), attributes(5), standards(5)

Notes These functions are also available by way of:

```
#include <thread.h>
```

By default, there is no defined order of unblocking for multiple threads waiting for a semaphore.

Name sem_close – close a named semaphore

Synopsis

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Description The `sem_close()` function is used to indicate that the calling process is finished using the named semaphore indicated by *sem*. The effects of calling `sem_close()` for an unnamed semaphore (one created by `sem_init(3C)`) are undefined. The `sem_close()` function deallocates (that is, make available for reuse by a subsequent `sem_open(3C)` by this process) any system resources allocated by the system for use by this process for this semaphore. The effect of subsequent use of the semaphore indicated by *sem* by this process is undefined. If the semaphore has not been removed with a successful call to `sem_unlink(3C)`, then `sem_close()` has no effect on the state of the semaphore. If the `sem_unlink(3C)` function has been successfully invoked for *name* after the most recent call to `sem_open(3C)` with `O_CREAT` for this semaphore, then when all processes that have opened the semaphore close it, the semaphore is no longer be accessible.

Return Values If successful, `sem_close()` returns `0`, otherwise it returns `−1` and sets `errno` to indicate the error.

Errors The `sem_close()` function will fail if:

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

`ENOSYS` The `sem_close()` function is not supported by the system.

Usage The `sem_close()` function should not be called for an unnamed semaphore initialized by `sem_init(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `sem_init(3C)`, `sem_open(3C)`, `sem_unlink(3C)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name sem_destroy – destroy an unnamed semaphore

Synopsis

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Description The `sem_destroy()` function is used to destroy the unnamed semaphore indicated by *sem*. Only a semaphore that was created using `sem_init(3C)` may be destroyed using `sem_destroy()`; the effect of calling `sem_destroy()` with a named semaphore is undefined. The effect of subsequent use of the semaphore *sem* is undefined until *sem* is re-initialized by another call to `sem_init(3C)`.

It is safe to destroy an initialised semaphore upon which no threads are currently blocked. The effect of destroying a semaphore upon which other threads are currently blocked is undefined.

Return Values If successful, `sem_destroy()` returns `0`, otherwise it returns `−1` and sets `errno` to indicate the error.

Errors The `sem_destroy()` function will fail if:

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

The `sem_destroy()` function may fail if:

`EBUSY` There are currently processes (or LWPs or threads) blocked on the semaphore.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `sem_init(3C)`, `sem_open(3C)`, `attributes(5)`, `standards(5)`

Name sem_getvalue – get the value of a semaphore

Synopsis

```
#include <semaphore.h>

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

Description The `sem_getvalue()` function updates the location referenced by the *sval* argument to have the value of the semaphore referenced by *sem* without affecting the state of the semaphore. The updated value represents an actual semaphore value that occurred at some unspecified time during the call, but it need not be the actual value of the semaphore when it is returned to the calling process.

If *sem* is locked, then the value returned by `sem_getvalue()` is either zero or a negative number whose absolute value represents the number of processes waiting for the semaphore at some unspecified time during the call.

The value set in *sval* may be 0 or positive. If *sval* is 0, there may be other processes (or LWPs or threads) waiting for the semaphore; if *sval* is positive, no process is waiting.

Return Values Upon successful completion, `sem_getvalue()` returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `sem_getvalue()` function will fail if:

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

`ENOSYS` The `sem_getvalue()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `semctl(2)`, `semget(2)`, `semop(2)`, `sem_post(3C)`, `sem_wait(3C)`, `attributes(5)`, `standards(5)`

Name sem_init – initialize an unnamed semaphore

Synopsis

```
#include <semaphore.h>

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

Description The `sem_init()` function is used to initialize the unnamed semaphore referred to by *sem*. The value of the initialized semaphore is *value*. Following a successful call to `sem_init()`, the semaphore may be used in subsequent calls to `sem_wait(3C)`, `sem_trywait(3C)`, `sem_post(3C)`, and `sem_destroy(3C)`. This semaphore remains usable until the semaphore is destroyed.

If the *pshared* argument has a non-zero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore *sem* can use *sem* for performing `sem_wait(3C)`, `sem_trywait(3C)`, `sem_post(3C)`, and `sem_destroy(3C)` operations.

Only *sem* itself may be used for performing synchronization. The result of referring to copies of *sem* in calls to `sem_wait(3C)`, `sem_trywait(3C)`, `sem_post(3C)`, and `sem_destroy(3C)`, is undefined.

If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use *sem* for performing `sem_wait(3C)`, `sem_trywait(3C)`, `sem_post(3C)`, and `sem_destroy(3C)` operations. The use of the semaphore by threads other than those created in the same process is undefined.

Attempting to initialize an already initialized semaphore results in undefined behavior.

The `sem_open(3C)` function is used with named semaphores.

Return Values Upon successful completion, the function initializes the semaphore in *sem*. Otherwise, it returns −1 and sets `errno` to indicate the error.

Errors The `sem_init()` function will fail if:

`EINVAL` The *value* argument exceeds `SEM_VALUE_MAX`.

`ENOSPC` A resource required to initialize the semaphore has been exhausted, or the resources have reached the limit on semaphores (`SEM_NSEMS_MAX`).

`ENOSYS` The `sem_init()` function is not supported by the system.

`EPERM` The process lacks the appropriate privileges to initialize the semaphore.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

MT-Level	MT-Safe
Standard	See standards(5).

See Also sem_destroy(3C), sem_open(3C), sem_post(3C), sem_wait(3C), attributes(5), standards(5)

Name sem_open – initialize/open a named semaphore

Synopsis

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag,
     /* unsigned long mode, unsigned int value */ ...);
```

Description The `sem_open()` function establishes a connection between a named semaphore and a process (or LWP or thread). Following a call to `sem_open()` with semaphore name *name*, the process may reference the semaphore associated with *name* using the address returned from the call. This semaphore may be used in subsequent calls to `sem_wait`(3C), `sem_trywait`(3C), `sem_post`(3C), and `sem_close`(3C). The semaphore remains usable by this process until the semaphore is closed by a successful call to `sem_close`(3C), `_Exit`(2), or one of the `exec` functions.

The *oflag* argument controls whether the semaphore is created or merely accessed by the call to `sem_open()`. The following flag bits may be set in *oflag*:

`O_CREAT` This flag is used to create a semaphore if it does not already exist. If `O_CREAT` is set and the semaphore already exists, then `O_CREAT` has no effect, except as noted under `O_EXCL`. Otherwise, `sem_open()` creates a named semaphore. The `O_CREAT` flag requires a third and a fourth argument: *mode*, which is of type `mode_t`, and *value*, which is of type `unsigned int`. The semaphore is created with an initial value of *value*. Valid initial values for semaphores are less than or equal to `SEM_VALUE_MAX`.

The user ID of the semaphore is set to the effective user ID of the process; the group ID of the semaphore is set to a system default group ID or to the effective group ID of the process. The permission bits of the semaphore are set to the value of the *mode* argument except those set in the file mode creation mask of the process (see `umask`(2)). When bits in *mode* other than the file permission bits are specified, the effect is unspecified.

After the semaphore named *name* has been created by `sem_open()` with the `O_CREAT` flag, other processes can connect to the semaphore by calling `sem_open()` with the same value of *name*.

`O_EXCL` If `O_EXCL` and `O_CREAT` are set, `sem_open()` fails if the semaphore *name* exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist are atomic with respect to other processes executing `sem_open()` with `O_EXCL` and `O_CREAT` set. If `O_EXCL` is set and `O_CREAT` is not set, the effect is undefined.

If flags other than `O_CREAT` and `O_EXCL` are specified in the *oflag* parameter, the effect is unspecified.

The *name* argument points to a string naming a semaphore object. It is unspecified whether the name appears in the file system and is visible to functions that take pathnames as arguments. The *name* argument conforms to the construction rules for a pathname. The first character of *name* must be a slash (/) character and the remaining characters of *name* cannot include any slash characters. For maximum portability, *name* should include no more than 14 characters, but this limit is not enforced.

If a process makes multiple successful calls to `sem_open()` with the same value for *name*, the same semaphore address is returned for each such successful call, provided that there have been no calls to `sem_unlink(3C)` for this semaphore.

References to copies of the semaphore produce undefined results.

The `sem_init(3C)` function is used with unnamed semaphores.

Return Values Upon successful completion, the function returns the address of the semaphore. Otherwise, it will return a value of `SEM_FAILED` and set `errno` to indicate the error. The symbol `SEM_FAILED` is defined in the header `<semaphore.h>`. No successful return from `sem_open()` will return the value `SEM_FAILED`.

Errors If any of the following conditions occur, the `sem_open()` function will return `SEM_FAILED` and set `errno` to the corresponding value:

`EACCES`	The named semaphore exists and the `O_RDWR` permissions are denied, or the named semaphore does not exist and permission to create the named semaphore is denied.
`EEXIST`	`O_CREAT` and `O_EXCL` are set and the named semaphore already exists.
`EINTR`	The `sem_open()` function was interrupted by a signal.
`EINVAL`	The `sem_open()` operation is not supported for the given name, or `O_CREAT` was set in *oflag* and *value* is greater than `SEM_VALUE_MAX`.
`EMFILE`	The number of open semaphore descriptors in this process exceeds `SEM_NSEMS_MAX`, or the number of open file descriptors in this process exceeds `OPEN_MAX`.
`ENAMETOOLONG`	The length of *name* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
`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 new named semaphore.
`ENOSYS`	The `sem_open()` function is not supported by the system.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also exec(2), exit(2), umask(2), sem_close(3C), sem_init(3C), sem_post(3C), sem_unlink(3C), sem_wait(3C), sysconf(3C), attributes(5), standards(5)

Name sem_post – increment the count of a semaphore

Synopsis

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Description The `sem_post()` function unlocks the semaphore referenced by *sem* by performing a semaphore unlock operation on that semaphore.

If the semaphore value resulting from this operation is positive, then no threads were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

If the value of the semaphore resulting from this operation is 0, then one of the threads blocked waiting for the semaphore will be allowed to return successfully from its call to `sem_wait(3C)`. If the symbol `_POSIX_PRIORITY_SCHEDULING` is defined, the thread to be unblocked will be chosen in a manner appropriate to the scheduling policies and parameters in effect for the blocked threads. In the case of the schedulers `SCHED_FIFO` and `SCHED_RR`, the highest priority waiting thread will be unblocked, and if there is more than one highest priority thread blocked waiting for the semaphore, then the highest priority thread that has been waiting the longest will be unblocked. If the symbol `_POSIX_PRIORITY_SCHEDULING` is not defined, the choice of a thread to unblock is unspecified.

Return Values If successful, `sem_post()` returns `0`; otherwise it returns `−1` and sets `errno` to indicate the error.

Errors The `sem_post()` function will fail if:

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

`ENOSYS` The `sem_post()` function is not supported by the system.

`EOVERFLOW` The semaphore value exceeds `SEM_VALUE_MAX`.

Usage The `sem_post()` function is reentrant with respect to signals and may be invoked from a signal-catching function. The semaphore functionality described on this manual page is for the POSIX (see `standards(5)`) threads implementation. For the documentation of the Solaris threads interface, see `semaphore(3C)`).

Examples See `sem_wait(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also sched_setscheduler(3C), sem_wait(3C), semaphore(3C), attributes(5), standards(5)

Name sem_timedwait, sem_reltimedwait_np – lock a semaphore

Synopsis

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

int sem_timedwait(sem_t *restrict sem,
     const struct timespec *restrict abs_timeout);

int sem_reltimedwait_np(sem_t *restrict sem,
     const struct timespec *restrict rel_timeout);
```

Description The `sem_timedwait()` function locks the semaphore referenced by *sem* as in the `sem_wait(3C)` function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a `sem_post(3C)` function, this wait is terminated when the specified timeout expires.

The `sem_reltimedwait_np()` function is identical to the `sem_timedwait()` function, except that the timeout is specified as a relative time interval.

For `sem_timedwait()`, the timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the `CLOCK_REALTIME` clock (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

For `sem_reltimedwait_np()`, the timeout expires when the time interval specified by *rel_timeout* passes, as measured by the `CLOCK_REALTIME` clock, or if the time interval specified by *rel_timeout* is negative at the time of the call.

The resolution of the timeout is the resolution of the `CLOCK_REALTIME` clock. The `timespec` data type is defined as a structure in the `<time.h>` header.

Under no circumstance does the function fail with a timeout if the semaphore can be locked immediately. The validity of the *abs_timeout* need not be checked if the semaphore can be locked immediately.

Return Values The `sem_timedwait()` and `sem_reltimedwait_np()` functions return 0 if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore is be unchanged and the function returns -1 and sets `errno` to indicate the error.

Errors The `sem_timedwait()` and `sem_reltimedwait_np()` functions will fail if:

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

`EINVAL` The process or thread would have blocked, and the timeout parameter specified a nanoseconds field value less than zero or greater than or equal to 1,000 million.

`ETIMEDOUT` The semaphore could not be locked before the specified timeout expired.

The `sem_timedwait()` and `sem_reltimedwait_np()` functions may fail if:

EDEADLK A deadlock condition was detected.

EINTR A signal interrupted this function.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Commmitted
MT-Level	MT-Safe
Committed	See below.
Standard	See standards(5).

For sem_timedwait(), see standards(5).

See Also semctl(2), semget(2), semop(2), time(2), sem_post(3C), sem_trywait(3C)sem_wait(3C), attributes(5), standards(5)

Name sem_unlink – remove a named semaphore

Synopsis

```
#include <semaphore.h>

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

Description The `sem_unlink()` function removes the semaphore named by the string *name*. If the semaphore named by *name* is currently referenced by other processes, then `sem_unlink()` has no effect on the state of the semaphore. If one or more processes have the semaphore open when `sem_unlink()` is called, destruction of the semaphore is postponed until all references to the semaphore have been destroyed by calls to `sem_close(3C)`, `_Exit(2)`, or one of the `exec` functions (see `exec(2)`) . Calls to `sem_open(3C)` to re-create or re-connect to the semaphore refer to a new semaphore after `sem_unlink()` is called. The `sem_unlink()` call does not block until all references have been destroyed; it returns immediately.

Return Values Upon successful completion, `sem_unlink()` returns `0`. Otherwise, the semaphore is not changed and the function returns a value of `−1` and sets `errno` to indicate the error.

Errors The `sem_unlink()` function will fail if:

`EACCES`	Permission is denied to unlink the named semaphore.
`ENAMETOOLONG`	The length of *name* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
`ENOENT`	The named semaphore does not exist.
`ENOSYS`	The `sem_unlink()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `exec(2)`, `exit(2)`, `sem_close(3C)`, `sem_open(3C)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name sem_wait, sem_trywait – acquire or wait for a semaphore

Synopsis

```
#include <semaphore.h>

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);
```

Description The `sem_wait()` function locks the semaphore referenced by *sem* by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread will not return from the call to `sem_wait()` until it either locks the semaphore or the call is interrupted by a signal. The `sem_trywait()` function locks the semaphore referenced by *sem* only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore.

Upon successful return, the state of the semaphore is locked and remains locked until the `sem_post(3C)` function is executed and returns successfully.

The `sem_wait()` function is interruptible by the delivery of a signal.

Return Values The `sem_wait()` and `sem_trywait()` functions return `0` if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore is unchanged, and the function returns `−1` and sets `errno` to indicate the error.

Errors The `sem_wait()` and `sem_trywait()` functions will fail if:

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

`ENOSYS` The `sem_wait()` and `sem_trywait()` functions are not supported by the system.

The `sem_trywait()` function will fail if:

`EAGAIN` The semaphore was already locked, so it cannot be immediately locked by the `sem_trywait()` operation.

The `sem_wait()` and `sem_trywait()` functions may fail if:

`EDEADLK` A deadlock condition was detected; that is, two separate processes are waiting for an available resource to be released via a semaphore "held" by the other process.

`EINTR` A signal interrupted this function.

Usage Realtime applications may encounter priority inversion when using semaphores. The problem occurs when a high priority thread “locks” (that is, waits on) a semaphore that is about to be “unlocked” (that is, posted) by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of

priority inversion. They can deal with it in a number of ways, such as by having critical sections that are guarded by semaphores execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

Examples

EXAMPLE 1 The customer waiting-line in a bank may be analogous to the synchronization scheme of a semaphore utilizing `sem_wait()` and `sem_trywait()`:

```
#include <errno.h>
#define TELLERS 10
sem_t  bank_line;      /* semaphore */
int banking_hours(), deposit_withdrawal;
void *customer(), do_business(), skip_banking_today();
thread_t tid;
...

sem_init(&bank_line,TRUE,TELLERS);  /* 10 tellers
                                       available */
while(banking_hours())
        thr_create(NULL, NULL, customer,
            (void *)deposit_withdrawal, THREAD_NEW_LWP, &tid);
...

void *
customer(deposit_withdrawal)
void *deposit_withdrawal;
{
    int this_customer, in_a_hurry = 50;
    this_customer = rand() % 100;
    if (this_customer == in_a_hurry)  {
        if (sem_trywait(&bank_line) != 0)
        if (errno == EAGAIN) {    /* no teller available */
            skip_banking_today(this_customer);
            return;
        }     /*else go immediately to available teller
               & decrement bank_line*/
    }
    else
        sem_wait(&bank_line); /* wait for next teller,
             then proceed, and decrement bank_line */
    do_business((int *)deposit_withdrawal);
    sem_getvalue(&bank_line,&num_tellers);
    sem_post(&bank_line); /* increment bank_line;
        this_customer's teller is now available */
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also sem_post(3C), attributes(5), standards(5)

Name setbuf, setvbuf – assign buffering to a stream

Synopsis

```
#include <stdio.h>

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

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

Description The `setbuf()` function may be used after the stream pointed to by *stream* (see `Intro(3)`) is opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the null pointer, input/output will be completely unbuffered. The constant `BUFSIZ`, defined in the `<stdio.h>` header, indicates the size of the array pointed to by *buf*.

The `setvbuf()` function may be used after a stream is opened but before it is read or written. The *type* argument determines how *stream* will be buffered. Legal values for *type* (defined in `<stdio.h>`) are:

`_IOFBF` Input/output to be fully buffered.

`_IOLBF` Output to be line buffered; the buffer will be flushed when a `NEWLINE` is written, the buffer is full, or input is requested.

`_IONBF` Input/output to be completely unbuffered.

If *buf* is not the null pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. The *size* argument specifies the size of the buffer to be used. If input/output is unbuffered, *buf* and *size* are ignored.

For a further discussion of buffering, see `stdio(3C)`.

Return Values If an illegal value for *type* is provided, `setvbuf()` returns a non-zero value. Otherwise, it returns `0`.

Usage A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

When using `setbuf()`, *buf* should always be sized using `BUFSIZ`. If the array pointed to by *buf* is larger than `BUFSIZ`, a portion of *buf* will not be used. If *buf* is smaller than `BUFSIZ`, other memory may be unexpectedly overwritten.

Parts of `buf` will be used for internal bookkeeping of the stream and, therefore, `buf` will contain less than *size* bytes when full. It is recommended that `stdio(3C)` be used to handle buffer allocation when using `setvbuf()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `fopen(3C)`, `getc(3C)`, `malloc(3C)`, `putc(3C)`, `stdio(3C)`, `attributes(5)`, `standards(5)`

Name setbuffer, setlinebuf – assign buffering to a stream

Synopsis

```
#include <stdio.h>

void setbuffer(FILE *iop, char *abuf, size_t asize);

int setlinebuf(FILE *iop);
```

Description The `setbuffer()` and `setlinebuf()` functions assign buffering to a stream. The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered, many characters are saved and written as a block; when it is line buffered, characters are saved until either a NEWLINE is encountered or input is read from `stdin`. The `fflush(3C)` function may be used to force the block out early. Normally all files are block buffered. A buffer is obtained from `malloc(3C)` upon the first `getc(3C)` or `putc(3C)` performed on the file. If the standard stream `stdout` refers to a terminal, it is line buffered. The standard stream `stderr` is unbuffered by default.

The `setbuffer()` function can be used after a stream *iop* has been opened but before it is read or written. It uses the character array *abuf* whose size is determined by the *asize* argument instead of an automatically allocated buffer. If *abuf* is the null pointer, input/output will be completely unbuffered. A manifest constant `BUFSIZ`, defined in the `<stdio.h>` header, tells how large an array is needed:

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 the stream *iop* is active.

A stream can be changed from unbuffered or line buffered to block buffered by using `freopen(3C)`. A stream can be changed from block buffered or line buffered to unbuffered by using `freopen(3C)` followed by `setbuf(3C)` with a buffer argument of `NULL`.

Return Values The `setlinebuf()` function returns no useful value.

See Also `malloc(3C)`, `fclose(3C)`, `fopen(3C)`, `fread(3C)`, `getc(3C)`, `printf(3C)`, `putc(3C)`, `puts(3C)`, `setbuf(3C)`, `setvbuf(3C)`

Notes A common source of error is allocating buffer space as an “automatic” variable in a code block, and then failing to close the stream in the same block.

Name setcat – define default catalog

Synopsis

```
#include <pfmt.h>

char *setcat(const char *catalog);
```

Description The `setcat()` function defines the default message catalog to be used by subsequent calls to `gettxt`(3C), `lfmt`(3C), or `pfmt`(3C) that do not explicitly specify a message catalog.

The *catalog* argument must be limited to 14 characters. These characters must be selected from a set of all characters values, excluding `\0` (null) and the ASCII codes for `/` (slash) and `:` (colon).

The `setcat()` function assumes that the catalog exists. No checking is done on the argument.

A null pointer passed as an argument will result in the return of a pointer to the current default message catalog name. A pointer to an empty string passed as an argument will cancel the default catalog.

If no default catalog is specified, or if *catalog* is an invalid catalog name, subsequent calls to `gettxt`(3C), `lfmt`(3C), or `pfmt`(3C) that do not explicitely specify a catalog name will use `Message not found!!\n` as default string.

Return Values Upon successful completion, `setcat()` returns a pointer to the catalog name. Otherwise, it returns a null pointer.

Examples **EXAMPLE 1** Example of `setcat()` function.

```
setcat("test");
gettxt(":10", "hello world\n")
```

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `gettxt`(3C), `lfmt`(3C), `pfmt`(3C), `setlocale`(3C), `attributes`(5), `environ`(5)

Name setenv – add or change environment variable

Synopsis

```
#include <stdlib.h>

int setenv(const char *envname, const char *envval,
     int overwrite);
```

Description The `setenv()` function updates or adds a variable in the environment of the calling process. The *envname* argument points to a string containing the name of an environment variable to be added or altered. The environment variable is set to the value to which *envval* points. The function fails if *envname* points to a string which contains an '=' character. If the environment variable named by *envname* already exists and the value of *overwrite* is non-zero, the function returns successfully and the environment is updated. If the environment variable named by *envname* already exists and the value of *overwrite* is zero, the function returns successfully and the environment remains unchanged.

If the application modifies *environ* or the pointers to which it points, the behavior of `setenv()` is undefined. The `setenv()` function updates the list of pointers to which *environ* points.

The strings described by *envname* and *envval* are copied by this function.

Return Values Upon successful completion, 0 is returned. Otherwise, -1 is returned, `errno` set to indicate the error, and the environment is left unchanged.

Errors The `setenv()` function will fail if:

`EINVAL` The *envname* argument is a null pointer, points to an empty string, or points to a string containing an '=' character.

`ENOMEM` Insufficient memory was available to add a variable or its value to the environment.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `getenv(3C)`, `unsetenv(3C)`, `attributes(5)`, `standards(5)`

Name setjmp, sigsetjmp, longjmp, siglongjmp – non-local goto

Synopsis

```
#include <setjmp.h>

int setjmp(jmp_buf env);

int sigsetjmp(sigjmp_buf env, int savemask);

void longjmp(jmp_buf env, int val);

void siglongjmp(sigjmp_buf env, int val);
```

Description These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

The `setjmp()` function saves its stack environment in *env* for later use by `longjmp()`.

The `sigsetjmp()` function saves the calling process's registers and stack environment (see `sigaltstack(2)`) in *env* for later use by `siglongjmp()`. If *savemask* is non-zero, the calling process's signal mask (see `sigprocmask(2)`) and scheduling parameters (see `priocntl(2)`) are also saved.

The `longjmp()` function restores the environment saved by the last call of `setjmp()` with the corresponding *env* argument. After `longjmp()` completes, program execution continues as if the corresponding call to `setjmp()` had just returned the value *val*. The caller of `setjmp()` must not have returned in the interim. The `longjmp()` function cannot cause `setjmp()` to return the value 0. If `longjmp()` is invoked with a second argument of 0, `setjmp()` will return 1. At the time of the second return from `setjmp()`, all external and static variables have values as of the time `longjmp()` is called (see `EXAMPLES`).

The `siglongjmp()` function restores the environment saved by the last call of `sigsetjmp()` with the corresponding *env* argument. After `siglongjmp()` completes, program execution continues as if the corresponding call to `sigsetjmp()` had just returned the value *val*. The `siglongjmp()` function cannot cause `sigsetjmp()` to return the value 0. If `siglongjmp()` is invoked with a second argument of 0, `sigsetjmp()` will return 1. At the time of the second return from `sigsetjmp()`, all external and static variables have values as of the time `siglongjmp()` was called.

If a signal-catching function interrupts `sleep(3C)` and calls `siglongjmp()` to restore an environment saved prior to the `sleep()` call, the action associated with `SIGALRM` and time it is scheduled to be generated are unspecified. It is also unspecified whether the `SIGALRM` signal is blocked, unless the process's signal mask is restored as part of the environment.

The `siglongjmp()` function restores the saved signal mask if and only if the *env* argument was initialized by a call to the `sigsetjmp()` function with a non-zero *savemask* argument.

The values of register and automatic variables are undefined. Register or automatic variables whose value must be relied upon must be declared as volatile.

Return Values If the return is from a direct invocation, `setjmp()` and `sigsetjmp()` return `0`. If the return is from a call to `longjmp()`, `setjmp()` returns a non-zero value. If the return is from a call to `siglongjmp()`, `sigsetjmp()` returns a non-zero value.

After `longjmp()` is completed, program execution continues as if the corresponding invocation of `setjmp()` had just returned the value specified by *val*. The `longjmp()` function cannot cause `setjmp()` to return 0; if *val* is 0, `setjmp()` returns 1.

After `siglongjmp()` is completed, program execution continues as if the corresponding invocation of `sigsetjmp()` had just returned the value specified by *val*. The `siglongjmp()` function cannot cause `sigsetjmp()` to return 0; if *val* is 0, `sigsetjmp()` returns 1.

Examples **EXAMPLE 1** Example of `setjmp()` and `longjmp()` functions.

The following example uses both `setjmp()` and `longjmp()` to return the flow of control to the appropriate instruction block:

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
jmp_buf env; static void signal_handler();

main( )  {
        int returned_from_longjump, processing = 1;
        unsigned int time_interval = 4;
        if ((returned_from_longjump = setjmp(env)) != 0)
            switch (returned_from_longjump)     {
              case SIGINT:
                printf("longjumped from interrupt %d\n",SIGINT);
                break;
              case SIGALRM:
                printf("longjumped from alarm %d\n",SIGALRM);
                break;
            }
        (void) signal(SIGINT, signal_handler);
        (void) signal(SIGALRM, signal_handler);
        alarm(time_interval);
        while (processing)        {
          printf(" waiting for you to INTERRUPT (cntrl-C) ...\n");
          sleep(1);
        }         /* end while forever loop */
}

static void signal_handler(sig)
int sig; {
        switch (sig)     {
          case SIGINT:    ...       /* process for interrupt */
                          longjmp(env,sig);
```

EXAMPLE 1 Example of `setjmp()` and `longjmp()` functions. *(Continued)*

```
                                /* break never reached */
          case SIGALRM:     ...       /* process for alarm */
                            longjmp(env,sig);
                                /* break never reached */
          default:          exit(sig);
        }
}
```

When this example is compiled and executed, and the user sends an interrupt signal, the output will be:

```
longjumped from interrupt
```

Additionally, every 4 seconds the alarm will expire, signalling this process, and the output will be:

```
longjumped from alarm
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	See `standards(5)`.

See Also

`getcontext(2)`, `priocntl(2)`, `sigaction(2)`, `sigaltstack(2)`, `sigprocmask(2)`, `signal(3C)`, `attributes(5)`, `standards(5)`

Warnings

If `longjmp()` or `siglongjmp()` are called even though *env* was never primed by a call to `setjmp()` or `sigsetjmp()`, or when the last such call was in a function that has since returned, the results are undefined.

Name setkey – set encoding key

Synopsis

```
#include <stdlib.h>

void setkey(const char *key);
```

Description The `setkey()` function provides (rather primitive) access to the hashing algorithm employed by the `crypt(3C)` function. The argument of `setkey()` is an array of length 64 bytes containing only the bytes with numerical value of 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is used by the algorithm. This is the key that will be used with the algorithm to encode a string *block* passed to `encrypt(3C)`.

Return Values No values are returned.

Errors The `setkey()` function will fail if:

`ENOSYS` The functionality is not supported on this implementation.

Usage In some environments, decoding may not be implemented. This is related to U.S. Government restrictions on encryption and decryption routines: the DES decryption algorithm cannot be exported outside the U.S.A. Historical practice has been to ship a different version of the encryption library without the decryption feature in the routines supplied. Thus the exported version of `encrypt()` does encoding but not decoding.

Because `setkey()` does not return a value, applications wishing to check for errors should set `errno` to 0, call `setkey()`, then test `errno` and, if it is non-zero, assume an error has occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `crypt(3C)`, `encrypt(3C)`, `attributes(5)`, `standards(5)`

Name setlabel – define the label for `pfmt()` and `lfmt()`

Synopsis

```
#include <pfmt..h>

int setlabel(const char *label);
```

Description The `setlabel()` function defines the label for messages produced in standard format by subsequent calls to `lfmt`(3C) and `pfmt`(3C).

The *label* argument is a character string no more than 25 characters in length.

No label is defined before `setlabel()` is called. The label should be set once at the beginning of a utility and remain constant. A null pointer or an empty string passed as argument will reset the definition of the label.

Return Value Upon successful completion, `setlabel()` returns `0`; otherwise, it returns a non-zero value.

Examples The following code (without previous call to `setlabel()`):

```
pfmt(stderr, MM_ERROR, "test:2:Cannot open file\n");
setlabel("UX:test");
pfmt(stderr, MM_ERROR, "test:2:Cannot open file\n");
```

will produce the following output:

```
ERROR: Cannot open file
UX:test: ERROR: Cannot open file
```

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `getopt`(3C), `lfmt`(3C), `pfmt`(3C), `attributes`(5)

Name setlocale – modify and query a program's locale

Synopsis

```
#include <locale.h>

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

Description The `setlocale()` function selects the appropriate piece of the program's locale as specified by the *category* and *locale* arguments. The *category* argument may have the following values: `LC_CTYPE`, `LC_NUMERIC`, `LC_TIME`, `LC_COLLATE`, `LC_MONETARY`, `LC_MESSAGES`, and `LC_ALL`. These names are defined in the <`locale.h`> header. The `LC_ALL` variable names all of a program's locale categories.

The `LC_CTYPE` variable affects the behavior of character handling functions such as `isdigit`(3C) and `tolower`(3C), and multibyte character functions such as `mbtowc`(3C) and `wctomb`(3C).

The `LC_NUMERIC` variable affects the decimal point character and thousands separator character for the formatted input/output functions and string conversion functions.

The `LC_TIME` variable affects the date and time format as delivered by `ascftime`(3C), `cftime`(3C), `getdate`(3C), `strftime`(3C), and `strptime`(3C).

The `LC_COLLATE` variable affects the sort order produced by collating functions such as `strcoll`(3C) and `strxfrm`(3C).

The `LC_MONETARY` variable affects the monetary formatted information returned by `localeconv`(3C).

The `LC_MESSAGES` variable affects the behavior of messaging functions such as `dgettext`(3C), `gettext`(3C), and `gettxt`(3C).

A value of "C" for *locale* specifies the traditional UNIX system behavior. At program startup, the equivalent of

```
setlocale(LC_ALL, "C")
```

is executed. This has the effect of initializing each category to the locale described by the environment "C".

A value of "" for *locale* specifies that the locale should be taken from environment variables. The order in which the environment variables are checked for the various categories is given below:

Category	1st Env Var	2nd Env Var	3rd Env Var
LC_CTYPE:	LC_ALL	LC_CTYPE	LANG
LC_COLLATE:	LC_ALL	LC_COLLATE	LANG

Category	1st Env Var	2nd Env Var	3rd Env Var
LC_TIME:	LC_ALL	LC_TIME	LANG
LC_NUMERIC:	LC_ALL	LC_NUMERIC	LANG
LC_MONETARY:	LC_ALL	LC_MONETARY	LANG
LC_MESSAGES:	LC_ALL	LC_MESSAGES	LANG

If a pointer to a string is given for *locale*, `setlocale()` attempts to set the locale for the given category to *locale*. If `setlocale()` succeeds, *locale* is returned. If `setlocale()` fails, a null pointer is returned and the program's locale is not changed.

For category `LC_ALL`, the behavior is slightly different. If a pointer to a string is given for *locale* and `LC_ALL` is given for *category*, `setlocale()` attempts to set the locale for all the categories to *locale*. The *locale* may be a simple locale, consisting of a single locale, or a composite locale. If the locales for all the categories are the same after all the attempted locale changes, `setlocale()` will return a pointer to the common simple locale. If there is a mixture of locales among the categories, `setlocale()` will return a composite locale.

Return Values Upon successful completion, `setlocale()` returns the string associated with the specified category for the new locale. Otherwise, `setlocale()` returns a null pointer and the program's locale is not changed.

A null pointer for *locale* causes `setlocale()` to return a pointer to the string associated with the *category* for the program's current locale. The program's locale is not changed.

The string returned by `setlocale()` is such that a subsequent call with that string and its associated *category* will restore that part of the program's locale. The string returned must not be modified by the program, but may be overwritten by a subsequent call to `setlocale()`.

Errors No errors are defined.

Files `/usr/lib/locale/`*locale* locale database directory for *locale*

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also locale(1), ctype(3C), getdate(3C) gettext(3C), gettxt(3C), isdigit(3C), libc(3LIB), localeconv(3C), mbtowc(3C), strcoll(3C), strftime(3C), strptime(3C) strxfrm(3C) tolower(3C), wctomb(3C), attributes(5), environ(5), locale(5), standards(5)

Notes It is unsafe for any thread to change locale (by calling `setlocale()` with a non-null locale argument) in a multithreaded application while any other thread in the application is using any locale-sensitive routine. To change locale in a multithreaded application, `setlocale()` should be called prior to using any locale-sensitive routine. Using `setlocale()` to query the current locale is safe and can be used anywhere in a multithreaded application except when some other thread is changing locale.

It is the user's responsibility to ensure that mixed locale categories are compatible. For example, setting `LC_CTYPE=C` and `LC_TIME=ja` (where `ja` indicates Japanese) will not work, because Japanese time cannot be represented in the “C” locale's ASCII codeset.

Name shm_open – open a shared memory object

Synopsis

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

int shm_open(const char *name, int oflag, mode_t mode);
```

Description The `shm_open()` function establishes a connection between a shared memory object and a file descriptor. It creates an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. The file descriptor is used by other functions to refer to that shared memory object. The *name* argument points to a string naming a shared memory object. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The *name* argument conforms to the construction rules for a pathname. The first character of *name* must be a slash (/) character and the remaining characters of *name* cannot include any slash characters. For maximum portability, *name* should include no more than 14 characters, but this limit is not enforced.

If successful, `shm_open()` returns a file descriptor for the shared memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other processes. It is unspecified whether the file offset is set. The `FD_CLOEXEC` file descriptor flag associated with the new file descriptor is set.

The file status flags and file access modes of the open file description are according to the value of *oflag*. The *oflag* argument is the bitwise inclusive OR of the following flags defined in the header <`fcntl.h`>. Applications specify exactly one of the first two values (access modes) below in the value of *oflag*:

`O_RDONLY` Open for read access only.

`O_RDWR` Open for read or write access.

Any combination of the remaining flags may be specified in the value of *oflag*:

`O_CREAT` If the shared memory object exists, this flag has no effect, except as noted under `O_EXCL` below. Otherwise the shared memory object is created; the user ID of the shared memory object will be set to the effective user ID of the process; the group ID of the shared memory object will be set to a system default group ID or to the effective group ID of the process. The permission bits of the shared memory object will be set to the value of the *mode* argument except those set in the file mode creation mask of the process. When bits in *mode* other than the file permission bits are set, the effect is unspecified. The *mode* argument does not affect whether the shared memory object is opened for reading, for writing, or for both. The shared memory object has a size of zero.

`O_EXCL` If `O_EXCL` and `O_CREAT` are set, `shm_open()` fails if the shared memory object exists. The check for the existence of the shared memory object and the creation of the object if it does not exist is atomic with respect to other processes

executing `shm_open()` naming the same shared memory object with `O_EXCL` and `O_CREAT` set. If `O_EXCL` is set and `O_CREAT` is not set, the result is undefined.

`O_TRUNC` If the shared memory object exists, and it is successfully opened `O_RDWR`, the object will be truncated to zero length and the mode and owner will be unchanged by this function call. The result of using `O_TRUNC` with `O_RDONLY` is undefined.

When a shared memory object is created, the state of the shared memory object, including all data associated with the shared memory object, persists until the shared memory object is unlinked and all other references are gone. It is unspecified whether the name and shared memory object state remain valid after a system reboot.

Return Values Upon successful completion, the `shm_open()` function returns a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it returns `−1` and sets `errno` to indicate the error condition.

Errors The `shm_open()` function will fail if:

`EACCES` The shared memory object exists and the permissions specified by *oflag* are denied, or the shared memory object does not exist and permission to create the shared memory object is denied, or `O_TRUNC` is specified and write permission is denied.

`EEXIST` `O_CREAT` and `O_EXCL` are set and the named shared memory object already exists.

`EINTR` The `shm_open()` operation was interrupted by a signal.

`EINVAL` The `shm_open()` operation is not supported for the given name.

`EMFILE` Too many file 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` while `_POSIX_NO_TRUNC` is in effect.

`ENFILE` Too many shared memory objects are currently open in the system.

`ENOENT` `O_CREAT` is not set and the named shared memory object does not exist.

`ENOSPC` There is insufficient space for the creation of the new shared memory object.

`ENOSYS` The `shm_open()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE

Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `close(2)`, `dup(2)`, `exec(2)`, `fcntl(2)`, `mmap(2)`, `umask(2)`, `shm_unlink(3C)`, `sysconf(3C)`, `fcntl.h(3HEAD)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name shm_unlink – remove a shared memory object

Synopsis

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

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

Description The `shm_unlink()` function removes the name of the shared memory object named by the string pointed to by *name*. If one or more references to the shared memory object exists when the object is unlinked, the name is removed before `shm_unlink()` returns, but the removal of the memory object contents will be postponed until all open and mapped references to the shared memory object have been removed.

Return Values Upon successful completion, `shm_unlink()` returns `0`. Otherwise it returns `−1` and sets `errno` to indicate the error condition, and the named shared memory object is not affected by this function call.

Errors The `shm_unlink()` function will fail if:

`EACCES` Permission is denied to unlink the named shared memory object.

`ENAMETOOLONG` The length of the *name* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

`ENOENT` The named shared memory object does not exist.

`ENOSYS` The `shm_unlink()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `close(2)`, `mmap(2)`, `mlock(3C)`, `shm_open(3C)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name sigfpe – signal handling for specific SIGFPE codes

Synopsis

```
#include <floatingpoint.h>
#include <siginfo.h>

sigfpe_handler_type sigfpe(sigfpe_code_type code,
     sigfpe_handler_type hdl);
```

Description The `sigfpe()` function allows signal handling to be specified for particular `SIGFPE` codes. A call to `sigfpe()` defines a new handler *hdl* for a particular `SIGFPE` *code* and returns the old handler as the value of the function `sigfpe()`. Normally handlers are specified as pointers to functions; the special cases `SIGFPE_IGNORE`, `SIGFPE_ABORT`, and `SIGFPE_DEFAULT` allow ignoring, dumping core using `abort`(3C), or default handling respectively. Default handling is to dump core using `abort`(3C).

The *code* argument is usually one of the five IEEE 754-related `SIGFPE` codes:

```
FPE_FLTRES   fp_inexact — floating-point inexact result
FPE_FLTDIV   fp_division — floating-point division by zero
FPE_FLTUND   fp_underflow — floating-point underflow
FPE_FLTOVF   fp_overflow — floating-point overflow
FPE_FLTINV   fp_invalid — floating-point invalid operation
```

Three steps are required to intercept an IEEE 754-related `SIGFPE` code with `sigfpe()`:

1. Set up a handler with `sigfpe()`.
2. Enable the relevant IEEE 754 trapping capability in the hardware, perhaps by using assembly-language instructions.
3. Perform a floating-point operation that generates the intended IEEE 754 exception.

The `sigfpe()` function never changes floating-point hardware mode bits affecting IEEE 754 trapping. No IEEE 754-related `SIGFPE` signals will be generated unless those hardware mode bits are enabled.

`SIGFPE` signals can be handled using `sigfpe()`, `sigaction`(2) or `signal`(3C). In a particular program, to avoid confusion, use only one of these interfaces to handle `SIGFPE` signals.

Examples **EXAMPLE 1** Example Of A User-Specified Signal Handler

A user-specified signal handler might look like this:

```
#include <floatingpoint.h>
#include <siginfo.h>
#include <ucontext.h>
/*
* The sample_handler prints out a message then commits suicide.
*/
void
sample_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
```

EXAMPLE 1 Example Of A User-Specified Signal Handler *(Continued)*

```
    char *label;
        switch (sip->si_code) {
    case FPE_FLTINV: label = "invalid operand"; break;
    case FPE_FLTRES: label = "inexact"; break;
    case FPE_FLTDIV: label = "division-by-zero"; break;
    case FPE_FLTUND: label = "underflow"; break;
    case FPE_FLTOVF: label = "overflow"; break;
    default: label = "???"; break;
    }
    fprintf(stderr,
        "FP exception %s (0x%x) occurred at address %p.\n",
        label, sip->si_code, (void *) sip->si_addr);
    abort();
}
```

and it might be set up like this:

```
#include <floatingpoint.h>
#include <siginfo.h>
#include <ucontext.h>
extern void sample_handler(int, siginfo_t *, ucontext_t *);
main(void) {
      sigfpe_handler_type hdl, old_handler1, old_handler2;
/*
 * save current fp_overflow and fp_invalid handlers; set the new
* fp_overflow handler to sample_handler( ) and set the new
* fp_invalid handler to SIGFPE_ABORT (abort on invalid)
*/
    hdl = (sigfpe_handler_type) sample_handler;
    old_handler1 = sigfpe(FPE_FLTOVF, hdl);
    old_handler2 = sigfpe(FPE_FLTINV, SIGFPE_ABORT);
    . . .
/*
 * restore old fp_overflow and fp_invalid handlers
 */
     sigfpe(FPE_FLTOVF, old_handler1);
     sigfpe(FPE_FLTINV, old_handler2);
}
```

Files `/usr/include/floatingpoint.h`

`/usr/include/siginfo.h`

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `sigaction(2)`, `abort(3C)`, `signal(3C)`, `attributes(5)`, `floatingpoint.h(3HEAD)`

Diagnostics The `sigfpe()` function returns (void(*)())-1 if *code* is not zero or a defined `SIGFPE` code.

Name siginterrupt – allow signals to interrupt functions

Synopsis

```
#include <signal.h>

int siginterrupt(int sig, int flag);
```

Description The `siginterrupt()` function changes the restart behavior when a function is interrupted by the specified signal. The function `siginterrupt`(*sig*, *flag*) has an effect as if implemented as:

```
siginterrupt(int sig, int flag) {
     int ret;
     struct sigaction act;
     (void) sigaction(sig, NULL, &act);
     if (flag)
          act.sa_flags &=  SA_RESTART;
     else
          act.sa_flags |= SA_RESTART;
     ret = sigaction(sig, &act, NULL);
     return ret;
}
```

Return Values Upon successful completion, `siginterrupt()` returns 0. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `siginterrupt()` function will fail if:

`EINVAL` The *sig* argument is not a valid signal number.

Usage The `siginterrupt()` function supports programs written to historical system interfaces. A standard-conforming application, when being written or rewritten, should use `sigaction(2)` with the `SA_RESTART` flag instead of `siginterrupt()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `sigaction(2)`, `signal.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name signal, sigset, sighold, sigrelse, sigignore, sigpause – simplified signal management for application processes

Synopsis

```
#include <signal.h>

void (*signal(int sig, void (*disp)(int)))(int);

void (*sigset(int sig, void (*disp)(int)))(int);

int sighold(int sig);

int sigrelse(int sig);

int sigignore(int sig);

int sigpause(int sig);
```

Description These functions provide simplified signal management for application processes. See `signal.h`(3HEAD) for an explanation of general signal concepts.

The `signal()` and `sigset()` functions modify signal dispositions. The *sig* argument specifies the signal, which may be any signal except `SIGKILL` and `SIGSTOP`. The *disp* argument specifies the signal's disposition, which may be `SIG_DFL`, `SIG_IGN`, or the address of a signal handler. If `signal()` is used, *disp* is the address of a signal handler, and *sig* is not `SIGILL`, `SIGTRAP`, or `SIGPWR`, the system first sets the signal's disposition to `SIG_DFL` before executing the signal handler. If `sigset()` is used and *disp* is the address of a signal handler, the system adds *sig* to the calling process's signal mask before executing the signal handler; when the signal handler returns, the system restores the calling process's signal mask to its state prior to the delivery of the signal. In addition, if `sigset()` is used and *disp* is equal to `SIG_HOLD`, *sig* is added to the calling process's signal mask and the signal's disposition remains unchanged.

The `sighold()` function adds *sig* to the calling process's signal mask.

The `sigrelse()` function removes *sig* from the calling process's signal mask.

The `sigignore()` function sets the disposition of *sig* to `SIG_IGN`.

The `sigpause()` function removes *sig* from the calling process's signal mask and suspends the calling process until a signal is received.

Return Values Upon successful completion, `signal()` returns the signal's previous disposition. Otherwise, it returns `SIG_ERR` and sets `errno` to indicate the error.

Upon successful completion, `sigset()` returns `SIG_HOLD` if the signal had been blocked or the signal's previous disposition if it had not been blocked. Otherwise, it returns `SIG_ERR` and sets `errno` to indicate the error.

Upon successful completion, `sighold()`, `sigrelse()`, `sigignore()`, and `sigpause()`, return `0`. Otherwise, they return `−1` and set `errno` to indicate the error.

Errors These functions fail if:

`EINTR` A signal was caught during the execution `sigpause()`.

`EINVAL` The value of the *sig* argument is not a valid signal or is equal to `SIGKILL` or `SIGSTOP`.

Usage The `sighold()` function used in conjunction with `sigrelse()` or `sigpause()` may be used to establish critical regions of code that require the delivery of a signal to be temporarily deferred.

If `signal()` or `sigset()` is used to set `SIGCHLD`'s disposition to a signal handler, `SIGCHLD` will not be sent when the calling process's children are stopped or continued.

If any of the above functions are used to set `SIGCHLD`'s disposition to `SIG_IGN`, the calling process's child processes will not create zombie processes when they terminate (see `exit(2)`). If the calling process subsequently waits for its children, it blocks until all of its children terminate; it then returns −1 with `errno` set to `ECHILD` (see `wait(3C)` and `waitid(2)`).

The system guarantees that if more than one instance of the same signal is generated to a process, at least one signal will be received. It does not guarantee the reception of every generated signal.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `exit(2)`, `kill(2)`, `pause(2)`, `sigaction(2)`, `sigsend(2)`, `waitid(2)`, `signal.h(3HEAD)`, `wait(3C)`, `attributes(5)`, `standards(5)`

Name sigqueue – queue a signal to a process

Synopsis

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

int sigqueue(pid_t pid, int signo, const union sigval value);
```

Description The `sigqueue()` function causes the signal specified by *signo* to be sent with the value specified by *value* to the process specified by *pid*. If *signo* is 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of *pid*.

The conditions required for a process to have permission to queue a signal to another process are the same as for the `kill(2)` function.

The `sigqueue()` function returns immediately. If `SA_SIGINFO` is set for *signo* and if the resources were available to queue the signal, the signal is queued and sent to the receiving process. If `SA_SIGINFO` is not set for *signo*, then *signo* is sent at least once to the receiving process; it is unspecified whether *value* will be sent to the receiving process as a result of this call.

If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked for the calling thread and if no other thread has *signo* unblocked or is waiting in a `sigwait(2)` function for *signo*, either *signo* or at least the pending, unblocked signal will be delivered to the calling thread before the `sigqueue()` function returns. Should any of multiple pending signals in the range `SIGRTMIN` to `SIGRTMAX` be selected for delivery, it will be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

Return Values Upon successful completion, the specified signal will have been queued, and the `sigqueue()` function returns `0`. Otherwise, the function returns `−1` and sets `errno` to indicate the error.

Errors The `sigqueue()` function will fail if:

`EAGAIN` No resources are available to queue the signal. The process has already queued `SIGQUEUE_MAX` signals that are still pending at the receiver(s), or a system wide resource limit has been exceeded.

`EINVAL` The value of *signo* is an invalid or unsupported signal number.

`ENOSYS` The `sigqueue()` function is not supported by the system.

`EPERM` The process does not have the appropriate privilege to send the signal to the receiving process.

`ESRCH` The process *pid* does not exist.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See standards(5).

See Also kill(2), siginfo.h(3HEAD), signal.h(3HEAD), sigwaitinfo(3C), attributes(5), standards(5)

Name sigsetops, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate sets of signals

Synopsis

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);

int sigismember(sigset_t *set, int signo);
```

Description These functions manipulate `sigset_t` data types, representing the set of signals supported by the implementation.

The `sigemptyset()` function initializes the set pointed to by *set* to exclude all signals defined by the system.

The `sigfillset()` function initializes the set pointed to by *set* to include all signals defined by the system.

The `sigaddset()` function adds the individual signal specified by the value of *signo* to the set pointed to by *set*.

The `sigdelset()` function deletes the individual signal specified by the value of *signo* from the set pointed to by *set*.

The `sigismember()` function checks whether the signal specified by the value of *signo* is a member of the set pointed to by *set*.

Any object of type *sigset_t* must be initialized by applying either `sigemptyset()` or `sigfillset()` before applying any other operation.

Return Values Upon successful completion, the `sigismember()` function returns `1` if the specified signal is a member of the specified set, or `0` if it is not.

Upon successful completion, the other functions return `0`. Otherwise −`1` is returned and `errno` is set to indicate the error.

Errors The `sigaddset()`, `sigdelset()`, and `sigismember()` functions will fail if:

`EINVAL` The value of the *signo* argument is not a valid signal number.

The `sigfillset()` function will fail if:

`EFAULT` The *set* argument specifies an invalid address.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See standards(5).

See Also sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2), signal.h(3HEAD), attributes(5), standards(5)

Name sigstack – set and/or get alternate signal stack context

Synopsis

```
#include <signal.h>

int sigstack(struct sigstack *ss, struct sigstack *oss);
```

Description The `sigstack()` function allows the calling process to indicate to the system an area of its address space to be used for processing signals received by the process.

If the *ss* argument is not a null pointer, it must point to a `sigstack` structure. The length of the application-supplied stack must be at least `SIGSTKSZ` bytes. If the alternate signal stack overflows, the resulting behavior is undefined. (See `USAGE` below.)

- The value of the `ss_onstack` member indicates whether the process wants the system to use an alternate signal stack when delivering signals.
- The value of the `ss_sp` member indicates the desired location of the alternate signal stack area in the process' address space.
- If the *ss* argument is a null pointer, the current alternate signal stack context is not changed.

If the *oss* argument is not a null pointer, it points to a `sigstack` structure in which the current alternate signal stack context is placed. The value stored in the `ss_onstack` member of *oss* will be non-zero if the process is currently executing on the alternate signal stack. If the *oss* argument is a null pointer, the current alternate signal stack context is not returned.

When a signal's action indicates its handler should execute on the alternate signal stack (specified by calling `sigaction(2)`), `sigstack()` checks to see if the process is currently executing on that stack. If the process is not currently executing on the alternate signal stack, the system arranges a switch to the alternate signal stack for the duration of the signal handler's execution.

After a successful call to one of the *exec* functions, there are no alternate signal stacks in the new process image.

Return Values Upon successful completion, `sigstack()` returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `sigstack()` function will fail if:

`EPERM` An attempt was made to modify an active stack.

Usage A portable application, when being written or rewritten, should use `sigaltstack(2)` instead of `sigstack()`.

The direction of stack growth is not indicated in the historical definition of `struct sigstack`. The only way to portably establish a stack pointer is for the application to determine stack growth direction, or to allocate a block of storage and set the stack pointer to the middle.

`sigstack()` may assume that the size of the signal stack is `SIGSTKSZ` as found in `<signal.h>`. An application that would like to specify a signal stack size other than `SIGSTKSZ` should use `sigaltstack(2)`.

Applications should not use `longjmp(3C)` to leave a signal handler that is running on a stack established with `sigstack()`. Doing so may disable future use of the signal stack. For abnormal exit from a signal handler, `siglongjmp(3C)`, `setcontext(2)`, or `swapcontext(3C)` may be used. These functions fully support switching from one stack to another.

The `sigstack()` function requires the application to have knowledge of the underlying system's stack architecture. For this reason, `sigaltstack(2)` is recommended over this function.

See Also `fork(2)`, `_longjmp(3C)`, `longjmp(3C)`, `setjmp(3C)`, `sigaltstack(2)`, `siglongjmp(3C)`, `sigsetjmp(3C)`

Name sigwaitinfo, sigtimedwait – wait for queued signals

Synopsis

```
#include <signal.h>

int sigwaitinfo(const sigset_t *restrict set,
     siginfo_t *restrict info);

int sigtimedwait(const sigset_t *restrict set,
     siginfo_t *restrict info,
     const struct timespec *restrict timeout);
```

Description The `sigwaitinfo()` function selects the pending signal from the set specified by `set`. Should any of multiple pending signals in the range `SIGRTMIN` to `SIGRTMAX` be selected, it will be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified. If no signal in `set` is pending at the time of the call, the calling thread is suspended until one or more signals in `set` become pending or until it is interrupted by an unblocked, caught signal.

The `sigwaitinfo()` function behaves the same as the `sigwait(2)` function if the *info* argument is `NULL`. If the *info* argument is non-`NULL`, the `sigwaitinfo()` function behaves the same as `sigwait(2)`, except that the selected signal number is stored in the *si_signo* member, and the cause of the signal is stored in the *si_code* member. If any value is queued to the selected signal, the first such queued value is dequeued and, if the *info* argument is non-`NULL`, the value is stored in the *si_value* member of *info*. The system resource used to queue the signal will be released and made available to queue other signals. If no value is queued, the content of the *si_value* member is undefined. If no further signals are queued for the selected signal, the pending indication for that signal will be reset. If the value of the `si_code` member is `SI_NOINFO`, only the `si_signo` member of `siginfo_t` is meaningful, and the value of all other members is unspecified.

The `sigtimedwait()` function behaves the same as `sigwaitinfo()` except that if none of the signals specified by `set` are pending, `sigtimedwait()` waits for the time interval specified in the `timespec` structure referenced by *timeout*. If the `timespec` structure pointed to by *timeout* is zero-valued and if none of the signals specified by `set` are pending, then `sigtimedwait()` returns immediately with an error. If *timeout* is the `NULL` pointer, the behavior is unspecified.

If, while `sigwaitinfo()` or `sigtimedwait()` is waiting, a signal occurs which is eligible for delivery (that is, not blocked by the process signal mask), that signal is handled asynchronously and the wait is interrupted.

Return Values Upon successful completion (that is, one of the signals specified by *set* is pending or is generated) `sigwaitinfo()` and `sigtimedwait()` will return the selected signal number. Otherwise, the function returns −1 and sets `errno` to indicate the error.

Errors The `sigwaitinfo()` and `sigtimedwait()` functions will fail if:

`EINTR` The wait was interrupted by an unblocked, caught signal.

ENOSYS The `sigwaitinfo()` and `sigtimedwait()` functions are not supported.

The `sigtimedwait()` function will fail if:

EAGAIN No signal specified by `set` was generated within the specified timeout period.

The `sigwaitinfo()` and `sigtimedwait()` functions may fail if:

EFAULT The *set*, *info*, or *timeout* argument points to an invalid address.

The `sigtimedwait()` function may fail if:

EINVAL The *timeout* argument specified a `tv_nsec` value less than zero or greater than or equal to 1000 million. The system only checks for this error if no signal is pending in *set* and it is necessary to wait.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Safe
Standard	See `standards(5)`.

See Also `time(2)`, `sigqueue(3C)`, `siginfo.h(3HEAD)`, `signal.h(3HEAD)`, `time.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name sleep – suspend execution for an interval of time

Synopsis

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Description The caller is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested because any caught signal will terminate the `sleep()` following execution of that signal's catching routine. The suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system. The value returned by `sleep()` will be the "unslept" amount (the requested time minus the time actually slept) if the caller incurred premature arousal because of a caught signal.

The use of the `sleep()` function has no effect on the action or blockage of any signal. In a multithreaded process, only the invoking thread is suspended from execution.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `nanosleep(3C)`, `attributes(5)`, `standards(5)`

Name smt_pause – busy wait idle function

Synopsis

```
#include <synch.h>

void smt_pause(void);
```

Description The `smt_pause()` function delays for a short implementation-dependent period before returning to the caller, consuming as few processor resources as possible. This primitive is recommended for use in busy wait loops to lessen the impact the loop has on the rest of the system. For example, on CMT systems it enables other hardware strands sharing the core to go faster during the busy wait.

Usage Typical usage is as follows:

```
volatile int *wait;
while (*wait == 1)
      smt_pause();
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `attributes(5)`

Name ssignal, gsignal – software signals

Synopsis

```
#include <signal.h>

void(*ssignal (int sig, int (*action)(int)))(int);

int gsignal(int sig);
```

Description The `ssignal()` and `gsignal()` functions implement a software facility similar to `signal(3C)`. This facility is made available to users for their own purposes.

ssignal( ) Software signals made available to users are associated with integers in the inclusive range 1 through 17. A call to `ssignal()` associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to `gsignal()`. Raising a software signal causes the action established for that signal to be taken.

The first argument to `ssignal()` is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) *action function* or one of the manifest constants `SIG_DFL` (default) or `SIG_IGN` (ignore). The `ssignal()` function returns the action previously established for that signal type; if no action has been established or the signal number is illegal, `ssignal()` returns `SIG_DFL`.

gsignal( ) The `gsignal()` raises the signal identified by its argument, *sig*.

If an action function has been established for *sig*, then that action is reset to `SIG_DFL` and the action function is entered with argument *sig*. The `gsignal()` function returns the value returned to it by the action function.

If the action for *sig* is `SIG_IGN`, `gsignal()` returns the value 1 and takes no other action.

If the action for *sig* is `SIG_DFL`, `gsignal()` returns the value 0 and takes no other action.

If *sig* has an illegal value or no action was ever specified for *sig*, `gsignal()` returns the value 0 and takes no other action.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `raise(3C)`, `signal(3C)`, `attributes(5)`

Name stack_getbounds – retrieve stack boundaries

Synopsis

```
#include <ucontext.h>

int stack_getbounds(stack_t *sp);
```

Description The `stack_getbounds()` function retrieves the stack boundaries that the calling thread is currently operating on. If the thread is currently operating on the alternate signal stack, this function will retrieve the bounds of that stack.

If successful, `stack_getbounds()` sets the `ss_sp` member of the `stack_t` structure pointed to by *sp* to the base of the stack region and the `ss_size` member to its size (maximum extent) in bytes. The `ss_flags` member is set to `SS_ONSTACK` if the calling thread is executing on its alternate signal stack, and zero otherwise.

Return Values Upon successful completion, `stack_getbounds()` returns 0. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `stack_getbounds()` function will fail if:

`EFAULT` The *sp* argument does not refer to a valid address.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also `getustack(2)`, `sigaction(2)`, `sigaltstack(2)`, `stack_setbounds(3C)`, `attributes(5)`

Name _stack_grow – express an intention to extend the stack

Synopsis

```
#include <ucontext.h>

void *_stack_grow(void *addr);
```

Description The `_stack_grow()` function indicates to the system that the stack is about to be extended to the address specified by *addr*. If extending the stack to this address would violate the stack boundaries as retreived by `stack_getbounds(3C)`, a `SIGSEGV` is raised.

If the disposition of `SIGSEGV` is `SIG_DFL`, the process is terminated and a core dump is generated. If the application has installed its own `SIGSEGV` handler to run on the alternate signal stack, the signal information passed to the handler will be such that a call to `stack_violation(3C)` with these parameters returns 1.

The *addr* argument is a biased stack pointer value. See the Solaris 64–bit Developer's Guide.

This function has no effect if the specified address, *addr*, is within the bounds of the current stack.

Return Values If the `_stack_grow()` function succeeds and does not detect a stack violation, it returns *addr*.

Errors No errors are defined.

Usage The `_stack_grow()` function does not actually adjust the stack pointer register. The caller is responsible for manipulating the stack pointer register once `_stack_grow()` returns.

The `_stack_grow()` function is typically invoked by code created by the compilation environment prior to executing code that modifies the stack pointer. It can also be used by hand-written assembly routines to allocate stack-based storage safely.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also `stack_getbounds(3C)`, `stack_inbounds(3C)`, `stack_violation(3C)`, `attributes(5)`

Solaris 64–bit Developer's Guide

Name stack_inbounds – determine if address is within stack boundaries

Synopsis

```
#include <ucontext.h>

int stack_inbounds(void *addr);
```

Description The `stack_inbounds()` function returns a boolean value indicating whether the address specified by *addr* is within the boundaries of the stack of the calling thread. The address is compared to the stack boundary information returned by a call to `stack_getbounds(3C)`.

Return Values The `stack_inbounds()` function returns 0 to indicate that *addr* is not within the current stack bounds, or a non-zero value to indicate that *addr* is within the stack bounds.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also `stack_getbounds(3C)`, `attributes(5)`

Name stack_setbounds – update stack boundaries

Synopsis

```
#include <ucontext.h>

int stack_setbounds(const stack_t *sp);
```

Description The `stack_setbounds()` function updates the current base and bounds of the stack for the current thread to the bounds specified by the `stack_t` structure pointed to by *sp*. The `ss_sp` member refers to the virtual address of the base of the stack memory. The `ss_size` member refers to the size of the stack in bytes. The `ss_flags` member must be set to 0.

Return Values Upon successful completion, `stack_setbounds()` returns 0. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `stack_setbounds()` function will fail if:

`EFAULT` The *sp* argument does not refer to a valid address or the `ss_sp` member of the `stack_t` structure pointed to by *sp* points to an illegal address.

`EINVAL` The `ss_sp` member of the `stack_t` structure pointed to by sp is not properly aligned, the `ss_size` member is too small or is not properly aligned, or the `ss_flags` member is non-zero.

Usage The `stack_setbounds()` function is intended for use by applications that are managing their own alternate stacks.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also `getustack(2)`, `_stack_grow(3C)`, `stack_getbounds(3C)`, `stack_inbounds(3C)`, `stack_violation(3C)`, `attributes(5)`

Name stack_violation – determine stack boundary violation event

Synopsis

```
#include <ucontext.h>

int stack_violation(int sig, const siginfo_t *sip,
     const ucontext_t *ucp);
```

Description The `stack_violation()` function returns a boolean value indicating whether the signal, *sig*, and accompanying signal information, *sip*, and saved context, *ucp*, represent a stack boundary violation event or a stack overflow.

Return Values The `stack_violation()` function returns 0 if the signal does not represent a stack boundary violation event and 1 if the signal does represent a stack boundary violation event.

Errors No errors are defined.

Examples **EXAMPLE 1** Set up a signal handler to run on an alternate stack.

The following example sets up a signal handler for `SIGSEGV` to run on an alternate signal stack. For each signal it handles, the handler emits a message to indicate if the signal was produced due to a stack boundary violation.

```
#include <stdlib.h>
#include <unistd.h>
#include <ucontext.h>
#include <signal.h>


static void
handler(int sig, siginfo_t *sip, void *p)
{
        ucontext_t *ucp = p;
        const char *str;

        if (stack_violation(sig, sip, ucp))
                str = "stack violation.\n";
        else
                str = "no stack violation.\n";

        (void) write(STDERR_FILENO, str, strlen(str));

        exit(1);
}

int
main(int argc, char **argv)
{
        struct sigaction sa;
        stack_t altstack;
```

EXAMPLE 1 Set up a signal handler to run on an alternate stack. *(Continued)*

```
        altstack.ss_size = SIGSTKSZ;
        altstack.ss_sp = malloc(SIGSTKSZ);
        altstack.ss_flags = 0;

        (void) sigaltstack(&altstack, NULL);

        sa.sa_sigaction = handler;
        (void) sigfillset(&sa.sa_mask);
        sa.sa_flags = SA_ONSTACK | SA_SIGINFO;
        (void) sigaction(SIGSEGV, &sa, NULL);

        /*
         * The application is now set up to use stack_violation(3C).
         */

        return (0);
}
```

Usage

An application typically uses `stack_violation()` in a signal handler that has been installed for `SIGSEGV` using `sigaction(2)` with the `SA_SIGINFO` flag set and is configured to run on an alternate signal stack.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also

`sigaction(2)`, `sigaltstack(2)`, `stack_getbounds(3C)`, `stack_inbounds(3C)`, `stack_setbounds(3C)`, `attributes(5)`

Name stdio – standard buffered input/output package

Synopsis

```
#include <stdio.h>

extern FILE *stdin;

extern FILE *stdout;

extern FILE *stderr;
```

Description The standard I/O functions described in section 3C of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros `getc()` and `putc()` handle characters quickly. The macros `getchar(3C)` and `putchar(3C)`, and the higher-level routines `fgetc(3C)`, `fgets(3C)`, `fprintf(3C)`, `fputc(3C)`, `fputs(3C)`, `fread(3C)`, `fscanf(3C)`, `fwrite(3C)`, `gets(3C)`, `getw(3C)`, `printf(3C)`, `puts(3C)`, `putw(3C)`, and `scanf(3C)` all use or act as if they use `getc()` and `putc()`; they can be freely intermixed.

A file with associated buffering is called a *stream* (see `Intro(3)`) and is declared to be a pointer to a defined type `FILE`. The `fopen(3C)` function creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the `<stdio.h>` header and associated with the standard open files:

`stdin`	standard input file
`stdout`	standard output file
`stderr`	standard error file

The following symbolic values in `<unistd.h>` define the file descriptors that will be associated with the C-language `stdin`, `stdout` and `stderr` when the application is started:

`STDIN_FILENO`	Standard input value	0	`stdin`
`STDOUT_FILENO`	Standard output value	1	`stdout`
`STDERR_FILENO`	Standard error value	2	`stderr`

The constant `NULL` designates a null pointer.

The integer-constant `EOF` is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

The integer constant `BUFSIZ` specifies the size of the buffers used by the particular implementation.

The integer constant `FILENAME_MAX` specifies the number of bytes needed to hold the longest pathname of a file allowed by the implementation. If the system does not impose a maximum limit, this value is the recommended size for a buffer intended to hold a file's pathname.

The integer constant FOPEN_MAX specifies the minimum number of files that the implementation guarantees can be open simultaneously. Note that no more than 255 files may be opened using fopen(), and only file descriptors 0 through 255 can be used in a stream.

The functions and constants mentioned in the entries of section 3S of this manual are declared in that header and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): getc(), getchar(), putc(), putchar(), ferror(3C), feof(3C), clearerr(3C), and fileno(3C). There are also function versions of getc(), getchar(), putc(), putchar(), ferror(), feof(), clearerr(), and fileno().

Output streams, with the exception of the standard error stream stderr, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream stderr is by default unbuffered, but use of freopen() (see fopen(3C)) will cause it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). The setbuf() or setvbuf() functions (both described on the setbuf(3C) manual page) may be used to change the stream's buffering strategy.

Interactions of Other FILE-Type C Functions

A single open file description can be accessed both through streams and through file descriptors. Either a file descriptor or a stream will be called a *handle* on the open file description to which it refers; an open file description may have several handles.

Handles can be created or destroyed by user action without affecting the underlying open file description. Some of the ways to create them include fcntl(2), dup(2), fdopen(3C), fileno(3C) and fork(2) (which duplicates existing ones into new processes). They can be destroyed by at least fclose(3C) and close(2), and by the exec functions (see exec(2)), which close some file descriptors and destroy streams.

A file descriptor that is never used in an operation and could affect the file offset (for example read(2), write(2), or lseek(2)) is not considered a handle in this discussion, but could give rise to one (as a consequence of fdopen(), dup(), or fork(), for example). This exception does include the file descriptor underlying a stream, whether created with fopen() or fdopen(), as long as it is not used directly by the application to affect the file offset. (The read() and write() functions implicitly affect the file offset; lseek() explicitly affects it.)

If two or more handles are used, and any one of them is a stream, their actions shall be coordinated as described below. If this is not done, the result is undefined.

A handle that is a stream is considered to be closed when either an fclose() or freopen(3C) is executed on it (the result of freopen() is a new stream for this discussion, which cannot be a handle on the same open file description as its previous value) or when the process owning

that stream terminates the exit(2) or abort(3C). A file descriptor is closed by close(), _exit() (see exit(2)), or by one of the exec functions when FD_CLOEXEC is set on that file descriptor.

For a handle to become the active handle, the actions below must be performed between the last other user of the first handle (the current active handle) and the first other user of the second handle (the future active handle). The second handle then becomes the active handle. All activity by the application affecting the file offset on the first handle shall be suspended until it again becomes the active handle. (If a stream function has as an underlying function that affects the file offset, the stream function will be considered to affect the file offset. The underlying functions are described below.)

The handles need not be in the same process for these rules to apply. Note that after a fork(), two handles exist where one existed before. The application shall assure that, if both handles will ever be accessed, that they will both be in a state where the other could become the active handle first. The application shall prepare for a fork() exactly as if it were a change of active handle. (If the only action performed by one of the processes is one of the exec functions or _exit(), the handle is never accessed in that process.)

1. For the first handle, the first applicable condition below shall apply. After the actions required below are taken, the handle may be closed if it is still open.
   a. If it is a file descriptor, no action is required.
   b. If the only further action to be performed on any handle to this open file description is to close it, no action need be taken.
   c. If it is a stream that is unbuffered, no action need be taken.
   d. If it is a stream that is line-buffered and the last character written to the stream was a newline (that is, as if a putc('\n') was the most recent operation on that stream), no action need be taken.
   e. If it is a stream that is open for writing or append (but not also open for reading), either an fflush(3C) shall occur or the stream shall be closed.
   f. If the stream is open for reading and it is at the end of the file (feof(3C) is true), no action need be taken.
   g. If the stream is open with a mode that allows reading and the underlying open file description refers to a device that is capable of seeking, either an fflush() shall occur or the stream shall be closed.
   h. Otherwise, the result is undefined.
2. For the second handle: if any previous active handle has called a function that explicitly changed the file offset, except as required above for the first handle, the application shall perform an lseek() or an fseek(3C) (as appropriate to the type of the handle) to an appropriate location.

3. If the active handle ceases to be accessible before the requirements on the first handle above have been met, the state of the open file description becomes undefined. This might occur, for example, during a `fork()` or an `_exit()`.
4. The `exec` functions shall be considered to make inaccessible all streams that are open at the time they are called, independent of what streams or file descriptors may be available to the new process image.
5. Implementation shall assure that an application, even one consisting of several processes, shall yield correct results (no data is lost or duplicated when writing, all data is written in order, except as requested by seeks) when the rules above are followed, regardless of the sequence of handles used. If the rules above are not followed, the result is unspecified. When these rules are followed, it is implementation defined whether, and under what conditions, all input is seen exactly once.

Use of stdio in Multithreaded Applications

All the `stdio` functions are safe unless they have the `_unlocked` suffix. Each `FILE` pointer has its own lock to guarantee that only one thread can access it. In the case that output needs to be synchronized, the lock for the `FILE` pointer can be acquired before performing a series of `stdio` operations. For example:

```
FILE iop;
flockfile(iop);
fprintf(iop, "hello ");
fprintf(iop, "world);
fputc(iop, 'a');
funlockfile(iop);
```

will print everything out together, blocking other threads that might want to write to the same file between calls to `fprintf()`.

An unlocked interface is available in case performace is an issue. For example:

```
flockfile(iop);
while (!feof(iop)) {
        *c++ = getc_unlocked(iop);
}
funlockfile(iop);
```

Return Values

Invalid stream pointers usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

See Also

`close(2)`, `lseek(2)`, `open(2)`, `pipe(2)`, `read(2)`, `write(2)`, `ctermid(3C)`, `cuserid(3C)`, `fclose(3C)`, `ferror(3C)`, `fopen(3C)`, `fread(3C)`, `fseek(3C)`, `flockfile(3C)`, `getc(3C)`, `gets(3C)`, `popen(3C)`, `printf(3C)`, `putc(3C)`, `puts(3C)`, `scanf(3C)`, `setbuf(3C)`, `system(3C)`, `tmpfile(3C)`, `tmpnam(3C)`, `ungetc(3C)`

Name str2sig, sig2str – translation between signal name and signal number

Synopsis

```
#include <signal.h>

int str2sig(const char *str, int *signum);

int sig2str(int signum, char *str);
```

Description The `str2sig()` function translates the signal name *str* to a signal number, and stores that result in the location referenced by *signum*. The name in *str* can be either the symbol for that signal, without the "SIG" prefix, or a decimal number. All the signal symbols defined in `<sys/signal.h>` are recognized. This means that both "CLD" and "CHLD" are recognized and return the same signal number, as do both "POLL" and "IO". For access to the signals in the range `SIGRTMIN` to `SIGRTMAX`, the first four signals match the strings "RTMIN", "RTMIN+1", "RTMIN+2", and "RTMIN+3" and the last four match the strings "RTMAX-3", "RTMAX-2", "RTMAX-1", and "RTMAX".

The `sig2str()` function translates the signal number *signum* to the symbol for that signal, without the "SIG" prefix, and stores that symbol at the location specified by *str*. The storage referenced by *str* should be large enough to hold the symbol and a terminating null byte. The symbol `SIG2STR_MAX` defined by `<signal.h>` gives the maximum size in bytes required.

Return Values The `str2sig()` function returns `0` if it recognizes the signal name specified in *str*; otherwise, it returns −1.

The `sig2str()` function returns `0` if the value *signum* corresponds to a valid signal number; otherwise, it returns −1.

Examples **EXAMPLE 1** A sample program using the `str2sig()` function.

```
int i;
char buf[SIG2STR_MAX];     /*storage for symbol */

str2sig("KILL",&i);        /*stores 9 in i */
str2sig("9", &i);          /* stores 9 in i */
sig2str(SIGKILL,buf);      /* stores "KILL" in buf */
sig2str(9,buf);            /* stores "KILL" in buf */
```

See Also `kill(1)`, `strsignal(3C)`

Name strcoll – string collation

Synopsis

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

Description Both `strcoll()` and `strxfrm(3C)` provide for locale-specific string sorting. `strcoll()` is intended for applications in which the number of comparisons per string is small. When strings are to be compared a number of times, `strxfrm(3C)` is a more appropriate function because the transformation process occurs only once.

The `strcoll()` function does not change the setting of `errno` if successful.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to 0, then call `strcoll()`, then check `errno`.

Return Values Upon successful completion, `strcoll()` returns an integer greater than, equal to, or less than zero in direct correlation to whether string *s1* is greater than, equal to, or less than the string *s2*. The comparison is based on strings interpreted as appropriate to the program's locale for category `LC_COLLATE` (see `setlocale(3C)`).

On error, `strcoll()` may set `errno`, but no return value is reserved to indicate an error.

Errors The `strcoll()` function may fail if:

`EINVAL` The *s1* or *s2* arguments contain characters outside the domain of the collating sequence.

Files `/usr/lib/locale/`*locale*`/`*locale*`.so.*` `LC_COLLATE` database for *locale*

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

The `strcoll()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

See Also `localedef(1)`, `setlocale(3C)`, `string(3C)`, `strxfrm(3C)`, `wsxfrm(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Name strerror, strerror_r – get error message string

Synopsis

```
#include <string.h>

char *strerror(int errnum);

int strerror_r(int errnum, char *strerrbuf, size_t  buflen);
```

Description The `strerror()` function maps the error number in *errnum* to an error message string, and returns a pointer to that string. It uses the same set of error messages as `perror(3C)`. The returned string should not be overwritten.

The `strerror_r()` function maps the error number in *errnum* to anerror message string and returns the string in the buffer pointed to by *strerrbuf* with length *buflen*.

Return Values Upon successful completion, `strerror()` returns a pointer to the generated message string. Otherwise, it sets `errno` and returns a pointer to an error message string. It returns the string "Unknown error" if *errnum* is not a valid error number.

Upon successful completion, `strerror_r()` returns 0. Otherwise it sets `errno` and returns the value of `errno` to indicate the error. It returns the string "Unknown error" in the buffer pointed to by *strerrbuf* if *errnum* is not a valid error number.

Errors These functions may fail if:

`EINVAL` The value of *errnum* is not a valid error number.

The `strerror_r()` function may fail if:

`ERANGE` The *buflen* argument specifies insufficient storage to contain the generated message string.

Usage Messages returned from these functions are in the native language specified by the `LC_MESSAGES` locale category. See `setlocale(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `gettext(3C)`, `perror(3C)`, `setlocale(3C)`, `attributes(5)`, `standards(5)`

Name strfmon – convert monetary value to string

Synopsis

```
#include <monetary.h>

ssize_t strfmon(char *restrict s, size_t maxsize,
     const char *restrict format...);
```

Description The `strfmon()` function places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The format is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in the fetching of zero or more arguments which are converted and formatted. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

A conversion specification consists of the following sequence:

- a % character
- optional flags
- optional field width
- optional left precision
- optional right precision
- a required conversion character that determines the conversion to be performed.

Flags One or more of the following optional flags can be specified to control the conversion:

=*f*	An = followed by a single character *f* which is used as the numeric fill character. The fill character must be representable in a single byte in order to work with precision and width counts. The default numeric fill character is the space character. This flag does not affect field width filling which always uses the space character. This flag is ignored unless a left precision (see below) is specified.
^	Do not format the currency amount with grouping characters. The default is to insert the grouping characters if defined for the current locale.
+ or (	Specify the style of representing positive and negative currency amounts. Only one of '+' or '(' may be specified. If '+' is specified, the locale's equivalent of + and '–' are used. If '(' is specified, negative amounts are enclosed within parentheses. If neither flag is specified, the '+' style is used.
!	Suppress the currency symbol from the output conversion.
–	Specify the alignment. If this flag is present all fields are left-justified (padded to the right) rather than right-justified.

Field Width *w* A decimal digit string *w* specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the flag '–' is specified). The default is zero.

Left Precision *#n* A '#' followed by a decimal digit string *n* specifying a maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to the `strfmon()` aligned in the same columns. It can also be used to fill unused positions with a special character as in `$***123.45`. This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more than *n* digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character (see the *=f* flag above).

If grouping has not been suppressed with the '^' flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters even if the fill character is a digit.

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with space characters to make their positive and negative formats an equal length.

Right Precision *.p* A period followed by a decimal digit string *p* specifying the number of digits after the radix character. If the value of the right precision *p* is zero, no radix character appears. If a right precision is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

Conversion Characters The conversion characters and their meanings are:

i The `double` argument is formatted according to the locale's international currency format (for example, in the U.S.A.: USD 1,234.56).

n The `double` argument is formatted according to the locale's national currency format (for example, in the U.S.A.: $1,234.56).

% Convert to a % no argument is converted. The entire conversion specification must be %%.

Locale Information The `LC_MONETARY` category of the program's locale affects the behavior of this function including the monetary radix character (which may be different from the numeric radix character affected by the `LC_NUMERIC` category), the grouping separator, the currency symbols and formats. The international currency symbol should be in conformance with the ISO 4217: 1987 standard.

Return Values If the total number of resulting bytes (including the terminating null byte) is not more than *maxsize*, `strfmon()` returns the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, `−1` is returned, the contents of the array are indeterminate, and `errno` is set to indicate the error.

Errors The `strfmon()` function will fail if:

`ENOSYS` The function is not supported.

`E2BIG` Conversion stopped due to lack of space in the buffer.

Usage The behavior of `strfmon()` in an SUSv3–conforming application differs from its behavior in a non-conforming application as follows:

- With the conversion '`i`', `strfmon()` uses information set to `int_p_cs_precedes`, `int_n_cs_precedes`, `int_p_sep_by_space`, `int_n_sep_by_space`, `int_p_sign_posn`, and `int_n_sign_posn` of the current locale instead of `p_cs_precedes`, `n_cs_precedes`, `p_sep_by_space`, `n_sep_by_space`, `p_sign_posn`, and `n_sign_posn`, respectively.
- With the conversion '`i`', `strfmon()` uses the fourth character of the string set to `int_curr_symbol` of the current locale instead of a space for`int_p_sep_by_space` and `int_n_sep_by_space`.
- When the value of `p_sep_by_space`, `n_sep_by_space`, `int_p_sep_by_space`, or `int_n_sep_by_space` is set to 2 in the current locale, `strfmon()` separates the currency symbol from the sign string by a space, if adjacent; otherwise, `strfmon()` separates the sign string from the value by a space.

Examples **EXAMPLE 1** A sample output of `strfmon()`.

Given a locale for the U.S.A. and the values 123.45, –123.45, and 3456.781:

Conversion Specification	Output	Comments
%n	$123.45	default formatting
	-$123.45	
	$3,456.78	
%11n	$123.45	right align within an 11
	-$123.45	character field
	$3,456.78	
%#5n	$123.45	aligned columns for values
	-$123.45	up to 99,999
	$3,456.78	
%=*#5n	$***123.45	specify a fill character
	-$***123.45	

EXAMPLE 1 A sample output of `strfmon()`. *(Continued)*

Conversion Specification	Output	Comments
	$*3,456.78	
%=0#5n	$000123.45	fill characters do not use
	-$000123.45	grouping even if the fill
	$03,456.78	character is a digit
%^#5n	$123.45	disable the grouping
	-$123.45	separator
	$3456.78	
%^#5.0n	$123	round off to whole units
	-$123	
	$3457	
%^#5.4n	$123.4500	increase the precision
	-$123.4500	
	$3456.7810	
%(#5n	123.45	use an alternative
	($123.45)	pos/neg style
	$3,456.78	
%!(#5n	123.45	disable the currency
	(123.45)	symbol
	3,456.78	

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

Th `strfmon()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not called to change the locale.

See Also `localeconv(3C)`, `setlocale(3C)`, `attributes(5)`, `standards(5)`

Name strftime, cftime, ascftime – convert date and time to string

Synopsis

```
#include <time.h>

size_t strftime(char *restrict s, size_t maxsize,
     const char *restrict format,
     const struct tm *restrict timeptr);

int cftime(char *s, char *format, const time_t *clock);

int ascftime(char *s, const char *format,
     const struct tm *timeptr);
```

Description The `strftime()`, `ascftime()`, and `cftime()` functions place bytes into the array pointed to by *s* as controlled by the string pointed to by *format*. The *format* string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a '`%`' (percent) character and one or two terminating conversion characters that determine the conversion specification's behavior. All ordinary characters (including the terminating null byte) are copied unchanged into the array pointed to by *s*. If copying takes place between objects that overlap, the behavior is undefined. For `strftime()`, no more than *maxsize* bytes are placed into the array.

If *format* is `(char *)0`, then the locale's default format is used. For `strftime()` the default format is the same as `%c`; for `cftime()` and `ascftime()` the default format is the same as `%C`. `cftime()` and `ascftime()` first try to use the value of the environment variable `CFTIME`, and if that is undefined or empty, the default format is used.

Each conversion specification is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the `LC_TIME` category of the program's locale and by the values contained in the structure pointed to by *timeptr* for `strftime()` and `ascftime()`, and by the time represented by *clock* for `cftime()`.

`%%` Same as `%`.

`%a` Locale's abbreviated weekday name.

`%A` Locale's full weekday name.

`%b` Locale's abbreviated month name.

`%B` Locale's full month name.

Default `%c` Locale's appropriate date and time represented as:

```
%a %b %d %H:%M:%S %Y
```

This is the default behavior as well as standard-conforming behavior for standards first supported by releases prior to Solaris 2.4. See `standards(5)`.

Standard conforming	%c	Locale's appropriate date and time represented as: `%a %b %e %H:%M:%S %Y` This is standard-conforming behavior for standards first supported by Solaris 2.4 through Solaris 10.
Default	%C	Locale's date and time representation as produced by date(1). This is the default behavior as well as standard-conforming behavior for standards first supported by releases prior to Solaris 2.4.
Standard conforming	%C	Century number (the year divided by 100 and truncated to an integer as a decimal number [01,99]). This is standard-conforming behavior for standards first supported by Solaris 2.4 through Solaris 10.
	%d	Day of month [01,31].
	%D	Date as %m/%d/%y.
	%e	Day of month [1,31]; single digits are preceded by a space.
	%F	Equivalent to %Y-%m-%d (the ISO 8601:2000 standard date format).
	%g	Week-based year within century [00,99].
	%G	Week-based year, including the century [0000,9999].
	%h	Locale's abbreviated month name.
	%H	Hour (24-hour clock) [00,23].
	%I	Hour (12-hour clock) [01,12].
	%j	Day number of year [001,366].
	%k	Hour (24-hour clock) [0,23]; single digits are preceded by a space.
	%l	Hour (12-hour clock) [1,12]; single digits are preceded by a space.
	%m	Month number [01,12].
	%M	Minute [00,59].
	%n	Insert a NEWLINE.
	%p	Locale's equivalent of either a.m. or p.m.
	%r	Appropriate time representation in 12-hour clock format with %p.
	%R	Time as %H:%M.
	%S	Seconds [00,60]; the range of values is [00,60] rather than [00,59] to allow for the occasional leap second.

`%t` Insert a TAB.

`%T` Time as `%H:%M:%S`.

`%u` Weekday as a decimal number [1,7], with 1 representing Monday. See `NOTES` below.

`%U` Week number of year as a decimal number [00,53], with Sunday as the first day of week 1.

`%V` The ISO 8601 week number as a decimal number [01,53]. In the ISO 8601 week-based system, weeks begin on a Monday and week 1 of the year is the week that includes both January 4th and the first Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year. See `NOTES` below.

`%w` Weekday as a decimal number [0,6], with 0 representing Sunday.

`%W` Week number of year as a decimal number [00,53], with Monday as the first day of week 1.

`%x` Locale's appropriate date representation.

`%X` Locale's appropriate time representation.

`%y` Year within century [00,99].

`%Y` Year, including the century (for example 1993).

`%z` Replaced by offset from UTC in ISO 8601:2000 standard format (`+hhmm` or `-hhmm`), or by no characters if no time zone is determinable. For example, "-0430" means 4 hours 30 minutes behind UTC (west of Greenwich). If `tm_isdst` is zero, the standard time offset is used. If `tm_isdst` is greater than zero, the daylight savings time offset if used. If `tm_isdst` is negative, no characters are returned.

`%Z` Time zone name or abbreviation, or no bytes if no time zone information exists.

If a conversion specification does not correspond to any of the above or to any of the modified conversion specifications listed below, the behavior is undefined and `0` is returned.

The difference between `%U` and `%W` (and also between modified conversion specifications `%OU` and `%OW`) lies in which day is counted as the first of the week. Week number 1 is the first week in January starting with a Sunday for `%U` or a Monday for `%W`. Week number 0 contains those days before the first Sunday or Monday in January for `%U` and `%W`, respectively.

Modified Conversion Specifications

Some conversion specifications can be modified by the `E` and `O` modifiers to indicate that an alternate format or specification should be used rather than the one normally used by the unmodified conversion specification. If the alternate format or specification does not exist in the current locale, the behavior will be as if the unmodified specification were used.

`%Ec` Locale's alternate appropriate date and time representation.

`%EC`	Name of the base year (period) in the locale's alternate representation.
`%Eg`	Offset from `%EC` of the week-based year in the locale's alternative representation.
`%EG`	Full alternative representation of the week-based year.
`%Ex`	Locale's alternate date representation.
`%EX`	Locale's alternate time representation.
`%Ey`	Offset from `%EC` (year only) in the locale's alternate representation.
`%EY`	Full alternate year representation.
`%Od`	Day of the month using the locale's alternate numeric symbols.
`%Oe`	Same as `%Od`.
`%Og`	Week-based year (offset from `%C`) in the locale's alternate representation and using the locale's alternate numeric symbols.
`%OH`	Hour (24-hour clock) using the locale's alternate numeric symbols.
`%OI`	Hour (12-hour clock) using the locale's alternate numeric symbols.
`%Om`	Month using the locale's alternate numeric symbols.
`%OM`	Minutes using the locale's alternate numeric symbols.
`%OS`	Seconds using the locale's alternate numeric symbols.
`%Ou`	Weekday as a number in the locale's alternate numeric symbols.
`%OU`	Week number of the year (Sunday as the first day of the week) using the locale's alternate numeric symbols.
`%Ow`	Number of the weekday (Sunday=0) using the locale's alternate numeric symbols.
`%OW`	Week number of the year (Monday as the first day of the week) using the locale's alternate numeric symbols.
`%Oy`	Year (offset from `%C`) in the locale's alternate representation and using the locale's alternate numeric symbols.

Selecting the Output Language

By default, the output of `strftime()`, `cftime()`, and `ascftime()` appear in U.S. English. The user can request that the output of `strftime()`, `cftime()`, or `ascftime()` be in a specific language by setting the `LC_TIME` category using `setlocale()`.

Time Zone

Local time zone information is used as though `tzset(3C)` were called.

Return Values

The `strftime()`, `cftime()`, and `ascftime()` functions return the number of characters placed into the array pointed to by *s*, not including the terminating null character. If the total number of resulting characters including the terminating null character is more than *maxsize*, `strftime()` returns `0` and the contents of the array are indeterminate.

Examples

EXAMPLE 1 An example of the `strftime()` function.

The following example illustrates the use of `strftime()` for the POSIX locale. It shows what the string in *str* would look like if the structure pointed to by *tmptr* contains the values corresponding to Thursday, August 28, 1986 at 12:44:36.

```
strftime (str, strsize, "%A %b %d %j", tmptr)
```

This results in *str* containing "Thursday Aug 28 240".

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See below.

For `strftime()`, see `standards(5)`.

See Also

`date(1)`, `ctime(3C)`, `mktime(3C)`, `setlocale(3C)`, `strptime(3C)`, `tzset(3C)`, `TIMEZONE(4)`, `zoneinfo(4)`, `attributes(5)`, `environ(5)`, `standards(5)`

Notes

The conversion specification for `%V` was changed in the Solaris 7 release. This change was based on the public review draft of the ISO C9x standard at that time. Previously, the specification stated that if the week containing 1 January had fewer than four days in the new year, it became week 53 of the previous year. The ISO C9x standard committee subsequently recognized that that specification had been incorrect.

The conversion specifications for `%g`, `%G`, `%Eg`, `%EG`, and `%Og` were added in the Solaris 7 release. This change was based on the public review draft of the ISO C9x standard at that time. These specifications are evolving. If the ISO C9x standard is finalized with a different conclusion, these specifications will change to conform to the ISO C9x standard decision.

The conversion specification for `%u` was changed in the Solaris 8 release. This change was based on the XPG4 specification.

If using the `%Z` specifier and `zoneinfo` timezones and if the input date is outside the range 20:45:52 UTC, December 13, 1901 to 03:14:07 UTC, January 19, 2038, the timezone name may not be correct.

Name string, strcasecmp, strncasecmp, strcat, strncat, strlcat, strchr, strrchr, strchrnul, strcmp, strncmp, strcpy, strncpy, strlcpy, stpcpy, stpncpy, strcspn, strspn, strdup, strndup, strdupa, strndupa, strlen, strnlen, strpbrk, strsep, strstr, strnstr, strcasestr, strtok, strtok_r – string operations

Synopsis

```
#include <strings.h>

int strcasecmp(const char *s1, const char *s2);

int strncasecmp(const char *s1, const char *s2, size_t n);

#include <string.h>

char *strcat(char *restrict s1, const char *restrict s2);

char *strncat(char *restrict s1, const char *restrict s2, size_t n);

size_t strlcat(char *dst, const char *src, size_t dstsize);

char *strchr(const char *s, int c);

char *strrchr(const char *s, int c);

char *strchrnul(const char *s, int c);

int strcmp(const char *s1, const char *s2);

int strncmp(const char *s1, const char *s2, size_t n);

char *strcpy(char *restrict s1, const char *restrict s2);

char *strncpy(char *restrict s1, const char *restrict s2, size_t n);

size_t strlcpy(char *dst, const char *src, size_t dstsize);

char *stpcpy(char *restrict s1, const char *restrict s2);

char *strncpy(char *restrict s1, const char *restrict s2, size_t n);

size_t strcspn(const char *s1, const char *s2);

size_t strspn(const char *s1, const char *s2);

char *strdup(const char *s);

char *strndup(const char *s, size_t size);

char *strdupa(const char *s);

char *strndupa(const char *s, size_t size);

size_t strlen(const char *s);

size_t strnlen(const char *s, size_t n);

char *strpbrk(const char *s1, const char *s2);

char *strsep(char **stringp, const char *delim);

char *strstr(const char *s1, const char *s2);
```

```
char *strnstr(const char *s1, const char *s2, size_t n);

char *strcasestr(const char *s1, const char *s2);

char *strtok(char *restrict s1, const char *restrict s2);

char *strtok_r(char *s1, const char *s2, char **lasts);
```

ISO C++

```
#include <string.h>

const char *strchr(const char *s, int c);

const char *strpbrk(const char *s1, const char *s2);

const char *strrchr(const char *s, int c);

const char *strstr(const char *s1, const char *s2);

#include <cstring>

char *std::strchr(char *s, int c);

char *std::strpbrk(char *s1, const char *s2);

char *std::strrchr(char *s, int c);

char *std::strstr(char *s1, const char *s2);
```

Description

The arguments *s*, *s1*, and *s2* point to strings (arrays of characters terminated by a null character). The `strcat()`, `strncat()`, `strlcat()`, `strcpy()`, `strncpy()`, `strlcpy()`, `strsep()`, `strtok()`, and `strtok_r()` functions all alter their first argument. Additionally, the `strcat()` and `strcpy()` functions do not check for overflow of the array.

`strcasecmp()`, `strncasecmp()`

The `strcasecmp()` and `strncasecmp()` functions are case-insensitive versions of `strcmp()` and `strncmp()` respectively, described below. They ignore differences in case when comparing lower and upper case characters, using the current locale of the process to determine the case of the characters.

`strcat()`, `strncat()`, `strlcat()`

The `strcat()` function appends a copy of string *s2*, including the terminating null character, to the end of string *s1*. The `strncat()` function appends at most *n* characters. Each returns a pointer to the null-terminated result. The initial character of *s2* overrides the null character at the end of *s1*. If copying takes place between objects that overlap, the behavior of `strcat()`, `strncat()`, and `strlcat()` is undefined.

The `strlcat()` function appends at most (*dstsize*-`strlen`(*dst*)-1) characters of *src* to *dst* (*dstsize* being the size of the string buffer *dst*). If the string pointed to by *dst* contains a null-terminated string that fits into *dstsize* bytes when `strlcat()` is called, the string pointed to by *dst* will be a null-terminated string that fits in *dstsize* bytes (including the terminating null character) when it completes, and the initial character of *src* will override the null character at the end of *dst*. If the string pointed to by *dst* is longer than *dstsize* bytes when `strlcat()` is called, the string pointed to by *dst* will not be changed. The function returns `min`{*dstsize*,`strlen`(*dst*)}+`strlen`(*src*). Buffer overflow can be checked as follows:

```
if (strlcat(dst, src, dstsize) >= dstsize)
        return –1;
```

`strchr()`,`strrchr()`, `strchrnul()`

The `strchr()` function returns a pointer to the first occurrence of *c* (converted to a `char`) in string *s*, or a null pointer if *c* does not occur in the string.

The `strrchr()` function returns a pointer to the last occurrence of *c*. The null character terminating a string is considered to be part of the string.

The `strchrnul()` function is similar to `strchr()` except that if *c* is not found in *s*, it returns a pointer to the null byte at the end of *s*, rather than `NULL`.

`strcmp()`,`strncmp()`

The `strcmp()` function compares two strings byte-by-byte, according to the ordering of your machine's character set. The function returns an integer greater than, equal to, or less than 0, if the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared. The `strncmp()` function makes the same comparison but looks at a maximum of *n* bytes. Bytes following a null byte are not compared.

`strcpy()`,`stpcpy()`, `strncpy()`, `stpncpy()`, `strlcpy()`

The `strcpy()` and `stpcpy()` functions copy string *s2* to *s1*, including the terminating null character, stopping after the null character has been copied. The `strcpy()` function returns *s1*. The `stpcpy()` function returns a pointer to the terminating null character copied into the *s1* array.

The `strncpy()` `stpncpy()` and functions copy not more than *n* bytes (bytes that follow a null byte are not copied) from the array pointed to by *s2* to the array pointed to by *s1*. If the array pointed to by *s2* is a string that is shorter than *n* bytes, null bytes are appended to the copy in the array pointed to by *s1*, until *n* bytes in all are written. The `stpcpy()` function returns *s1*. If *s1* contains null bytes, `stpncpy()` returns a pointer to the first such null byte. Otherwise, it returns &*s1*[*n*].

The `strlcpy()` function copies at most *dstsize*–1 characters (*dstsize* being the size of the string buffer *dst*) from *src* to *dst*, truncating *src* if necessary. The result is always null-terminated. The function returns `strlen`(*src*). Buffer overflow can be checked as follows:

```
if (strlcpy(dst, src, dstsize) >= dstsize)
        return –1;
```

If copying takes place between objects that overlap, the behavior of these functions is undefined.

`strcspn()`,`strspn()`

The `strcspn()` function returns the length of the initial segment of string *s1* that consists entirely of characters not from string *s2*. The `strspn()` function returns the length of the initial segment of string *s1* that consists entirely of characters from string *s2*.

`strdup()`,`strndup()`, `strdupa()`, `strndupa()`

The `strdup()` function returns a pointer to a new string that is a duplicate of the string pointed to by *s*. The returned pointer can be passed to `free()`. The space for the new string is obtained using `malloc(3C)`. If the new string cannot be created, a null pointer is returned and `errno` may be set to `ENOMEM` to indicate that the storage space available is insufficient.

The `strndup()` function is similar to `strdup()`, except that it copies at most *size* bytes. If the length of *s* is larger than *size*, only *size* bytes are copied and a terminating null byte is added. If *size* is larger than the length of *s*, all bytes in *s* are copied, including the terminating null character.

The `strdupa()` and `strndupa()` functions are similar to `strdup()` and `strndup()`, respectively, but use `alloca(3C)` to allocate the buffer.

`strlen()`,`strnlen()`

The `strlen()` function returns the number of bytes in *s*, not including the terminating null character.

The `strnlen()` function returns the smaller of *n* or the number of bytes in *s*, not including the terminating null character. The `strnlen()` function never examines more than *n* bytes of the string pointed to by *s*.

`strpbrk()`

The `strpbrk()` function 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*.

`strsep()`

The `strsep()` function locates, in the null-terminated string referenced by **stringp*, the first occurrence of any character in the string *delim* (or the terminating ‘\0' character) and replaces it with a ‘\0'. The location of the next character after the delimiter character (or `NULL`, if the end of the string was reached) is stored in **stringp*. The original value of **stringp* is returned.

An “empty" field (one caused by two adjacent delimiter characters) can be detected by comparing the location referenced by the pointer returned by `strsep()` to ‘\0'.

If **stringp* is initially `NULL`, `strsep()` returns `NULL`.

`strstr()`,`strnstr()`, `strcasestr()`

The `strstr()` function locates the first occurrence of the string *s2* (excluding the terminating null character) in string *s1* and returns a pointer to the located string, or a null pointer if the string is not found. If *s2* points to a string with zero length (that is, the string ""), the function returns *s1*.

The `strnstr()` function locates the first occurrence of the null-terminated string *s2* in the string *s1*, where not more than *n* characters are searched. Characters that appear after a ‘\0' character are not searched.

The `strcasestr()` function is similar to `strstr()`, but ignores the case of both strings.

`strtok()`

A sequence of calls to `strtok()` breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a byte from the string pointed to by *s2*. The first call in the sequence has *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *s2* can be different from call to call.

The first call in the sequence searches the string pointed to by *s1* for the first byte that is not contained in the current separator string pointed to by *s2*. If no such byte is found, then there are no tokens in the string pointed to by *s1* and `strtok()` returns a null pointer. If such a byte is found, it is the start of the first token.

The `strtok()` function then searches from there for a byte that is contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searches for a token return a null pointer. If such a byte is found, it is overwritten by a null byte that terminates the current token. The `strtok()` function saves a pointer to the following byte in thread-specific data, from which the next search for a token starts.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

See Example 1, 2, and 3 in the `EXAMPLES` section for examples of `strtok()` usage and the explanation in `NOTES`.

`strtok_r()` The `strtok_r()` function considers the null-terminated string *s1* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The argument *lasts* points to a user-provided pointer which points to stored information necessary for `strtok_r()` to continue scanning the same string.

In the first call to `strtok_r()`, *s1* points to a null-terminated string, *s2* to a null-terminated string of separator characters, and the value pointed to by *lasts* is ignored. The `strtok_r()` function returns a pointer to the first character of the first token, writes a null character into *s1* immediately following the returned token, and updates the pointer to which *lasts* points.

In subsequent calls, *s1* is a null pointer and *lasts* is unchanged from the previous call so that subsequent calls move through the string *s1*, returning successive tokens until no tokens remain. The separator string *s2* can be different from call to call. When no token remains in *s1*, a null pointer is returned.

See Example 3 in the `EXAMPLES` section for an example of `strtok_r()` usage and the explanation in `NOTES`.

Examples

EXAMPLE 1 Search for word separators.

The following example searches for tokens separated by space characters.

```
#include <string.h>
...
char *token;
char line[] = "LINE TO BE SEPARATED";
char *search = " ";

/* Token will point to "LINE". */
token = strtok(line, search);
```

EXAMPLE 1 Search for word separators. *(Continued)*

```
/* Token will point to "TO". */
token = strtok(NULL, search);
```

EXAMPLE 2 Break a Line.

The following example uses strtok to break a line into two character strings separated by any combination of SPACEs, TABs, or NEWLINEs.

```
#include <string.h>
...
struct element {
       char *key;
       char *data;
};
...
char line[LINE_MAX];
char *key, *data;
...
key = strtok(line, " \n");
data = strtok(NULL, " \n");
```

EXAMPLE 3 Search for tokens.

The following example uses both `strtok()` and `strtok_r()` to search for tokens separated by one or more characters from the string pointed to by the second argument, "/".

```
#define __EXTENSIONS__
#include <stdio.h>
#include <string.h>

int
main() {
        char *buf="5/90/45";
        char *token;
        char *lasts;

        printf("tokenizing \"%s\" with strtok():\n", buf);
        if ((token = strtok(buf, "/")) != NULL) {
                printf("token = "%s\"\n", token);
                while ((token = strtok(NULL, "/")) != NULL) {
                        printf("token = \"%s\"\n", token);
                }
        }

        buf = "//5//90//45//";
        printf("\ntokenizing \"%s\" with strtok_r():\n", buf);
        if ((token = strtok_r(buf, "/", &lasts)) != NULL) {
```

EXAMPLE 3 Search for tokens. *(Continued)*

```
                printf("token = \"%s\"\n", token);
                while ((token = strtok_r(NULL, "/", &lasts)) != NULL) {
                        printf("token = \"%s\"\n", token);
                }
        }
}
```

When compiled and run, this example produces the following output:

```
tokenizing "5/90/45" with strtok():
token = "5"
token = "90"
token = "45"

tokenizing "//5//90//45//" with strtok_r():
token = "5"
token = "90"
token = "45"
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See below.
Standard	See below.

The `strtok()` and `strdup()` functions are MT-Safe. The remaining functions are Async-Signal-Safe.

For all except `strlcat()`, `strlcpy()`, and `strsep()`, see `standards(5)`.

See Also

`alloca(3C)`, `malloc(3C)`, `setlocale(3C)`, `strxfrm(3C)`, `attributes(5)`, `standards(5)`

Notes

When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.

A single-threaded application can gain access to `strtok_r()` only by defining `__EXTENSIONS__` or by defining `_POSIX_C_SOURCE` to a value greater than or equal to 199506L.

All of these functions assume the default locale "C." For some locales, `strxfrm(3C)` should be applied to the strings before they are passed to the functions.

The `strtok()` function is safe to use in multithreaded applications because it saves its internal state in a thread-specific data area. However, its use is discouraged, even for single-threaded applications. The `strtok_r()` function should be used instead.

Do not pass the address of a character string literal as the argument *s1* to either `strtok()` or `strtok_r()`. Similarly, do not pass a pointer to the address of a character string literal as the argument *stringp* to `strsep()`. These functions can modify the storage pointed to by *s1* in the case of `strtok()` and `strtok_r()` or **stringp* in the case of `strsep()`. The C99 standard specifies that attempting to modify the storage occupied by a string literal results in undefined behavior. This allows compilers (including `gcc` and the Sun Studio compilers when the `-xstrconst` flag is used) to place string literals in read-only memory. Note that in Example 1 above, this problem is avoided because the variable *line* is declared as a writable array of type `char` that is initialized by a string literal rather than a pointer to `char` that points to a string literal.

Name string_to_decimal, file_to_decimal, func_to_decimal – parse characters into decimal record

Synopsis

```
#include <floatingpoint.h>

void string_to_decimal(char **pc, int nmax,
     int fortran_conventions, decimal_record *pd,
     enum decimal_string_form *pform, char **pechar);

void func_to_decimal(char **pc, int nmax,
     int fortran_conventions, decimal_record *pd,
     enum decimal_string_form *pform, char **pechar,
     int (*pget)(void), int *pnread, int (*punget)(int c));

#include <stdio.h>

void file_to_decimal(char **pc, int nmax,
     int fortran_conventions, decimal_record *pd,
     enum decimal_string_form *pform, char **pechar,
     FILE *pf, int *pnread);
```

Description These functions attempt to parse a numeric token from at most *nmax* characters read from a string **pc*, a file **pf*, or function (**pget*). They set the decimal record **pd* to reflect the value of the numeric token recognized and set **pform* and **pechar* to indicate its form.

The accepted forms for the numeric token consist of an initial, possibly empty, sequence of white-space characters, as defined by `isspace(3C)`, followed by a subject sequence representing a numeric value, infinity, or NaN. The subject sequence consists of an optional plus or minus sign followed by one of the following:

- a non-empty sequence of decimal digits optionally containing a decimal point character, then an optional exponent part
- one of INF or INFINITY, ignoring case
- one of NAN or NAN(*string*), ignoring case in the NAN part; *string* can be any sequence of characters not containing ')' (right parenthesis) or '\0' (null).

The *fortran_conventions* argument provides additional control over the set of accepted forms. It must be one of the following values:

0 no Fortran conventions

1 Fortran list-directed input conventions

2 Fortran formatted input conventions, blanks are ignored

3 Fortran formatted input conventions, blanks are interpreted as zeroes

When *fortran_conventions* is zero, the decimal point character is the current locale's decimal point character, and the exponent part consists of the letter `E` or `e` followed by an optional sign and a non-empty string of decimal digits.

When *fortran_conventions* is non-zero, the decimal point character is "." (period), and the exponent part consists of either a sign or one of the letters `E`, `e`, `D`, `d`, `Q`, or `q` followed by an optional sign, then a non-empty string of decimal digits.

When *fortran_conventions* is `2` or `3`, blanks can appear in the digit strings for the integer, fraction, and exponent parts, between the exponent delimiter and optional exponent sign, and after an INF, INFINITY, NAN, or NAN(*string*). When *fortran_conventions* is `2`, all blanks are ignored. When *fortran_conventions* is `3`, blanks in digit strings are interpreted as zeros and other blanks are ignored.

The following table summarizes the accepted forms and shows the corresponding values to which **pform* and *pd*->`fpclass` are set. Here *digits* represents any string of decimal digits, "." (period) stands for the decimal point character, and *exponent* represents the exponent part as defined above. Numbers in brackets refer to the notes following the table.

form	**pform*	**pd->fpclass**
all white space [1]	`whitespace_form`	`fp_zero`
digits	`fixed_int_form`	`fp_normal` [2]
digits.	`fixed_intdot_form`	`fp_normal` [2]
.digits	`fixed_dotfrac_form`	`fp_normal` [2]
digits.digits	`fixed_intdotfrac_form`	`fp_normal` [2]
digits exponent	`floating_int_form`	`fp_normal` [2]
digits. exponent	`floating_intdot_form`	`fp_normal` [2]
.digits *exponent*	`floating_dotfrac_form`	`fp_normal` [2]
digits.digits exponent	`floating_intdotfrac_form`	`fp_normal` [2]
INF	`inf_form`	`fp_infinity`
INFINITY	`infinity_form`	`fp_infinity`
NAN	`nan_form`	`fp_quiet`
NAN(*string*)	`nanstring_form`	`fp_quiet`
none of the above	`invalid_form`	`fp_signaling`

Notes:

1. The `whitespace_form` is accepted only when *fortran_conventions* is 2 or 3 and is interpreted as zero.
2. For all numeric forms, *pd*->`fpclass` is set to `fp_normal` if any non-zero digits appear in the integer or fraction parts, and otherwise *pd*->`fpclass` is set to `fp_zero`.

If the accepted token has one of the numeric forms and represents a non-zero number *x*, its significant digits are stored in *pd*->`ds`. Leading and trailing zeroes and the radix point are omitted. *pd*->`sign` and *pd*->`exponent` are set so that if *m* is the integer represented by pd->`ds`,

```
−1**(pd->sign) * m * 10**(pd->exponent)
```

approximates *x* to at least 511 significant digits. *pd*->`more` is set to 1 if this approximation is not exact (that is, the accepted token contains additional non-zero digits beyond those copied to *pd*->`ds`) and to 0 otherwise.

If the accepted token has the NAN(*string*) form, up to 511 characters from the string part are copied to *pd*->`ds`.

pd->`ds` is always terminated by a null byte, and *pd*->`ndigits` is set to the length of the string stored in *pd*->`ds`.

On entry, **pc* points to the beginning of a character string buffer. The `string_to_decimal()` function reads characters from this buffer until either enough characters are read to delimit the accepted token (for example, a null character marking the end of the string is found) or the limit of *nmax* characters is reached. The `file_to_decimal()` function reads characters from the file **pf* and stores them in the buffer. The `func_to_decimal()` function reads characters one at a time by calling the function (**pget*)() and stores them in the buffer; (**pget*)() must return integer values in the range −1 to 255, where −1 is interpreted as EOF and 0, ..., 255 are interpreted as `unsigned char` values. Both `file_to_decimal()` and `func_to_decimal()` read characters until either enough characters are read to delimit the accepted token, EOF is encountered, or the limit of *nmax* characters is reached. These functions, therefore, typically read one or more additional characters beyond the end of the accepted token and attempt to push back any excess characters read. Provided that the *punget* argument is not `NULL`, `func_to_decimal()` pushes back characters one at a time by calling (**punget*)(*c*), where *c* is an integer in the range 0 to 255 corresponding to a value previously read via (**pget*)(). After pushing back as many excess characters as possible, `file_to_decimal()` and `func_to_decimal()` store a null byte in the buffer following the last character read and not pushed back and set **pnread* to the number of characters stored in the buffer prior to this null byte. Since these functions can read up to *nmax* characters, the buffer must be large enough to hold *nmax* + 1.

On exit, **pc* points to the next character in the buffer past the last one that was accepted as part of the numeric token. If no valid token is found, **pc* is unchanged. If `file_to_decimal()` and `func_to_decimal()` successfully push back all unused characters, **pc* points to the null byte stored in the buffer following the last character read and not pushed back.

If the accepted token contains an exponent part, **pechar* is set to point to the position in the buffer where the first character of the exponent field is stored. If the accepted token does not contain an exponent part, **pechar* is set to `NULL`.

Usage If the `_IOWRT` flag is set in **pf*, `file_to_decimal()` reads characters directly from the file buffer until a null character is found. (The `_IOWRT` flag should only be set when `file_to_decimal()` is called from `sscanf(3C)`.) Otherwise, `file_to_decimal()` uses `getc_unlocked(3C)`, so it is not MT-safe unless the caller holds the stream lock.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe with exceptions

See Also `ctype(3C)`, `decimal_to_floating(3C)`, `getc_unlocked(3C)`, `isspace(3C)`, `localeconv(3C)`, `scanf(3C)`, `setlocale(3C)`, `strtod(3C)`, `ungetc(3C)`, `attributes(5)`

Name strptime – date and time conversion

Synopsis

```
#include <time.h>

char *strptime(const char *restrict buf,
     const char *restrict format, struct tm *restrict tm);
```

Non-zeroing Behavior

```
cc [flag...]  file... -D_STRPTIME_DONTZERO [library...]
#include <time.h>

char *strptime(const char *restrict buf,
     const char *restrict format, struct tm *restrict tm);
```

Description The `strptime()` function converts the character string pointed to by *buf* to values which are stored in the `tm` structure pointed to by *tm*, using the format specified by *format*.

The *format* argument is composed of zero or more conversion specifications. Each conversion specification is composed of a "%" (percent) character followed by one or two conversion characters which specify the replacement required. One or more white space characters (as specified by `isspace(3C)`) may precede or follow a conversion specification. There must be white-space or other non-alphanumeric characters between any two conversion specifications.

A non-zeroing version of `strptime()`, described below under `Non-zeroing Behavior`, is provided if `_STRPTIME_DONTZERO` is defined.

Conversion Specifications

The following conversion specifications are supported:

`%%` Same as `%`.

`%a` Day of week, using the locale's weekday names; either the abbreviated or full name may be specified.

`%A` Same as `%a`.

`%b` Month, using the locale's month names; either the abbreviated or full name may be specified.

`%B` Same as `%b`.

`%c` Locale's appropriate date and time representation.

`%C` Century number (the year divided by 100 and truncated to an integer as a decimal number [1,99]); single digits are preceded by 0. If `%C` is used without the `%y` specifier, `strptime()` assumes the year offset is zero in whichever century is specified. Note the behavior of `%C` in the absence of `%y` is not specified by any of the standards or specifications described on the `standards(5)` manual page, so portable applications should not depend on it. This behavior may change in a future release.

`%d` Day of month [1,31]; leading zero is permitted but not required.

`%D` Date as `%m/%d/%y`.

`%e` Same as `%d`.

`%h` Same as `%b`.

`%H` Hour (24-hour clock) [0,23]; leading zero is permitted but not required.

`%I` Hour (12-hour clock) [1,12]; leading zero is permitted but not required.

`%j` Day number of the year [1,366]; leading zeros are permitted but not required.

`%m` Month number [1,12]; leading zero is permitted but not required.

`%M` Minute [0-59]; leading zero is permitted but not required.

`%n` Any white space.

`%p` Locale's equivalent of either a.m. or p.m.

`%r` Appropriate time representation in the 12-hour clock format with `%p`.

`%R` Time as `%H:%M`.

SUSv3

`%S` Seconds [0,60]; leading zero is permitted but not required. The range of values is [00,60] rather than [00,59] to allow for the occasional leap second.

Default and other standards

`%S` Seconds [0,61]; leading zero is permitted but not required. The range of values is [00,61] rather than [00,59] to allow for the occasional leap second and even more occasional double leap second.

`%t` Any white space.

`%T` Time as `%H:%M:%S`.

`%U` Week number of the year as a decimal number [0,53], with Sunday as the first day of the week; leading zero is permitted but not required.

`%w` Weekday as a decimal number [0,6], with 0 representing Sunday.

`%W` Week number of the year as a decimal number [0,53], with Monday as the first day of the week; leading zero is permitted but not required.

`%x` Locale's appropriate date representation.

`%X` Locale's appropriate time representation.

`%y` Year within century. When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969 to 1999 inclusive); values in the range 00-68 refer to years in the twenty-first century (2000 to 2068 inclusive).

`%Y` Year, including the century (for example, 1993).

`%Z` Time zone name or no characters if no time zone exists.

Modified Conversion Specifications

Some conversion specifications can be modified by the `E` and `O` modifier characters to indicate that an alternate format or specification should be used rather than the one normally used by the unmodified specification. If the alternate format or specification does not exist in the current locale, the behavior will be as if the unmodified conversion specification were used.

`%Ec` Locale's alternate appropriate date and time representation.

`%EC` Name of the base year (era) in the locale's alternate representation.

`%Ex` Locale's alternate date representation.

`%EX` Locale's alternate time representation.

`%Ey` Offset from `%EC` (year only) in the locale's alternate representation.

`%EY` Full alternate year representation.

`%Od` Day of the month using the locale's alternate numeric symbols.

`%Oe` Same as `%Od`.

`%OH` Hour (24-hour clock) using the locale's alternate numeric symbols.

`%OI` Hour (12-hour clock) using the locale's alternate numeric symbols.

`%Om` Month using the locale's alternate numeric symbols.

`%OM` Minutes using the locale's alternate numeric symbols.

`%OS` Seconds using the locale's alternate numeric symbols.

`%OU` Week number of the year (Sunday as the first day of the week) using the locale's alternate numeric symbols.

`%Ow` Number of the weekday (Sunday=0) using the locale's alternate numeric symbols.

`%OW` Week number of the year (Monday as the first day of the week) using the locale's alternate numeric symbols.

`%Oy` Year (offset from `%C`) in the locale's alternate representation and using the locale's alternate numeric symbols.

General Specifications

A conversion specification that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the specification, the specification fails, and the differing and subsequent characters remain unscanned.

A series of specifications composed of `%n`, `%t`, white-space characters or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned. White space is defined by `isspace(3C)`.

Any other conversion specification is executed by scanning characters until a character matching the next specification is scanned, or until no more characters can be scanned. These characters, except the one matching the next specification, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate *tm* structure members are set to values corresponding to the locale information. If no match is found, `strptime()` fails and no more characters are scanned.

The month names, weekday names, era names, and alternate numeric symbols can consist of any combination of upper and lower case letters. The user can request that the input date or time specification be in a specific language by setting the `LC_TIME` category using `setlocale(3C)`.

Non-zeroing Behavior

In addition to the behavior described above by various standards, the Solaris implementation of `strptime()` provides the following extensions. These may change at any time in the future. Portable applications should not depend on these extended features:

- If `_STRPTIME_DONTZERO` is not defined, the `tm struct` is zeroed on entry and `strptime()` updates the fields of the `tm struct` associated with the specifiers in the format string.
- If `_STRPTIME_DONTZERO` is defined, `strptime()` does not zero the `tm struct` on entry. Additionally, for some specifiers, `strptime()` will use some values in the input `tm struct` to recalculate the date and re-assign the appropriate members of the `tm struct`.

The following describes extended features regardless of whether `_STRPTIME_DONTZERO` is defined or not defined:

- If `%j` is specified, `tm_yday` is set; if year is given, and if month and day are not given, `strptime()` calculates and sets `tm_mon`, `tm_mday`, and `tm_year`.
- If `%U` or `%W` is specified and if weekday and year are given and month and day of month are not given, `strptime()` calculates and sets `tm_mon`, `tm_mday`, `tm_wday`, and `tm_year`.

The following describes extended features when `_STRPTIME_DONTZERO` is not defined:

- If `%C` is specified and `%y` is not specified, `strptime()` assumes 0 as the year offset, then calculates the year, and assigns `tm_year`.

The following describes extended features when `_STRPTIME_DONTZERO` is defined:

- If `%C` is specified and `%y` is not specified, `strptime()` assumes the year offset of the year value of the `tm_year` member of the input `tm struct`, then calculates the year and assigns `tm_year`.
- If `%j` is specified and neither `%y`, `%Y`, nor `%C` are specified, and neither month nor day of month are specified, `strptime()` assumes the year value given by the value of the `tm_year` field of the input `tm struct`. Then, in addition to setting `tm_yday`, `strptime()` uses day-of-year and year values to calculate the month and day-of-month, and assigns `tm_month` and `tm_mday`.

- If `%U` or `%W` is specified, and if weekday and/or year are not given, and month and day of month are not given, `strptime()` will assume the weekday value and/or the year value as the value of the `tm_wday` field and/or `tm_year` field of the input `tm struct`. Then, `strptime()` will calculate the month and day-of-month and assign `tm_month`, `tm_mday`, and/or `tm_year`.
- If `%p` is specified and if hour is not specified, `strptime()` will reference, and if needed, update the `tm_hour` member. If the `am_pm` input is p.m. and the input `tm_hour` value is between 0 - 11, `strptime()` will add 12 hours and update `tm_hour`. If the `am_pm` input is a.m. and input `tm_hour` value is between 12 - 23, `strptime()` will subtract 12 hours and update `tm_hour`.

Return Values Upon successful completion, `strptime()` returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

Usage Several "same as" formats, and the special processing of white-space characters are provided in order to ease the use of identical *format* strings for `strftime(3C)` and `strptime()`.

The `strptime()` function tries to calculate `tm_year`, `tm_mon`, and `tm_mday` when given incomplete input. This allows the `struct tm` created by `strptime()` to be passed to `mktime(3C)` to produce a `time_t` value for dates and times that are representable by a `time_t`. As an example, since `mktime()` ignores `tm_yday`, `strptime()` calculates `tm_mon` and `tm_mday` as well as filling in `tm_yday` when `%j` is specified without otherwise specifying a month and day within month.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `ctime(3C)`, `getdate(3C)`, `isspace(3C)`, `mktime(3C)`, `setlocale(3C)`, `strftime(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Name strsignal – get name of signal

Synopsis

```
#include <string.h>

char *strsignal(int sig);
```

Description The `strsignal()` function maps the signal number in *sig* to a string describing the signal and returns a pointer to that string. It uses the same set of the messages as `psignal(3C)`. The returned string should not be overwritten.

Return Values The `strsignal()` function returns `NULL` if *sig* is not a valid signal number.

Usage Messages returned from this function are in the native language specified by the `LC_MESSAGES` locale category. See `setlocale(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `gettext(3C)`, `psignal(3C)`, `setlocale(3C)`, `str2sig(3C)`, `attributes(5)`

Name strtod, strtof, strtold, atof – convert string to floating-point number

Synopsis

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr);

float strtof(const char *restrict nptr, char **restrict endptr);

long double strtold(const char *restrict nptr, char **restrict endptr);

double atof(const char *str);
```

Description The `strtod()`, `strtof()`, and `strtold()` functions convert the initial portion of the string pointed to by *nptr* to `double`, `float`, and `long double` representation, respectively. First they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by `isspace(3C)`)
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

Then they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of digits optionally containing a radix character, then an optional exponent part
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- One of INF or INFINITY, ignoring case
- One of NAN or NAN(*n-char-sequence*$_{opt}$), ignoring case in the NAN part, where:

```
n-char-sequence:
    digit
    nondigit
    n-char-sequence digit
    n-char-sequence nondigit
```

In default mode for `strtod()`, only decimal, INF/INFINITY, and NAN/NAN(*n-char-sequence*) forms are recognized. In C99/SUSv3 mode, hexadecimal strings are also recognized.

In default mode for `strtod()`, the *n-char-sequence* in the NAN(*n-char-equence*) form can contain any character except ')' (right parenthesis) or '\0' (null). In C99/SUSv3 mode, the *n-char-sequence* can contain only upper and lower case letters, digits, and '_' (underscore).

The `strtof()` and `strtold()` functions always function in C99/SUSv3-conformant mode.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant of the C language, except that the radix character is used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A character sequence INF or INFINITY is interpreted as an infinity. A character sequence NAN or NAN($n\text{-}char\text{-}sequence_{opt}$) is interpreted as a quiet NaN. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has either the decimal or hexadecimal form, the value resulting from the conversion is rounded correctly according to the prevailing floating point rounding direction mode. The conversion also raises floating point inexact, underflow, or overflow exceptions as appropriate.

The radix character is defined in the program's locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period ('.').

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The `strtod()` function does not change the setting of `errno` if successful.

The `atof(`*str*`)` function call is equivalent to `strtod(`*nptr*`, (char **)NULL)`.

Return Values Upon successful completion, these functions return the converted value. If no conversion could be performed, `0` is returned.

If the correct value is outside the range of representable values, `±HUGE_VAL`, `±HUGE_VALF`, or `±HUGE_VALL` is returned (according to the sign of the value), a floating point overflow exception is raised, and `errno` is set to `ERANGE`.

If the correct value would cause an underflow, the correctly rounded result (which may be normal, subnormal, or zero) is returned, a floating point underflow exception is raised, and `errno` is set to `ERANGE`.

Errors These functions will fail if:

`ERANGE` The value to be returned would cause overflow or underflow

These functions may fail if:

`EINVAL` No conversion could be performed.

Usage Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set `errno` to 0, then call `strtod()`, `strtof()`, or `strtold()`, then check `errno`.

The changes to `strtod()` introduced by the ISO/IEC 9899: 1999 standard can alter the behavior of well-formed applications complying with the ISO/IEC 9899: 1990 standard and thus earlier versions of IEEE Std 1003.1-200x. One such example would be:

```
int
what_kind_of_number (char *s)
{
     char *endp;
     double d;
     long l;
     d = strtod(s, &endp);
     if (s != endp && *endp == '\\0')
         printf("It's a float with value %g\n", d);
     else
     {
         l = strtol(s, &endp, 0);
         if (s != endp && *endp == '\0')
             printf("It's an integer with value %ld\n", 1);
         else
             return 1;
     }
     return 0;
}
```

If the function is called with:

```
what_kind_of_number ("0x10")
```

an ISO/IEC 9899: 1990 standard-compliant library will result in the function printing:

```
It's an integer with value 16
```

With the ISO/IEC 9899: 1999 standard, the result is:

```
It's a float with value 16
```

The change in behavior is due to the inclusion of floating-point numbers in hexadecimal notation without requiring that either a decimal point or the binary exponent be present.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See standards(5).

See Also isspace(3C), localeconv(3C), scanf(3C), setlocale(3C), strtol(3C), attributes(5), standards(5)

Notes The `strtod()` and `atof()` functions can be used safely in multithreaded applications, as long as setlocale(3C) is not called to change the locale.

The DESCRIPTION and RETURN VALUES sections above are very similar to the wording used by the Single UNIX Specification version 2 (SUSv2) and the 1989 C Standard to describe the behavior of the `strtod()` function. Since some users have reported that they find the description confusing, the following notes might be helpful.

1. The `strtod()` function does not modify the string pointed to by *str* and does not `malloc()` space to hold the decomposed portions of the input string.
2. If *endptr* is not `(char **)NULL`, `strtod()` will set the pointer pointed to by *endptr* to the first byte of the "final string of unrecognized characters". (If all input characters were processed, the pointer pointed to by *endptr* will be set to point to the null character at the end of the input string.)
3. If `strtod()` returns 0.0, one of the following occurred:
   a. The "subject sequence" was not an empty string, but evaluated to 0.0. (In this case, `errno` will be left unchanged.)
   b. The "subject sequence" was an empty string . In this case, `errno` will be left unchanged. (The Single UNIX Specification version 2 allows `errno` to be set to `EINVAL` or to be left unchanged. The C Standard does not specify any specific behavior in this case.)
   c. The "subject sequence" specified a numeric value whose conversion resulted in a floating point underflow. In this case, an underflow exception is raised and `errno` is set to `ERANGE`.

   Note that the standards do not require that implementations distinguish between these three cases. An application can determine case (b) by making sure that there are no leading white-space characters in the string pointed to by *str* and giving `strtod()` an *endptr* that is not `(char **)NULL`. If *endptr* points to the first character of *str* when `strtod()` returns, you have detected case (b). Case (c) can be detected by examining the underflow flag or by looking for a non-zero digit before the exponent part of the "subject sequence". Note, however, that the decimal-point character is locale-dependent.

4. If `strtod()` returns `+HUGE_VAL` or `–HUGE_VAL`, one of the following occurred:
   a. If `+HUGE_VAL` is returned and `errno` is set to `ERANGE`, a floating point overflow occurred while processing a positive value, causing a floating point overflow exception to be raised.
   b. If `–HUGE_VAL` is returned and `errno` is set to `ERANGE`, a floating point overflow occurred while processing a negative value, causing a floating point overflow exception to be raised.
   c. If `strtod()` does not set `errno` to `ERANGE`, the value specified by the "subject string" converted to `+HUGE_VAL` or `–HUGE_VAL`, respectively.

   Note that if `errno` is set to `ERANGE` when `strtod()` is called, case (c) can be distinguished from cases (a) and (b) by examining either `ERANGE` or the overflow flag.

Name strtoimax, strtoumax – convert string to integer type

Synopsis

```
#include <inttypes.h>

intmax_t strtoimax(const char *restrict nptr,
     char **restrict endptr, int base);

uintmax_t strtoumax(const char *restrict nptr,
     char **restrict endptr, int base);
```

Description These functions are equivalent to the `strtol()`, `strtoll()`, `strtoul()`, and `strtoull()` functions, except that the initial portion of the string is converted to `intmax_t` and `uintmax_t` representation, respectively.

Return Values These functions return the converted value, if any.

If no conversion could be performed, 0 is returned.

If the correct value is outside the range of representable values, {`INTMAX_MAX`}, {`INTMAX_MIN`}, or {`UINTMAX_MAX`} is returned (according to the return type and sign of the value, if any), and `errno` is set to `ERANGE`.

Errors These functions will fail if:

`ERANGE` The value to be returned is not representable.

These functions may fail if:

`EINVAL` The value of *base* is not supported.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `strtol(3C)`, `strtoul(3C)`, `attributes(5)`, `standards(5)`

Name strtol, strtoll, atol, atoll, atoi, lltostr, ulltostr – string conversion routines

Synopsis

```
#include <stdlib.h>

long strtol(const char *restrict str, char **restrict endptr, int base);

long long strtoll(const char *restrict str, char **restrict endptr,
     int base);

long atol(const char *str);

long long atoll(const char *str);

int atoi(const char *str);

char *lltostr(long long value, char *endptr);

char *ulltostr(unsigned long long value, char *endptr);
```

Description

strtol() and strtoll()

The `strtol()` function converts the initial portion of the string pointed to by *str* to a type `long int` representation.

The `strtoll()` function converts the initial portion of the string pointed to by *str* to a type `long long` representation.

Both functions first decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by `isspace(3C)`); a subject sequence interpreted as an integer represented in some radix determined by the value of *base*; and a final string of one or more unrecognized characters, including the terminating null byte of the input string. They then attempt to convert the subject sequence to an integer and return the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a + or – sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a + or – sign. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence

contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the POSIX locale, additional implementation-dependent subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

`atol()`, `atoll()` and `atoi()`

Except for behavior on error, `atol()` is equivalent to: `strtol(str, (char **)NULL, 10)`.

Except for behavior on error, `atoll()` is equivalent to: `strtoll(str, (char **)NULL, 10)`.

Except for behavior on error, `atoi()` is equivalent to: `(int) strtol(str, (char **)NULL, 10)`.

If the value cannot be represented, the behavior is undefined.

`lltostr()` and `ulltostr()`

The `lltostr()` function returns a pointer to the string represented by the `long long` *value*. The *endptr* argument is assumed to point to the byte following a storage area into which the decimal representation of *value* is to be placed as a string. The `lltostr()` function converts *value* to decimal and produces the string, and returns a pointer to the beginning of the string. No leading zeros are produced, and no terminating null is produced. The low-order digit of the result always occupies memory position *endptr*–1. The behavior of `lltostr()` is undefined if *value* is negative. A single zero digit is produced if *value* is 0.

The `ulltostr()` function is similar to `lltostr()` except that *value* is an `unsigned long long`.

Return Values

Upon successful completion, `strtol()`, `strtoll()`, `atol()`, `atoll()`, and `atoi()` return the converted value, if any. If no conversion could be performed, `strtol()` and `strtoll()` return `0` and `errno` may be set to `EINVAL`.

If the correct value is outside the range of representable values, `strtol()` returns `LONG_MAX` or `LONG_MIN` and `strtoll()` returns `LLONG_MAX` or `LLONG_MIN` (according to the sign of the value), and `errno` is set to `ERANGE`.

Upon successful completion, `lltostr()` and `ulltostr()` return a pointer to the converted string.

Errors The `strtol()` and `strtoll()` functions will fail if:

`ERANGE` The value to be returned is not representable.

The `strtol()` and `strtoll()` functions may fail if:

`EINVAL` The value of *base* is not supported.

Usage Because 0, `LONG_MIN`, `LONG_MAX`, `LLONG_MIN`, and `LLONG_MAX` are returned on error and are also valid returns on success, an application wishing to check for error situations should set `errno` to 0, call the function, then check `errno` and if it is non-zero, assume an error has occurred.

The `strtol()` function no longer accepts values greater than `LONG_MAX` or `LLONG_MAX` as valid input. Use `strtoul(3C)` instead.

Calls to `atoi()` and `atol()` might be faster than corresponding calls to `strtol()`, and calls to `atoll()` might be faster than corresponding calls to `strtoll()`. However, applications should not use the `atoi()`, `atol()`, or `atoll()` functions unless they know the value represented by the argument will be in range for the corresponding result type.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See below.

For `strtol()`, `strtoll()`, `atol()`, `atoll()`, and `atoi()`, see `standards(5)`.

See Also `isalpha(3C)`, `isspace(3C)`, `scanf(3C)`, `strtod(3C)`, `strtoul(3C)`, `attributes(5)`, `standards(5)`

Name strtoul, strtoull – convert string to unsigned long

Synopsis

```
#include <stdlib.h>

unsigned long strtoul(const char *restrict str,
     char **restrict endptr, int base);

unsigned long long strtoull(const char *restrict str,
     char **restrict endptr, int base);
```

Description The `strtoul()` function converts the initial portion of the string pointed to by *str* to a type `unsigned long int` representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by `isspace(3C)`); a subject sequence interpreted as an integer represented in some radix determined by the value of *base*; and a final string of one or more unrecognised characters, including the terminating null byte of the input string. Then it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a + or – sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a + or – sign. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the POSIX locale, additional implementation-dependent subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The `strtoull()` function is identical to `strtoul()` except that it returns the value represented by *str* as an `unsigned long long`.

Return Values Upon successful completion `strtoul()` returns the converted value, if any. If no conversion could be performed, `0` is returned and `errno` may be set to `EINVAL`. If the correct value is outside the range of representable values, `ULONG_MAX` is returned and `errno` is set to `ERANGE`.

Errors The `strtoul()` function will fail if:

`EINVAL` The value of *base* is not supported.

`ERANGE` The value to be returned is not representable.

The `strtoul()` function may fail if:

`EINVAL` No conversion could be performed.

Usage Because 0 and `ULONG_MAX` are returned on error and are also valid returns on success, an application wishing to check for error situations should set `errno` to 0, then call `strtoul()`, then check `errno` and if it is non-zero, assume an error has occurred.

Unlike `strtod(3C)` and `strtol(3C)`, `strtoul()` must always return a non-negative number; so, using the return value of `strtoul()` for out-of-range numbers with `strtoul()` could cause more severe problems than just loss of precision if those numbers can ever be negative.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `isalpha(3C)`, `isspace(3C)`, `scanf(3C)`, `strtod(3C)`, `strtol(3C)`, `attributes(5)`, `standards(5)`

Name strtows, wstostr – code conversion for Process Code and File Code

Synopsis

```
#include <widec.h>

wchar_t *strtows(wchar_t *dst, char *src);

char *wstostr(char *dst, wchar_t *src);
```

Description The `strtows()` and `wstostr()` functions convert strings back and forth between File Code representation and Process Code.

The `strtows()` function takes a character string *src*, converts it to a Process Code string, terminated by a Process Code null, and places the result into *dst*.

The `wstostr()` function takes the Process Code string pointed to by *src*, converts it to a character string, and places the result into *dst*.

Return Values The `strtows()` function returns the Process Code string if it completes successfully. Otherwise, a null pointer will be returned and `errno` will be set to `EILSEQ`.

The `wstostr()` function returns the File Code string if it completes successfully. Otherwise, a null pointer will be returned and `errno` will be set to `EILSEQ`.

See Also `wstring(3C)`

Name strxfrm – string transformation

Synopsis

```
#include <string.h>

size_t strxfrm(char *restrict s1, const char *restrict 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(3C)` is applied to two transformed strings, it returns a value greater than, equal to or less than `0`, corresponding to the result of `strcoll(3C)` applied to the same two original strings. No more than *n* bytes are placed into the resulting array pointed to by *s1*, including the terminating null byte. If *n* is `0`, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

The `strxfrm()` function does not change the setting of `errno` if successful.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to 0, then call `strxfrm()`, then check `errno`.

Return Values Upon successful completion, `strxfrm()` returns the length of the transformed string (not including the terminating null byte). If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate.

On error, `strxfrm()` may set `errno` but no return value is reserved to indicate the error.

Usage The transformation function is such that two transformed strings can be ordered by `strcmp(3C)` as appropriate to collating sequence information in the program's locale (category `LC_COLLATE`).

The fact that when *n* is `0`, *s1* is permitted to be a null pointer, is useful to determine the size of the *s1* array prior to making the transformation.

Examples **EXAMPLE 1** A sample of using the `strxfm()` function.

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);
```

Files `/usr/lib/locale/`*locale*`/`*locale*`.so.*` `LC_COLLATE` database for *locale*

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

The strxfrm() function can be used safely in a multithreaded application, as long as setlocale(3C) is not being called to change the locale.

See Also localedef(1), setlocale(3C), strcmp(3C), strcoll(3C), wscoll(3C), attributes(5), environ(5), standards(5)

Name swab – swap bytes

Synopsis

```
#include <stdlib.h>

void swab(const char *src, char *dest, ssize_t nbytes);
```

XPG4, SUS, SUSv2, SUSv3

```
#include <unistd.h>

void swab(const void *restrict src, void *restrict dest, ssize_t nbytes);
```

Description The `swab()` function copies *nbytes* bytes, which are pointed to by *src*, to the object pointed to by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even. If *nbytes* is odd `swab()` copies and exchanges *nbytes*–1 bytes and the disposition of the last byte is unspecified. If copying takes place between objects that overlap, the behavior is undefined. If *nbytes* is negative, `swab()` does nothing.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `attributes(5)`, `standards(5)`

Name sync_instruction_memory – make modified instructions executable

Synopsis

```
void sync_instruction_memory(caddr_t addr, int len);
```

Description The `sync_instruction_memory()` function performs whatever steps are required to make instructions modified by a program executable.

Some processor architectures, including some SPARC processors, have separate and independent instruction and data caches which are not kept consistent by hardware. For example, if the instruction cache contains an instruction from some address and the program then stores a new instruction at that address, the new instruction may not be immediately visible to the instruction fetch mechanism. Software must explicitly invalidate the instruction cache entries for new or changed mappings of pages that might contain executable instructions. The `sync_instruction_memory()` function performs this function, and/or any other functions needed to make modified instructions between *addr* and *addr+len* visible. A program should call `sync_instruction_memory()` after modifying instructions and before executing them.

On processors with unified caches (one cache for both instructions and data) and pipelines which are flushed by a branch instruction, such as the x86 architecture, the function may do nothing and just return.

The changes are immediately visible to the thread calling `sync_instruction_memory()` when the call returns, even if the thread should migrate to another processor during or after the call. The changes become visible to other threads in the same manner that stores do; that is, they eventually become visible, but the latency is implementation-dependent.

The result of executing `sync_instruction_memory()` are unpredictable if *addr* through *addr+len*-1 are not valid for the address space of the program making the call.

Return Values No values are returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `attributes(5)`

Name sysconf – get configurable system variables

Synopsis

```
#include <unistd.h>

long sysconf(int name);
```

Description The `sysconf()` function provides a method for an application to determine the current value of a configurable system limit or option (variable).

The *name* argument represents the system variable to be queried. The following table lists the minimal set of system variables from `<limits.h>` and `<unistd.h>` that can be returned by `sysconf()` and the symbolic constants defined in `<unistd.h>` that are the corresponding values used for *name* on the SPARC and x86 platforms.

Name	Return Value	Meaning
_SC_2_C_BIND	_POSIX2_C_BIND	Supports the C language binding option
_SC_2_C_DEV	_POSIX2_C_DEV	Supports the C language development utilities option
_SC_2_C_VERSION	_POSIX2_C_VERSION	Integer value indicates version of ISO POSIX-2 standard (Commands)
_SC_2_CHAR_TERM	_POSIX2_CHAR_TERM	Supports at least one terminal
_SC_2_FORT_DEV	_POSIX2_FORT_DEV	Supports FORTRAN Development Utilities Option
_SC_2_FORT_RUN	_POSIX2_FORT_RUN	Supports FORTRAN Run-time Utilities Option
_SC_2_LOCALEDEF	_POSIX2_LOCALEDEF	Supports creation of locales by the localedef utility
_SC_2_SW_DEV	_POSIX2_SW_DEV	Supports Software Development Utility Option
_SC_2_UPE	_POSIX2_UPE	Supports User Portability Utilities Option
_SC_2_VERSION	_POSIX2_VERSION	Integer value indicates version of ISO POSIX-2 standard (C language binding)
_SC_AIO_LISTIO_MAX	AIO_LISTIO_MAX	Max number of I/O operations in a

		single list I/O call supported
_SC_AIO_MAX	AIO_MAX	Max number of outstanding asynchronous I/O operations supported
_SC_AIO_PRIO_DELTA_MAX	AIO_PRIO_DELTA_MAX	Max amount by which process can decrease its asynchronous I/O priority level from its own scheduling priority
_SC_ARG_MAX	ARG_MAX	Max size of argv[] plus envp[]
_SC_ASYNCHRONOUS_IO	_POSIX_ASYNCHRONOUS_IO	Supports Asynchronous I/O
_SC_ATEXIT_MAX	ATEXIT_MAX	Max number of functions that can be registered with atexit()
_SC_AVPHYS_PAGES		Number of physical memory pages not currently in use by system
_SC_BARRIERS	_POSIX_BARRIERS	Supports Barriers option
_SC_BC_BASE_MAX	BC_BASE_MAX	Maximum obase values allowed by bc
_SC_BC_DIM_MAX	BC_DIM_MAX	Max number of elements permitted in array by bc
_SC_BC_SCALE_MAX	BC_SCALE_MAX	Max scale value allowed by bc
_SC_BC_STRING_MAX	BC_STRING_MAX	Max length of string constant allowed by bc
_SC_CHILD_MAX	CHILD_MAX	Max processes allowed to a UID
_SC_CLK_TCK	CLK_TCK	Ticks per second (clock_t)
_SC_CLOCK_SELECTION	_POSIX_CLOCK_SELECTION	Supports Clock Selection option
_SC_COLL_WEIGHTS_MAX	COLL_WEIGHTS_MAX	Max number of weights that can be assigned to entry of the LC_COLLATE order keyword in locale definition file

_SC_CPUID_MAX		Max possible processor ID
_SC_DELAYTIMER_MAX	DELAYTIMER_MAX	Max number of timer expiration overruns
_SC_EXPR_NEST_MAX	EXPR_NEST_MAX	Max number of parentheses by expr
_SC_FSYNC	_POSIX_FSYNC	Supports File Synchronization
_SC_GETGR_R_SIZE_MAX		Max size of group entry buffer
_SC_GETPW_R_SIZE_MAX		Max size of password entry buffer
_SC_HOST_NAME_MAX	_POSIX_HOST_NAME_MAX	Maximum length of a host name (excluding terminating null)
_SC_IOV_MAX	IOV_MAX	Max number of iovec structures available to one process for use with readv() and writev()
_SC_JOB_CONTROL	_POSIX_JOB_CONTROL	Job control supported?
_SC_LINE_MAX	LINE_MAX	Max length of input line
_SC_LOGIN_NAME_MAX	LOGNAME_MAX + 1	Max length of login name
_SC_LOGNAME_MAX	LOGNAME_MAX	
_SC_MAPPED_FILES	_POSIX_MAPPED_FILES	Supports Memory Mapped Files
_SC_MAXPID		Max pid value
_SC_MEMLOCK	_POSIX_MEMLOCK	Supports Process Memory Locking
_SC_MEMLOCK_RANGE	_POSIX_MEMLOCK_RANGE	Supports Range Memory Locking
_SC_MEMORY_PROTECTION	_POSIX_MEMORY_PROTECTION	Supports Memory Protection
_SC_MESSAGE_PASSING	_POSIX_MESSAGE_PASSING	Supports Message Passing
_SC_MONOTONIC_CLOCK	_POSIX_MONOTONIC_CLOCK	Supports Monotonic Clock option
_SC_MQ_OPEN_MAX	MQ_OPEN_MAX	Max number of open message queues a process can hold
_SC_MQ_PRIO_MAX	MQ_PRIO_MAX	Max number of message priorities supported
_SC_NGROUPS_MAX	NGROUPS_MAX	Max simultaneous groups to which

		one can belong
_SC_NPROCESSORS_CONF		Number of processors configured
_SC_NPROCESSORS_MAX		Max number of processors supported by platform
_SC_NPROCESSORS_ONLN		Number of processors online
_SC_OPEN_MAX	OPEN_MAX	Max open files per process
_SC_PAGESIZE	PAGESIZE	System memory page size
_SC_PAGE_SIZE	PAGESIZE	Same as _SC_PAGESIZE
_SC_PASS_MAX	PASS_MAX	Max number of significant bytes in a password
_SC_PHYS_PAGES		Total number of pages of physical memory in system
_SC_PRIORITIZED_IO	_POSIX_PRIORITIZED_IO	Supports Prioritized I/O
_SC_PRIORITY_SCHEDULING	_POSIX_PRIORITY_SCHEDULING	Supports Process Scheduling
_SC_RAW_SOCKETS	_POSIX_RAW_SOCKETS	Supports Raw Sockets option
_SC_RE_DUP_MAX	RE_DUP_MAX	Max number of repeated occurrences of a regular expression permitted when using interval notation \{m,n\}
_SC_READER_WRITER_LOCKS	_POSIX_READER_WRITER_LOCKS	Supports IPV6 option
_SC_REALTIME_SIGNALS	_POSIX_REALTIME_SIGNALS	Supports Realtime Signals
_SC_REGEXP	_POSIX_REGEXP	Supports Regular Expression Handling option
_SC_RTSIG_MAX	RTSIG_MAX	Max number of realtime signals reserved for application use
_SC_SAVED_IDS	_POSIX_SAVED_IDS	Saved IDs (seteuid()) supported?
_SC_SEM_NSEMS_MAX	SEM_NSEMS_MAX	Max number of POSIX semaphores a process can have
_SC_SEM_VALUE_MAX	SEM_VALUE_MAX	Max value a POSIX

		semaphore can have
_SC_SEMAPHORES	_POSIX_SEMAPHORES	Supports Semaphores
_SC_SHARED_MEMORY_ OBJECTS	_POSIX_SHARED_MEMORY_ OBJECTS	Supports Shared Memory Objects
_SC_SHELL	_POSIX_SHELL	Supports POSIX shell
_SC_SIGQUEUE_MAX	SIGQUEUE_MAX	Max number of queued signals that a process can send and have pending at receiver(s) at a time
_SC_SPAWN	_POSIX_SPAWN	Supports Spawn option
_SC_SPIN_LOCKS	_POSIX_SPIN_LOCKS	Supports Spin Locks option
_SC_STACK_PROT		Default stack protection
_SC_STREAM_MAX	STREAM_MAX	Number of streams one process can have open at a time
_SC_SYMLOOP_MAX	_POSIX_SYMLOOP_MAX	Max number of symbolic links that can be reliably traversed in the resolution of a pathname in the absence of a loop
_SC_SYNCHRONIZED_IO	_POSIX_SYNCHRONIZED_IO	Supports Synchronized I/O
_SC_THREAD_ATTR_ STACKADDR	_POSIX_THREAD_ATTR_ STACKADDR	Supports Thread Stack Address Attribute option
_SC_THREAD_ATTR_ STACKSIZE	_POSIX_THREAD_ATTR_ STACKSIZE	Supports Thread Stack Size Attribute option
_SC_THREAD_DESTRUCTOR_ ITERATIONS	PTHREAD_DESTRUCTOR_ ITERATIONS	Number attempts made to destroy thread-specific data on thread exit
_SC_THREAD_KEYS_MAX	PTHREAD_KEYS_MAX	Max number of data keys per process
_SC_THREAD_PRIO_ INHERIT	_POSIX_THREAD_PRIO_ INHERIT	Supports Priority Inheritance option
_SC_THREAD_PRIO_ PROTECT	_POSIX_THREAD_PRIO_ PROTECT	Supports Priority Protection option
_SC_THREAD_PRIORITY_ SCHEDULING	_POSIX_THREAD_PRIORITY_ SCHEDULING	Supports Thread Execution Scheduling option
_SC_THREAD_PROCESS_ SHARED	_POSIX_THREAD_PROCESS_ SHARED	Supports Process-Shared

		Synchronization option
_SC_THREAD_SAFE_FUNCTIONS	_POSIX_THREAD_SAFE_FUNCTIONS	Supports Thread-Safe Functions option
_SC_THREAD_STACK_MIN	PTHREAD_STACK_MIN	Min byte size of thread stack storage
_SC_THREAD_THREADS_MAX	PTHREAD_THREADS_MAX	Max number of threads per process
_SC_THREADS	_POSIX_THREADS	Supports Threads option
_SC_TIMEOUTS	_POSIX_TIMEOUTS	Supports Timeouts option
_SC_TIMER_MAX	TIMER_MAX	Max number of timer per process supported
_SC_TIMERS	_POSIX_TIMERS	Supports Timers
_SC_TTY_NAME_MAX	TTYNAME_MAX	Max length of tty device name
_SC_TZNAME_MAX	TZNAME_MAX	Max number of bytes supported for name of a time zone
_SC_V6_ILP32_OFF32	_POSIX_V6_ILP32_OFF32	Supports X/Open ILP32 w/32-bit offset build environment
_SC_V6_ILP32_OFFBIG	_POSIX_V6_ILP32_OFFBIG	Supports X/Open ILP32 w/64-bit offset build environment
_SC_V6_LP64_OFF64	_POSIX_V6_LP64_OFF64	Supports X/Open LP64 w/64-bit offset build environment
_SC_V6_LPBIG_OFFBIG	_POSIX_V6_LPBIG_OFFBIG	Same as _SC_V6_LP64_OFF64
_SC_VERSION	_POSIX_VERSION	POSIX.1 version supported
_SC_XBS5_ILP32_OFF32	_XBS_ILP32_OFF32	Indicates support for X/Open ILP32 w/32-bit offset build environment
_SC_XBS5_ILP32_OFFBIG	_XBS5_ILP32_OFFBIG	Indicates support for X/Open ILP32 w/64-bit offset build environment
_SC_XBS5_LP64_OFF64	_XBS5_LP64_OFF64	Indicates support of X/Open LP64, 64-bit offset

		build environment
_SC_XBS5_LPBIG_OFFBIG	_XBS5_LP64_OFF64	Same as _SC_XBS5_LP64_OFF64
_SC_XOPEN_CRYPT	_XOPEN_CRYPT	Supports X/Open Encryption Feature Group
_SC_XOPEN_ENH_I18N	_XOPEN_ENH_I18N	Supports X/Open Enhanced Internationalization Feature Group
_SC_XOPEN_LEGACY	_XOPEN_LEGACY	Supports X/Open Legacy Feature Group
_SC_XOPEN_REALTIME	_XOPEN_REALTIME	Supports X/Open POSIX Realtime Feature Group
_SC_XOPEN_REALTIME_ THREADS	_XOPEN_REALTIME_THREADS	Supports X/Open POSIX Reatime Threads Feature Group
_SC_XOPEN_SHM	_XOPEN_SHM	Supports X/Open Shared Memory Feature Group
_SC_XOPEN_STREAMS	_POSIX_XOPEN_STREAMS	Supports XSI Streams option group
_SC_XOPEN_UNIX	_XOPEN_UNIX	Supports X/Open CAE Specification, August 1994, System Interfaces and Headers, Issue 4, Version 2
_SC_XOPEN_VERSION	_XOPEN_VERSION	Integer value indicates version of X/Open Portability Guide to which implementation conforms
_SC_XOPEN_XCU_VERSION	_XOPEN_XCU_VERSION	Integer value indicates version of XCU specification to which implementation conforms

The following options are not supported and return −1:

_SC_2_PBS	_POSIX2_PBS
_SC_2_PBS_ACCOUNTING	_POSIX2_PBS_ACCOUNTING

_SC_2_PBS_CHECKPOINT	_POSIX2_PBS_CHECKPOINT
_SC_2_PBS_LOCATE	_POSIX2_PBS_LOCATE
_SC_2_PBS_MESSAGE	_POSIX2_PBS_MESSAGE
_SC_2_PBS_TRACK	_POSIX2_PBS_TRACK
_SC_ADVISORY_INFO	_POSIX_ADVISORY_INFO
_SC_CPUTIME	_POSIX_CPUTIME
_SC_SPORADIC_SERVER	_POSIX_SPORADIC_SERVER
_SC_SS_REPL_MAX	_POSIX_SS_REPL_MAX
_SC_THREAD_CPUTIME	_POSIX_THREAD_CPUTIME
_SC_THREAD_SPORADIC_SERVER	_POSIX_THREAD_SPORADIC_SERVER
_SC_TRACE	_POSIX_TRACE
_SC_TRACE_EVENT_FILTER	_POSIX_TRACE_EVENT_FILTER
_SC_TRACE_EVENT_NAME_MAX	_POSIX_TRACE_EVENT_NAME_MAX
_SC_TRACE_INHERIT	_POSIX_TRACE_INHERIT
_SC_TRACE_LOG	_POSIX_TRACE_LOG
_SC_TRACE_NAME_MAX	_POSIX_TRACE_NAME_MAX
_SC_TRACE_SYS_MAX	_POSIX_TRACE_SYS_MAX
_SC_TRACE_USER_EVENT_MAX	_POSIX_TRACE_USER_EVENT_MAX
_SC_TYPED_MEMORY_OBJECTS	_POSIX_TYPED_MEMORY_OBJECTS

Return Values Upon successful completion, `sysconf()` returns the current variable value on the system. The value returned will not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's <limits.h>, <unistd.h> or <time.h>. With only a few obvious exceptions such as `_SC_AVPHYS_PAGES` and `_SC_NPROCESSORS_ONLN`, the value will not change during the lifetime of the calling process.

If *name* is an invalid value, `sysconf()` returns −1 and sets `errno` to indicate the error. If the variable corresponding to *name* is associated with functionality that is not supported by the system, `sysconf()` returns −1 without changing the value of *errno*.

Calling `sysconf()` with the following returns −1 without setting `errno`, because no maximum limit can be determined. The system supports at least the minimum values and can support higher values depending upon system resources.

Variable	Minimum supported value
_SC_AIO_MAX	_POSIX_AIO_MAX
_SC_ATEXIT_MAX	32

_SC_MQ_OPEN_MAX	32
_SC_THREAD_THREADS_MAX	_POSIX_THREAD_THREADS_MAX
_SC_THREAD_KEYS_MAX	_POSIX_THREAD_KEYS_MAX
_SC_THREAD_DESTRUCTOR_ITERATIONS	_POSIX_THREAD_DESTRUCTOR_ITERATIONS

The following SPARC and x86 platform variables return EINVAL:

_SC_COHER_BLKSZ	_SC_DCACHE_ASSOC
_SC_DCACHE_BLKSZ	_SC_DCACHE_LINESZ
_SC_DCACHE_SZ	_SC_DCACHE_TBLKSZ
_SC_ICACHE_ASSOC	_SC_ICACHE_BLKSZ
_SC_ICACHE_LINESZ	_SC_ICACHE_SZ
_SC_SPLIT_CACHE	

Errors The sysconf() function will fail if:

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

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	SPARC and x86
Interface Stability	Committed
MT-Level	MT-Safe, Async-Signal-Safe
Standard	See standards(5).

See Also pooladm(1M), zoneadm(1M), fpathconf(2), seteuid(2), setrlimit(2), confstr(3C), attributes(5), standards(5)

Notes A call to setrlimit() can cause the value of OPEN_MAX to change.

Multiplying sysconf(_SC_PHYS_PAGES) or sysconf(_SC_AVPHYS_PAGES) by sysconf(_SC_PAGESIZE) to determine memory amount in bytes can exceed the maximum values representable in a 32–bit signed or unsigned integer.

The value of CLK_TCK can be variable and it should not be assumed that CLK_TCK is a compile-time constant.

If the caller is in a non-global zone and the pools facility is active, sysconf(_SC_NPROCESSORS_CONF) and sysconf(_SC_NPROCESSORS_ONLN) return the number of processors in the processor set of the pool to which the zone is bound.

Name syslog, openlog, closelog, setlogmask – control system log

Synopsis

```
#include <syslog.h>

void openlog(const char *ident, int logopt, int facility);

void syslog(int priority, const char *message, .../* arguments */);

void closelog(void);

int setlogmask(int maskpri);
```

Description The `syslog()` function sends a message to `syslogd(1M)`, which, depending on the configuration of `/etc/syslog.conf`, logs it in an appropriate system log, writes it to the system console, forwards it to a list of users, or forwards it to `syslogd` on another host over the network. The logged message includes a message header and a message body. The message header consists of a facility indicator, a severity level indicator, a timestamp, a tag string, and optionally the process ID.

The message body is generated from the *message* and following arguments in the same manner as if these were arguments to `printf(3C)`, except that occurrences of `%m` in the format string pointed to by the *message* argument are replaced by the error message string associated with the current value of `errno`. A trailing NEWLINE character is added if needed.

Symbolic constants for use as values of the *logopt*, *facility*, *priority*, and *maskpri* arguments are defined in the `<syslog.h>` header.

Values of the *priority* argument are formed by ORing together a *severity level* value and an optional *facility* value. If no facility value is specified, the current default facility value is used.

Possible values of severity level include, in decreasing order:

`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, such as hard device errors.
`LOG_ERR`	Errors.
`LOG_WARNING`	Warning messages.
`LOG_NOTICE`	Conditions that are not error conditions, but that may require special handling.
`LOG_INFO`	Informational messages.
`LOG_DEBUG`	Messages that contain information normally of use only when debugging a program.

The facility indicates the application or system component generating the message. Possible facility values include:

LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_MAIL	The mail system.
LOG_DAEMON	System daemons, such as in.ftpd(1M).
LOG_AUTH	The authentication / security / authorization system: login(1), su(1M), getty(1M).
LOG_LPR	The line printer spooling system: lpr(1B), lpc(1B).
LOG_NEWS	Designated for the USENET network news system.
LOG_UUCP	Designated for the UUCP system; it does not currently use syslog().
LOG_CRON	The cron/at facility; crontab(1), at(1), cron(1M).
LOG_AUDIT	The audit facility, for example, auditd(1M).
LOG_LOCAL0	Designated for local use.
LOG_LOCAL1	Designated for local use.
LOG_LOCAL2	Designated for local use.
LOG_LOCAL3	Designated for local use.
LOG_LOCAL4	Designated for local use.
LOG_LOCAL5	Designated for local use.
LOG_LOCAL6	Designated for local use.
LOG_LOCAL7	Designated for local use.

The openlog() function sets process attributes that affect subsequent calls to syslog(). The *ident* argument is a string that is prepended to every message. The openlog() function uses the passed-in *ident* argument directly, rather than making a private copy of it. The *logopt* argument indicates logging options. Values for *logopt* are constructed by a bitwise-inclusive OR of zero or more of the following:

LOG_PID	Log the process ID with each message. This is useful for identifying specific daemon processes (for daemons that fork).
LOG_CONS	Write messages to the system console if they cannot be sent to syslogd(1M). This option is safe to use in daemon processes that have no controlling terminal, since syslog() forks before opening the console.

`LOG_NDELAY` Open the connection to `syslogd(1M)` immediately. Normally the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated.

`LOG_ODELAY` Delay open until `syslog()` is called.

`LOG_NOWAIT` Do not wait for child processes that have been forked to log messages onto the console. This option should be used by processes that enable notification of child termination using `SIGCHLD`, since `syslog()` may otherwise block waiting for a child whose exit status has already been collected.

The *facility* argument encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded. The initial default facility is `LOG_USER`.

The `openlog()` and `syslog()` functions may allocate a file descriptor. It is not necessary to call `openlog()` prior to calling `syslog()`.

The `closelog()` function closes any open file descriptors allocated by previous calls to `openlog()` or `syslog()`.

The `setlogmask()` function sets the log priority mask for the current process to *maskpri* and returns the previous mask. If the *maskpri* argument is 0, the current log mask is not modified. Calls by the current process 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 log mask allows all priorities to be logged.

Return Values

The `setlogmask()` function returns the previous log priority mask. The `closelog()`, `openlog()` and `syslog()` functions return no value.

Errors

No errors are defined.

Examples

EXAMPLE 1 Example of `LOG_ALERT` message.

This call logs a message at priority `LOG_ALERT`:

```
syslog(LOG_ALERT, "who: internal error 23");
```

The FTP daemon `ftpd` would make this call to `openlog()` to indicate that all messages it logs should have an identifying string of `ftpd`, should be treated by `syslogd(1M)` as other messages from system daemons are, should include the process ID of the process logging the message:

```
openlog("ftpd", LOG_PID, LOG_DAEMON);
```

Then it would make the following call to `setlogmask()` to indicate that messages at priorities from `LOG_EMERG` through `LOG_ERR` should be logged, but that no messages at any other priority should be logged:

EXAMPLE 1 Example of LOG_ALERT message. *(Continued)*

```
setlogmask(LOG_UPTO(LOG_ERR));
```

Then, to log a message at priority LOG_INFO, it would make the following call to syslog:

```
syslog(LOG_INFO, "Connection from host %d", CallingHost);
```

A locally-written utility could use the following call to syslog() to log a message at priority LOG_INFO to be treated by syslogd(1M) as other messages to the facility LOG_LOCAL2 are:

```
syslog(LOG_INFO|LOG_LOCAL2, "error: %m");
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See standards(5).

See Also

at(1), crontab(1), logger(1), login(1), lpc(1B), lpr(1B), auditd(1M), cron(1M), getty(1M), in.ftpd(1M), su(1M), syslogd(1M), printf(3C), syslog.conf(4), attributes(5), standards(5)

Name system – issue a shell command

Synopsis

```
#include <stdlib.h>

int system(const char *string);
```

Description The `system()` function causes *string* to be given to the shell as input, as if *string* had been typed as a command at a terminal. The invoker waits until the shell has completed, then returns the exit status of the shell in the format specified by `waitpid(3C)`.

If *string* is a null pointer, `system()` checks if the shell exists and is executable. If the shell is available, `system()` returns a non-zero value; otherwise, it returns `0`. The standard to which the caller conforms determines which shell is used. See `standards(5)`.

The `system()` function sets the `SIGINT` and `SIGQUIT` signals to be ignored, and blocks the `SIGCHLD` signal for the calling thread, while waiting for the command to terminate. The `system()` function does not affect the termination status of any child of the calling processes other than the process it creates.

The termination status of the process created by the `system()` function is not affected by the actions of other threads in the calling process (it is invisible to `wait(3C)`) or by the disposition of the `SIGCHLD` signal in the calling process, even if it is set to be ignored. No `SIGCHLD` signal is sent to the process containing the calling thread when the command terminates.

Return Values The `system()` function executes `posix_spawn(3C)` to create a child process running the shell that in turn executes the commands in *string*. If `posix_spawn()` fails, `system()` returns `−1` and sets `errno` to indicate the error; otherwise the exit status of the shell is returned.

Errors The `system()` function may set `errno` values as described by `fork(2)`, in particular:

`EAGAIN` A resource control or limit on the total number of processes, tasks or LWPs under execution by a single user, task, project, or zone has been exceeded, or the total amount of system memory available is temporarily insufficient to duplicate this process.

`ENOMEM` There is not enough swap space.

`EPERM` The {`PRIV_PROC_FORK`} privilege is not asserted in the effective set of the calling process.

Usage The `system()` function manipulates the signal handlers for `SIGINT` and `SIGQUIT`. It is therefore not safe to call `system()` in a multithreaded process, since some other thread that manipulates these signal handlers and a thread that concurrently calls `system()` can interfere with each other in a destructive manner. If, however, no such other thread is active, `system()` can safely be called concurrently from multiple threads. See `popen(3C)` for an alternative to `system()` that is thread-safe.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	See standards(5).

See Also ksh(1), sh(1), popen(3C), posix_spawn(3C), wait(3C), waitpid(3C), attributes(5), standards(5)

Name tcdrain – wait for transmission of output

Synopsis

```
#include <termios.h>

int tcdrain(int fildes);
```

Description The `tcdrain()` function waits until all output written to the object referred to by *fildes* is transmitted. The *fildes* argument is an open file descriptor associated with a terminal.

Any attempts to use `tcdrain()` from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no signal is sent.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `tcdrain()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

`EINTR` A signal interrupted `tcdrain()`.

`ENOTTY` The file associated with *fildes* is not a terminal.

The `tcdrain()` function may fail if:

`EIO` The process group of the writing process is orphaned, and the writing process is not ignoring or blocking `SIGTTOU`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe, and Async-Signal-Safe
Standard	See `standards(5)`.

See Also `tcflush(3C)`, `attributes(5)`, `standards(5)`, `termio(7I)`

Name tcflow – suspend or restart the transmission or reception of data

Synopsis

```
#include <termios.h>

int tcflow(int fildes, int action);
```

Description The `tcflow()` function suspends transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*. The *fildes* argument is an open file descriptor associated with a terminal.

- If *action* is `TCOOFF`, output is suspended.
- If *action* is `TCOON`, suspended output is restarted.
- 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.
- 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 default on the opening of a terminal file is that neither its input nor its output are suspended.

Attempts to use `tcflow()` from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no signal is sent.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `tcflow()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

`EINVAL` The *action* argument is not a supported value.

`ENOTTY` The file associated with *fildes* is not a terminal.

The `tcflow()` function may fail if:

`EIO` The process group of the writing process is orphaned, and the writing process is not ignoring or blocking `SIGTTOU`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe, and Async-Signal-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also tcsendbreak(3C), attributes(5), standards(5), termio(7I)

Name tcflush – flush non-transmitted output data, non-read input data or both

Synopsis

```
#include <termios.h>

int tcflush(int fildes, int queue_selector);
```

Description Upon successful completion, `tcflush()` discards data written to the object referred to by *fildes* (an open file descriptor associated with a terminal) but not transmitted, or data received but not read, depending on the value of *queue_selector*:

- If *queue_selector* is `TCIFLUSH` it flushes data received but not read.
- If *queue_selector* is `TCOFLUSH` it flushes data written but not transmitted.
- If *queue_selector* is `TCIOFLUSH` it flushes both data received but not read and data written but not transmitted.

Attempts to use `tcflush()` from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no signal is sent.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `tcflush()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

`EINVAL` The *queue_selector* argument is not a supported value.

`ENOTTY` The file associated with *fildes* is not a terminal.

The `tcflush()` function may fail if:

`EIO` The process group of the writing process is orphaned, and the writing process is not ignoring or blocking `SIGTTOU`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe, and Async-Signal-Safe
Standard	See `standards(5)`.

See Also `tcdrain(3C)`, `attributes(5)`, `standards(5)`, `termio(7I)`

Name tcgetattr – get the parameters associated with the terminal

Synopsis

```
#include <termios.h>

int tcgetattr(int fildes, struct termios *termios_p);
```

Description The `tcgetattr()` function gets the parameters associated with the terminal referred to by *fildes* and stores them in the `termios` structure (see `termio(7I)`) referenced by *termios_p*. The *fildes* argument is an open file descriptor associated with a terminal.

The *termios_p* argument is a pointer to a `termios` structure.

The `tcgetattr()` operation is allowed from any process.

If the terminal device supports different input and output baud rates, the baud rates stored in the `termios` structure returned by `tcgetattr()` 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 is the actual baud rate. If the terminal device does not support split baud rates, the input baud rate stored in the `termios` structure will be 0.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `tcgetattr()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

`ENOTTY` The file associated with *fildes* is not a terminal.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe, and Async-Signal-Safe
Standard	See `standards(5)`.

See Also `tcsetattr(3C)`, `attributes(5)`, `standards(5)`, `termio(7I)`

Name tcgetpgrp – get foreground process group ID

Synopsis

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

pid_t tcgetpgrp(int fildes);
```

Description The `tcgetpgrp()` function will return the value of the process group ID of the foreground process group associated with the terminal.

If there is no foreground process group, `tcgetpgrp()` returns a value greater than 1 that does not match the process group ID of any existing process group.

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.

Return Values Upon successful completion, `tcgetpgrp()` returns the value of the process group ID of the foreground process associated with the terminal. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `tcgetpgrp()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

`ENOTTY` The calling process does not have a controlling terminal, or the file is not the controlling terminal.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe, and Async-Signal-Safe
Standard	See `standards(5)`.

See Also `setpgid(2)`, `setsid(2)`, `tcsetpgrp(3C)`, `attributes(5)`, `standards(5)`, `termio(7I)`

Name tcgetsid – get process group ID for session leader for controlling terminal

Synopsis

```
#include <termios.h>

pid_t tcgetsid(int fildes);
```

Description The `tcgetsid()` function obtains the process group ID of the session for which the terminal specified by *fildes* is the controlling terminal.

Return Values Upon successful completion, `tcgetsid()` returns the process group ID associated with the terminal. Otherwise, a value of `(pid_t)-1` is returned and `errno` is set to indicate the error.

Errors The `tcgetsid()` function will fail if:

`EACCES` The *fildes* argument is not associated with a controlling terminal.

`EBADF` The *fildes* argument is not a valid file descriptor.

`ENOTTY` The file associated with *fildes* is not a terminal.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `attributes(5)`, `standards(5)`, `termio(7I)`

Name tcsendbreak – send a "break" for a specific duration

Synopsis

```
#include <termios.h>

int tcsendbreak(int fildes, int duration);
```

Description The *fildes* argument is an open file descriptor associated with a terminal.

If the terminal is using asynchronous serial data transmission, `tcsendbreak()` will cause transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is 0, it will cause transmission of zero-valued bits for at least 0.25 seconds, and not more than 0.5 seconds. If *duration* is not 0, it behaves in a way similar to `tcdrain(3C)`.

If the terminal is not using asynchronous serial data transmission, it sends data to generate a break condition or returns without taking any action.

Attempts to use `tcsendbreak()` from a process which is a member of a background process group on a *fildes* associated with its controlling terminal will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no signal is sent.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `tcsendbreak()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

`ENOTTY` The file associated with *fildes* is not a terminal.

The `tcsendbreak()` function may fail if:

`EIO` The process group of the writing process is orphaned, and the writing process is not ignoring or blocking `SIGTTOU`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe, and Async-Signal-Safe
Standard	See `standards(5)`.

See Also `tcdrain(3C)`, `attributes(5)`, `standards(5)`, `termio(7I)`

Name tcsetattr – set the parameters associated with the terminal

Synopsis

```
#include <termios.h>

int tcsetattr(int fildes, int optional_actions,
     const struct termios *termios_p);
```

Description The `tcsetattr()` function sets the parameters associated with the terminal referred to by the open file descriptor *fildes* (an open file descriptor associated with a terminal) from the `termios` structure (see `termio(7I)`) referenced by *termios_p* as follows:

- If *optional_actions* is `TCSANOW`, the change will occur immediately.
- If *optional_actions* is `TCSADRAIN`, the change will occur after all output written to *fildes* is transmitted. This function should be used when changing parameters that affect output.
- If *optional_actions* is `TCSAFLUSH`, the change will occur after all output written to *fildes* is transmitted, and all input so far received but not read will be discarded before the change is made.

If the output baud rate stored in the `termios` structure pointed to by *termios_p* is the zero baud rate, B0, the modem control lines will no longer be asserted. Normally, this will disconnect the line.

If the input baud rate stored in the `termios` structure pointed to by *termios_p* is 0, the input baud rate given to the hardware will be the same as the output baud rate stored in the `termios` structure.

The `tcsetattr()` function will return successfully if it was able to perform any of the requested actions, even if some of the requested actions could not be performed. It will set all the attributes that implementation supports as requested and leave all the attributes not supported by the implementation unchanged. If no part of the request can be honoured, it will return `−1` and set `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(3C)` will return the actual state of the terminal device (reflecting both the changes made and not made in the previous `tcsetattr()` call). The `tcsetattr()` function will not change the values in the `termios` structure whether or not it actually accepts them.

The effect of `tcsetattr()` 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(3C)` on *fildes*; an application should modify only fields and flags defined by this document between the call to `tcgetattr(3C)` and `tcsetattr()`, leaving all other fields and flags unmodified.

No actions defined by this document, other than a call to `tcsetattr()` or a close of the last file descriptor in the system associated with this terminal device, will cause any of the terminal attributes defined by this document to change.

Attempts to use `tcsetattr()` from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, will cause the process group to be

sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation, and no signal is sent.

Usage If trying to change baud rates, applications should call tcsetattr() then call tcgetattr(3C) in order to determine what baud rates were actually selected.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The tcsetattr() function will fail if:

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

EINTR A signal interrupted tcsettattr().

EINVAL The *optional_actions* argument is not a supported value, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.

ENOTTY The file associated with *fildes* is not a terminal.

The tcsetattr() function may fail if:

EIO The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe, and Async-Signal-Safe
Standard	See standards(5).

See Also cfgetispeed(3C), tcgetattr(3C), attributes(5), standards(5), termio(7I)

Name tcsetpgrp – set foreground process group ID

Synopsis

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

int tcsetpgrp(int fildes, pid_t pgid_id);
```

Description If the process has a controlling terminal, `tcsetpgrp()` will set the foreground process group ID associated with the terminal to *pgid_id*. The file associated with *fildes* 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 *pgid_id* must match a process group ID of a process in the same session as the calling process.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `tcsetpgrp()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

`EINVAL` This implementation does not support the value in the *pgid_id* argument.

`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.

`EIO` The process is not ignoring or holding `SIGTTOU` and is a member of an orphaned process group.

`EPERM` The value of *pgid_id* does not match the process group ID of a process in the same session as the calling process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe, and Async-Signal-Safe
Standard	See `standards(5)`.

See Also `tcgetpgrp(3C)`, `attributes(5)`, `standards(5)`, `termio(7I)`

Name td_init – performs initialization for libc_db library of interfaces

Synopsis

```
cc [ flag... ] file...  -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_init();
```

Description The `td_init()` function is the global initialization function for the `libc_db()` library of interfaces. It must be called exactly once by any process using the `libc_db()` library before any other `libc_db()` function can be called.

Return Values

`TD_OK` The `libc_db()` library of interfaces successfully initialized.

`TD_ERR` Initialization failed.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also `libc_db(3LIB)`, `attributes(5)`

Name td_log – placeholder for future logging functionality

Synopsis

```
cc [ flag... ] file...  -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

void td_log(void);
```

Description This function presently does nothing. It is merely a placeholder for future logging functionality in libc_db(3LIB).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also libc_db(3LIB), attributes(5), threads(5)

Name td_sync_get_info, td_ta_sync_tracking_enable, td_sync_get_stats, td_sync_setstate, td_sync_waiters – operations on a synchronization object in libc_db

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_sync_get_info(const td_synchandle_t *sh_p, td_syncinfo_t *si_p);

td_err_e td_ta_sync_tracking_enable(const td_thragent_t *ta_p, int on_off);

td_err_e td_sync_get_stats(const td_synchandle_t *sh_p, td_syncstats_t *ss_p);

td_err_e td_sync_setstate(const td_synchandle_t *sh_p);

typedef int td_thr_iter_f(const td_thrhandle_t *th_p, void *cb_data_p);

td_err_e td_sync_waiters(const td_synchandle_t *sh_p, td_thr_iter_f *cb,
     void *cb_data_p);
```

Description Synchronization objects include mutexes, condition variables, semaphores, and reader-writer locks. In the same way that thread operations use a thread handle of type `td_thrhandle_t`, operations on synchronization objects use a synchronization object handle of type `td_synchandle_t`.

The controlling process obtains synchronization object handles either by calling the function `td_ta_sync_iter()` to obtain handles for all synchronization objects of the target process that are known to the `libc_db` library of interfaces, or by mapping the address of a synchronization object in the address space of the target process to a handle by calling `td_ta_map_addr2sync(3C_DB)`.

Not all synchronization objects that a process uses can be known to the `libc_db` library and returned by `td_ta_sync_iter(3C_DB)`. A synchronization object is known to `libc_db` only if it has been the target of a synchronization primitive in the process (such as `mutex_lock()`, described on the `mutex_init(3C)` manual page) after `td_ta_new(3C_DB)` has been called to attach to the process and `td_ta_sync_tracking_enable()` has been called to enable synchronization object tracking.

The `td_ta_sync_tracking_enable()` function turns synchronization object tracking on or off for the process identified by *ta_p*, depending on whether *on_off* is 0 (off) or non-zero (on).

The `td_sync_get_info()` function fills in the `td_syncinfo_t` structure **si_p* with values for the synchronization object identified by `sh_p`. The `td_syncinfo_t` structure contains the following fields:

`td_thragent_t *si_ta_p` The internal process handle identifying the target process through which this synchronization object handle was obtained. Synchronization objects may be process-private or process-shared. In the latter case, the same synchronization object may have multiple handles, one for each target process's "view" of the synchronization object.

`psaddr_t` *si_sv_addr*	The address of the synchronization object in this target process's address space.
`td_sync_type_e` *si_type*	The type of the synchronization variable: mutex, condition variable, semaphore, or readers-writer lock.
`int` *si_shared_type*	If *si_shared_type* is non-zero, this synchronization object is process-shared, otherwise it is process-private.
`td_sync_flags_t` *si_flags*	Flags dependent on the type of the synchronization object.
`int` *si_state.sema_count*	Semaphores only. The current value of the semaphore
`int` *si_state.nreaders*	Readers-writer locks only. The number of readers currently holding the lock, or `-1`, if a writer is currently holding the lock.
`int` *si_state.mutex_locked*	For mutexes only. Non-zero if and only if the mutex is currently locked.
`int` *si_size*	The size of the synchronization object.
`uint8_t` *si_has_waiters*	Non-zero if and only if at least one thread is blocked on this synchronization object.
`uint8_t` *si_is_wlocked*	For reader-writer locks only. The value is non-zero if and only if this lock is held by a writer.
`uint8_t` *si_rcount*	`PTHREAD_MUTEX_RECURSIVE` mutexes only. If the mutex is held, the recursion count.
`uint8_t` *si_prioceiling*	`PTHREAD_PRIO_PROTECT` protocol mutexes only. The priority ceiling.
`td_thrhandle_t` *si_owner*	Mutexes and readers-writer locks only. This is the thread holding the mutex, or the write lock, if this is a reader-writer lock. The value is `NULL` if no one holds the mutex or write-lock.
`pid_t` *si_ownerpid*	Mutexes only. For a locked process-shared mutex, this is the process-ID of the process containing the owning thread.

The `td_sync_get_stats()` function fills in the `td_syncstats_t` structure **ss_p* with values for the synchronization object identified by *sh_p*. The `td_syncstats_t` structure contains an embedded `td_syncinfo_t` structure that is filled in as described above for `td_sync_get_info()`. In addition, usage statistics gathered since `td_ta_sync_tracking_enable()` was called to enable synchronization object tracking are returned in the `ss_un.mutex`, `ss_un.cond`, `ss_un.rwlock`, or `ss_un.sema` members of the `td_syncstats_t` structure, depending on the type of the synchronization object.

The `td_sync_setstate` function modifies the state of synchronization object *si_p*, depending on the synchronization object type. For mutexes, `td_sync_setstate` is unlocked if the value is

0. Otherwise it is locked. For semaphores, the semaphore's count is set to the value. For reader-writer locks, the reader count set to the value if value is >0. The count is set to write-locked if value is –1. It is set to unlocked if the value is 0. Setting the state of a synchronization object from a libc_db interface may cause the synchronization object's semantics to be violated from the point of view of the threads in the target process. For example, if a thread holds a mutex, and td_sync_setstate is used to set the mutex to unlocked, then a different thread will also be able to subsequently acquire the same mutex.

The td_sync_waiters function iterates over the set of thread handles of threads blocked on *sh_p*. The callback function *cb* is called once for each such thread handle, and is passed the thread handle and *cb_data_p*. If the callback function returns a non-zero value, iteration is terminated early. See td_ta_thr_iter(3C_DB).

Return Values

TD_OK	The call returned successfully.
TD_BADTH	An invalid thread handle was passed in.
TD_DBERR	A call to one of the imported interface routines failed.
TD_ERR	A libc_db-internal error occurred.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also

libc_db(3LIB), mutex_init(3C), td_ta_map_addr2sync(3C_DB), td_ta_sync_iter(3C_DB), td_ta_thr_iter(3C_DB), attributes(5)

Name td_ta_enable_stats, td_ta_reset_stats, td_ta_get_stats – collect target process statistics for libc_db

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_ta_enable_stats(const td_thragent_t *ta_p, int on_off);

td_err_e_stats td_ta_reset(const td_thragent_t *ta_p);

td_err_e td_ta_get_stats(const td_thragent_t *ta_p, td_ta_stats_t *tstats);
```

Description The controlling process can request the collection of certain statistics about a target process. Statistics gathering is disabled by default. Each target process has a `td_ta_stats_t` structure that contains current values when statistic gathering is enabled.

The `td_ta_enable_stats()` function turns statistics gathering on or off for the process identified by *ta_p*, depending on whether or not *on_off* is non-zero. When statistics gathering is turned on, all statistics are implicitly reset as though `td_ta_reset_stats()` had been called. Statistics are not reset when statistics gathering is turned off. Except for `nthreads` and `r_concurrency`, the values do not change further, but they remain available for inspection by way of `td_ta_get_stats()`.

The `td_ta_reset_stats()` function resets all counters in the `td_ta_stats_t` structure to zero for the target process.

The `td_ta_get_stats()` function returns the structure for the process in *tstats*.

The `td_ta_stats_t` structure is defined in <`thread_db.h`> and contains the following members:

```
typedef struct {
   int nthreads;            /* total number of threads in use */
   int r_concurrency;       /* requested concurrency level */
   int nrunnable_num;       /* numerator of avg runnable threads */
   int nrunnable_den;       /* denominator of avg runnable threads */
   int a_concurrency_num;   /* numerator, avg achieved concurrency */
   int a_concurrency_den;   /* denominator, avg achieved concurrency */
   int nlwps_num;           /* numerator, avg number of LWPs in use */
   int nlwps_den;           /* denominator, avg number of LWPs in use */
   int nidle_num;           /* numerator, avg number of idling LWPs */
   int nidle_den;           /* denominator, avg number of idling LWPs */
} td_ta_stats_t;
```

The `nthreads` member is the number of threads that are currently part of the target process. The `r_concurrency` member is the current requested concurrency level, such as would be returned by `thr_setconcurrency(3C)`. The remaining members are averages over time, each expressed as a fraction with an integral numerator and denominator. The `nrunnable_num` and `nrunnable_den` members represent the average number of runnable threads. The

a_concurrency_num and a_concurrency_den members represent the average achieved concurrency, the number of actually running threads. The a_concurrency_num and a_concurrency_den members are less than or equal to nrunnable_num and nrunnable_den, respectively. The nlwps_num and nlwps_den members represent the average number of lightweight processes (LWPs) participating in this process. They must be greater than or equal to a_concurrency_num and a_concurrency_den, respectively, since every running thread is assigned to an LWP, but there can at times be additional idling LWPs with no thread assigned to them. The nidle_num and nidle_den members represent the average number of idle LWPs.

Return Values

TD_OK	The call completed successfully.
TD_BADTA	An invalid internal process handle was passed in.
TD_DBERR	A call to one of the imported interface routines failed.
TD_ERR	Something else went wrong.

Attributes

See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also

libc_db(3LIB), thr_getconcurrency(3C), attributes(5)

Name td_ta_event_addr, td_thr_event_enable, td_ta_set_event, td_thr_set_event, td_ta_clear_event, td_thr_clear_event, td_ta_event_getmsg, td_thr_event_getmsg, td_event_emptyset, td_event_fillset, td_event_addset, td_event_delset, td_eventismember, td_eventisempty – thread events in libc_db

Synopsis

```
cc [ flag... ] file...  -lc_db  [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_ta_event_addr(const td_thragent_t *ta_p, u_long event,td_notify_t *notify_p);

td_err_e td_thr_event_enable(const td_thrhandle_t *th_p, int on_off);

td_err_e td_thr_set_event(const td_thrhandle_t *th_p, td_thr_events_t *events);

td_err_e td_ta_set_event(const td_thragent_t *ta_p, td_thr_events_t *events);

td_err_e td_thr_clear_event(const td_thrhandle_t *th_p, td_thr_events_t *events);

td_err_e td_ta_clear_event(const td_thragent_t *ta_p, td_thr_events_t *events);

td_err_e td_thr_event_getmsg(const td_thrhandle_t *th_p, td_event_msg_t *msg);

td_err_e td_ta_event_getmsg(const td_thragent_t *ta_p, td_event_msg_t *msg);

void td_event_emptyset(td_thr_events_t  *);

void td_event_fillset(td_thr_events_t *);

void td_event_addset(td_thr_events_t *, td_thr_events_e n);

void td_event_delset(td_thr_events_t *, td_thr_events_e n);

void td_eventismember(td_thr_events_t *, td_thr_events_e n);

void td_eventisempty(td_thr_events_t*);
```

Description These functions comprise the thread event facility for libc_db(3LIB). This facility allows the controlling process to be notified when certain thread-related events occur in a target process and to retrieve information associated with these events. An event consists of an event type, and optionally, some associated event data, depending on the event type. See the section titled "Event Set Manipulation Macros" that follows.

The event type and the associated event data, if any, constitute an "event message." "Reporting an event" means delivering an event message to the controlling process by way of libc_db.

Several flags can control event reporting, both a per-thread and per event basis. Event reporting may further be enabled or disabled for a thread. There is not only a per-thread event mask that specifies which event types should be reported for that thread, but there is also a global event mask that applies to all threads.

An event is reported, if and only if, the executing thread has event reporting enabled, and either the event type is enabled in the executing thread's event mask, or the event type is enabled in the global event mask.

Each thread has associated with it an event buffer in which it stores the most recent event message it has generated, the type of the most recent event that it reported, and, depending on the event type, some additional information related to that event. See the section titled "Event Set Manipulation Macros" for a description of the `td_thr_events_e` and `td_event_msg_t` types and a list of the event types and the values reported with them. The thread handle, type `td_thrhandle_t`, the event type, and the possible value, together constitute an event message. Each thread's event buffer holds at most one event message.

Each event type has an event reporting address associated with it. A thread reports an event by writing the event message into the thread's event buffer and having control reach the event reporting address for that event type.

Typically, the controlling process sets a breakpoint at the event reporting address for one or more event types. When the breakpoint is hit, the controlling process knows that an event of the corresponding type has occurred.

The event types, and the additional information, if any, reported with each event, are:

`TD_READY`	The thread became ready to execute.
`TD_SLEEP`	The thread has blocked on a synchronization object.
`TD_SWITCHTO`	A runnable thread is being assigned to LWP.
`TD_SWITCHFROM`	A running thread is being removed from its LWP.
`TD_LOCK_TRY`	A thread is trying to get an unavailable lock.
`TD_CATCHSIG`	A signal was posted to a thread.
`TD_IDLE`	An LWP is becoming idle.
`TD_CREATE`	A thread is being created.
`TD_DEATH`	A thread has terminated.
`TD_PREEMPT`	A thread is being preempted.
`TD_PRI_INHERIT`	A thread is inheriting an elevated priority from another thread.
`TD_REAP`	A thread is being reaped.
`TD_CONCURRENCY`	The number of LWPs is changing.
`TD_TIMEOUT`	A condition-variable timed wait expired.

The `td_ta_event_addr()` function returns in **notify_p* the event reporting address associated with event type `event`. The controlling process may then set a breakpoint at that address. If a thread hits that breakpoint, it reports an event of type `event`.

The `td_thr_event_enable()` function enables or disables event reporting for thread *th_p*. If a thread has event reporting disabled, it will not report any events. Threads are started with

event reporting disabled. Event reporting is enabled if `on_off` is non-zero; otherwise, it is disabled. To determine whether or not event reporting is enabled on a thread, call `td_thr_getinfo()` for the thread and examine the `ti_traceme` member of the `td_thrinfo_t` structure it returns.

The `td_thr_set_event()` and `td_thr_clear_event()` functions set and clear, respectively, a set of event types in the event mask associated with the thread *th_p*. To inspect a thread's event mask, call `td_thr_getinfo()` for the thread and examine the `ti_events` member of the `td_thrinfo_t` structure it returns.

The `td_ta_set_event()` and `td_ta_clear_event()` functions identical to `td_thr_set_event()` and `td_thr_clear_event()`, respectively, except that the target process's global event mask is modified. There is no provision for inspecting the value of a target process's global event mask.

The `td_thr_event_getmsg()` function returns in **msg* the event message associated with thread **th_p*. Reading a thread's event message consumes the message, emptying the thread's event buffer. As noted above, each thread's event buffer holds at most one event message; if a thread reports a second event before the first event message has been read, the second event message overwrites the first.

The `td_ta_event_getmsg()` function is identical to `td_thr_event_getmsg()`, except that it is passed a process handle rather than a thread handle. It selects some thread that has an event message buffered and returns that thread's message. The thread selected is undefined, except that as long as at least one thread has an event message buffered, it returns an event message from some such thread.

Event Set Manipulation Macros

Several macros are provided for manipulating event sets of type `td_thr_events_t`:

`td_event_emptyset`	Sets its argument to the `NULL` event set.
`td_event_fillset`	Sets its argument to the set of all events.
`td_event_addset`	Adds a specific event type to an event set.
`td_event_delset`	Deletes a specific event type from an event set.
`td_eventismember`	Tests whether a specific event type is a member of an event set.
`td_eventisempty`	Tests whether an event set is the `NULL` set.

Return Values

The following values may be returned for all thread event routines:

`TD_OK`	The call returned successfully.
`TD_BADTH`	An invalid thread handle was passed in.
`TD_BADTA`	An invalid internal process handle was passed.

`TD_BADPH` There is a `NULL` external process handle associated with this internal process handle.

`TD_DBERR` A call to one of the imported interface routines failed.

`TD_NOMSG` No event message was available to return to `td_thr_event_getmsg()` or `td_ta_event_getmsg()`.

`TD_ERR` Some other parameter error occurred, or a `libc_db()` internal error occurred.

The following value can be returned for `td_thr_event_enable()`, `td_thr_set_event()`, and `td_thr_clear_event()` only:

`TD_NOCAPAB` Because the agent thread in the target process has not completed initialization, this operation cannot be performed. The operation can be performed after the target process has been allowed to make some forward progress. See `libc_db(3LIB)`.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `attributes(5)`

Name td_ta_get_nthreads – gets the total number of threads in a process for libc_db

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_ta_get_nthreads(const td_thragent_t *ta_p, int *nthread_p);
```

Description The `td_ta_get_nthreads()` function returns the total number of threads in process *ta_p*, including any system threads. System threads are those created by `libc` or `libc_db` on its own behalf. The number of threads is written into **nthread_p*.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTA`	An invalid internal process handle was passed in.
`TD_BADPH`	There is a `NULL` external process handle associated with this internal process handle.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_ERR`	The *nthread_p* argument was `NULL`, or a `libc_db` internal error occurred.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `attributes(5)`, `threads(5)`

Name td_ta_map_addr2sync – get a synchronization object handle from a synchronization object's address

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_ta_map_addr2sync(const td_thragent_t *ta_p, psaddr_t addr, td_synchandle_t *sh_p);
```

Description The `td_ta_map_addr2sync()` function produces the synchronization object handle of type `td_synchandle_t` that corresponds to the address of the synchronization object (mutex, semaphore, condition variable, or reader/writer lock). Some effort is made to validate *addr* and verify that it does indeed point at a synchronization object. The handle is returned in **sh_p*.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTA`	An invalid internal process handle was passed.
`TD_BADPH`	There is a `NULL` external process handle associated with this internal process handle.
`TD_BADSH`	The *sh_p* argument is `NULL` or *addr* does not appear to point to a valid synchronization object.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_ERR`	`addr` is `NULL`, or a `libc_db` internal error occurred.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `attributes(5)`

Name td_ta_map_id2thr, td_ta_map_lwp2thr – convert a thread ID or LWP ID to a thread handle

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_ta_map_id2thr(const td_thragent_t *ta_p, thread_t tid,td_thrhandle_t *th_p);

td_ta_map_lwp2thr(const td_thragent_t *ta_p, lwpid_t lwpid,td_thrhandle_t *th_p);
```

Description The `td_ta_map_id2thr()` function produces the `td_thrhandle_t` thread handle that corresponds to a particular thread ID, as returned by `thr_create(3C)` or `thr_self(3C)`. The thread handle is returned in **th_p*.

The `td_ta_map_lwp2thr()` function produces the `td_thrhandle_t` thread handle for the thread that is currently executing on the light weight process (LWP) and has an ID of `lwpid`.

Return Values

`TD_OK` The call completed successfully.

`TD_BADTA` An invalid internal process handle was passed in.

`TD_BADPH` There is a `NULL` external process handle associated with this internal process handle.

`TD_DBERR` A call to one of the imported interface routines failed.

`TD_NOTHR` Either there is no thread with the given thread ID (`td_ta_map_id2thr`) or no thread is currently executing on the given LWP (`td_ta_map_lwp2thr`).

`TD_ERR` The call did not complete successfully.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `thr_create(3C)`, `thr_self(3C)`, `attributes(5)`

Name td_ta_new, td_ta_delete, td_ta_get_ph – allocate and deallocate process handles for libc_db

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_ta_new(const struct ps_prochandle *ph_p, td_thragent_t **ta_pp);

td_err_e td_ta_delete(const td_thragent_t *ta_p);

td_err_e td_ta_get_ph(const td_thragent_t *ta_p, struct ps_prochandle **ph_pp);
```

Description The `td_ta_new()` function registers a target process with `libc_db(3LIB)` and allocates an internal process handle of type `td_thragent_t` for this target process. Subsequent calls to `libc_db` can use this handle to refer to this target process.

There are actually two process handles, an internal process handle assigned by `libc_db` and an external process handle assigned by the `libc_db` client. There is a one-to-one correspondence between the two handles. When the client calls a `libc_db` function, it uses the internal process handle. When `libc_db` calls one of the client-provided routines listed in `proc_service(3PROC)`, it uses the external process handle.

The *ph* argument is the external process handle that `libc_db` should use to identify this target process to the controlling process when it calls routines in the imported interface.

If this call is successful, the value of the newly allocated `td_thragent_t` handle is returned in **ta_pp*. The `td_ta_delete()` function deregisters a target process with `libc_db`, which deallocates its internal process handle and frees any other resources `libc_db` has acquired with respect to the target process. The *ta_p* argument specifies the target process to be deregistered.

The `td_ta_get_ph()` function returns in **ph_pp* the external process handle that corresponds to the internal process handle *ta_p*. This is useful for checking internal consistency.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADPH`	A `NULL` external process handle was passed to `td_ta_new()`.
`TD_ERR`	The *ta_pp* argument is `NULL` or an internal error occurred.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_MALLOC`	Memory allocation failure.
`TD_NOLIBTHREAD`	The target process does not appear to be multithreaded.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also libc_db(3LIB), proc_service(3PROC), attributes(5)

Name td_ta_setconcurrency – set concurrency level for target process

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_ta_setconcurrency(const td_thragent_t *ta_p, int level);
```

Description The `td_ta_setconcurrency()` function sets the desired concurrency level for the process identified by *ta_p* to level, just as if a thread within the process had called `thr_setconcurrency(3C)`.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTA`	An invalid internal process handle was passed in.
`TD_BADPH`	There is a `NULL` external process handle associated with this internal process handle. `TD_NOCAPAB` The client did not implement the `ps_kill(3PROC)` function in the imported interface.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_ERR`	A `libc_db` internal error occurred.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `ps_kill(3PROC)`, `thr_setconcurrency(3C)`, `attributes(5)`

Name td_ta_sync_iter, td_ta_thr_iter, td_ta_tsd_iter – iterator functions on process handles from libc_db

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

typedef int td_sync_iter_f(const td_synchandle_t *sh_p, void *cbdata_p);

typedef int td_thr_iter_f(const td_thrhandle_t *th_p, void *cbdata_p);

typedef int td_key_iter_f(thread_key_t key, void (*destructor)(), void *cbdata_p);

td_err_e td_ta_sync_iter(const td_thragent_t *ta_p, td_sync_iter_f *cb,
     void *cbdata_p);

td_err_e td_ta_thr_iter(const td_thragent_t *ta_p, td_thr_iter_f *cb,
     void *cbdata_p, td_thr_state_e state, int ti_pri, sigset_t *ti_sigmask_p,
     unsigned ti_user_flags);

td_err_e td_ta_tsd_iter(const td_thragent_t *ta_p, td_key_iter_f *cb,
     void *cbdata_p);
```

Description The `td_ta_sync_iter()`, `td_ta_thr_iter()`, and `td_ta_tsd_iter()` functions are iterator functions that when given a target process handle as an argument, return sets of handles for objects associated with the target process. The method is to call back a client-provided function once for each associated object, passing back a handle as well as the client-provided pointer *cb_data_p*. This enables a client to easily build a linked list of the associated objects. If the client-provided function returns non-zero, the iteration terminates, even if there are members remaining in the set of associated objects.

The `td_ta_sync_iter()` function returns handles of synchronization objects (mutexes, readers-writer locks, semaphores, and condition variables) associated with a process. Some synchronization objects might not be known to `libc_db` and will not be returned. If the process has initialized the synchronization object (by calling `mutex_init(3C)`, for example) or a thread in the process has called a synchronization primitive (`mutex_lock()`, for example) using this object after `td_ta_new(3C_DB)` was called to attach to the process and `td_ta_sync_tracking_enable()` was called to enable synchronization object tracking, then a handle for the synchronization object will be passed to the callback function. See `td_sync_get_info(3C_DB)` for operations that can be performed on synchronization object handles.

The `td_ta_thr_iter()` function returns handles for threads that are part of the target process. For `td_ta_thr_iter()`, the caller specifies several criteria to select a subset of threads for which the callback function should be called. Any of these selection criteria may be wild-carded. If all of them are wild-carded, then handles for all threads in the process will be returned.

The selection parameters and corresponding wild-card values are:

`state (TD_THR_ANY_STATE)`:	Select only threads whose state matches `state`. See `td_thr_get_info(3C_DB)` for a list of thread states.
`ti_pri (TD_THR_LOWEST_PRIORITY)`:	Select only threads for which the priority is at least `ti_pri`.
`ti_sigmask_p (TD_SIGNO_MASK)`:	Select only threads whose signal mask exactly matches **ti_sigmask_p*.
`ti_user_flags (TD_THR_ANY_USER_FLAGS)`:	Select only threads whose user flags (specified at thread creation time) exactly match *ti_user_flags*.

The `td_ta_tsd_iter()` function returns the thread-specific data keys in use by the current process. Thread-specific data for a particular thread and key can be obtained by calling `td_thr_tsd(3C_DB)`.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTA`	An invalid process handle was passed.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_ERR`	The call did not complete successfully.

Attributes

See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also

`libc_db(3LIB)`, `mutex_init(3C)`, `td_sync_get_info(3C_DB)`, `td_thr_get_info(3C_DB)`, `td_thr_tsd(3C_DB)`, `attributes(5)`

Name td_thr_dbsuspend, td_thr_dbresume – suspend and resume threads in libc_db

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_thr_dbsuspend(const td_thrhandle_t *th_p);

td_err_e td_thr_dbresume(const td_thrhandle_t *th_p);
```

Description These operations do nothing other than call `ps_lstop(3PROC)` and `ps_lcontinue(3PROC)`, respectively, on the lightweight process (LWP) identified by the thread handle, *th_p*. Since `ps_lstop()` and `ps_lcontinue()` must be provided by the caller's application (see `proc_service(3PROC)`), and the application (a debugger-like entity) has full control over the stopped state of the process and all of its LWPs, `td_thr_dbsuspend()` and `td_thr_dbresume()` are unnecessary interfaces. They exist only to maintain interface compatibility with the past.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTH`	An invalid thread handle was passed in.
`TD_DBERR`	A call to `ps_lstop()` or `ps_lcontinue()` failed.
`TD_ERR`	A `libc_db` internal error occurred.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `attributes(5)`

Name td_thr_getgregs, td_thr_setgregs, td_thr_getfpregs, td_thr_setfpregs, td_thr_getxregsize, td_thr_getxregs, td_thr_setxregs – reading and writing thread registers in libc_db

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_thr_getgregs(const td_thrhandle_t *th_p, prgregset_tgregset);

td_err_e td_thr_setgregs(const td_thrhandle_t *th_p, prgregset_tgregset);

td_err_e td_thr_getfpregs(const td_thrhandle_t *th_p, prfpregset_t *fpregset);

td_err_e td_thr_setfpregs(const td_thrhandle_t *th_p, prfpregset_t *fpregset);

td_err_e td_thr_getxregsize(const td_thrhandle_t *th_p, int *xregsize);

td_err_e td_thr_getxregs(const td_thrhandle_t *th_p, prxregset_t *xregset);

td_err_e td_thr_setxregs(const td_thrhandle_t *th_p, prxregset_t *xregset);
```

Description These functions read and write the register sets associated with thread *th_p*. The `td_thr_getgregs()` and `td_thr_setgregs()` functions get and set, respectively, the general registers of thread *th_p*. The `td_thr_getfpregs()` and `td_thr_setfpregs()` functions get and set, respectively, the thread's floating point register set. The `td_thr_getxregsize()`, `td_thr_getxregs()`, and `td_thr_setxregs()` functions are SPARC-specific. The `td_thr_getxregsize()` function returns in **xregsize* the size of the architecture-dependent extra state registers. The `td_thr_getxregs()` and `td_thr_setxregs()` functions get and set, respectively, those extra state registers. On non-SPARC architectures, these functions return `TD_NOXREGS`.

If the thread specified by *th_p* is currently executing on a lightweight process (LWP), these functions read or write, respectively, the appropriate register set to the LWP using the imported interface. If the thread is not currently executing on an LWP, the floating point and extra state registers may cannot be read or written. Some of the general registers might also not be readable or writable, depending on the architecture, in which case `td_thr_getfpregs()` and `td_thr_setfpregs()` return `TD_NOFPREGS` and `td_thr_getxregs()` and `td_thr_setxregs()` will `TD_NOXREGS`. Calls to `td_thr_getgregs()` and `td_thr_setgregs()` succeed, but values returned for unreadable registers are undefined, values specified for unwritable registers are ignored. In this instance, and `TD_PARTIALREGS` is returned. See the architecture-specific notes that follow regarding the registers that may be read and written for a thread not currently executing on an LWP.

SPARC On a thread not currently assigned to an LWP, only %i0-%i7, %l0-%l7, %g7, %pc, and %sp (%o6) can be read or written. %pc and %sp refer to the program counter and stack pointer that the thread will have when it resumes execution.

x86 Architecture On a thread not currently assigned to an LWP, only %pc, %sp, %ebp, %edi, %edi, and %ebx can be read.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTH`	An invalid thread handle was passed in.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_PARTIALREGS`	Because the thread is not currently assigned to a LWP, not all registers were read or written. See `DESCRIPTION` for a discussion about which registers are not saved when a thread is not assigned to an LWP.
`TD_NOFPREGS`	Floating point registers could not be read or written, either because the thread is not currently assigned to an LWP, or because the architecture does not have such registers.
`TD_NOXREGS`	Architecture-dependent extra state registers could not be read or written, either because the thread is not currently assigned to an LWP, or because the architecture does not have such registers, or because the architecture is not a SPARC architecture.
`TD_ERR`	A `libc_db` internal error occurred.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `attributes(5)`

Name td_thr_get_info – get thread information in libc_db library of interfaces

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_thr_get_info(const td_thrhandle_t *th_p, td_thrinfo_t *ti_p);
```

Description The `td_thr_get_info()` function fills in the `td_thrinfo_t` structure **ti_p* with values for the thread identified by *th_p*.

The `td_thrinfo_t` structure contains the following fields:

```
typedef struct td_thrinfo_t {
  td_thragen_tx    *ti_ta_p           /* internal process handle */
  unsigned         ti_user_flags;     /* value of flags parameter */
  thread_t         ti_tid;            /* thread identifier */
  char             *ti_tls;           /* pointer to thread-local storage*/
  paddr            ti_startfunc;      /* address of function at which thread
                                         execution began*/
  paddr            ti_stkbase;        /* base of thread's stack area*/
  int              ti_stksize;        /* size in bytes of thread's allocated
                                         stack region*/
  paddr            ti_ro_area;        /* address of ulwp_t structure*/
  int              ti_ro_size         /* size of the ulwp_t structure in
                                         bytes */
  td_thr_state_e   ti_state           /* state of the thread */
  uchar_t          ti_db_suspended    /* non-zero if thread suspended by
                                         td_thr_dbsuspend*/
  td_thr_type_e    ti_type            /* type of the thread*/
  int              ti_pc              /* value of thread's program counter*/
  int              ti_sp              /* value of thread's stack counter*/
  short            ti_flags           /* set of special flags used by
                                         libc*/
  int              ti_pri             /* priority of thread returned by
                                         thr_getprio(3T)*/
  lwpid_t          ti_lid             /* id of light weight process (LWP)
                                         executing this thread*/
  sigset_t         ti_sigmask         /* thread's signal mask.  See
                                         thr_sigsetmask(3T)*/
  u_char           ti_traceme         /* non-zero if event tracing is on*/
  u_char_t         ti_preemptflag     /* non-zero if thread preempted when
                                         last active*/
  u_char_t         ti_pirecflag      /* non-zero if thread runs priority
                                        beside regular */
  sigset_t         ti_pending        /* set of signals pending for this
                                        thread*/
  td_thr_events_t  ti_events         /* bitmap of events enabled for this
                                        thread*/
} ;
```

The ti_ta_p member is the internal process handle identifying the process of which the thread is a member.

The ti_user_flags member is the value of the flags parameter passed to thr_create(3C) when the thread was created.

The ti_tid member is the thread identifier for the thread returned by thr_create(3C).

The ti_tls member is the thread's pointer to thread-local storage.

The ti_startfunc member is the address of the function at which thread execution began, as specified when the thread was created with thr_create(3C).

The ti_stkbase member is the base of the thread's stack area.

The ti_stksize member is the size in bytes of the thread's allocated stack region.

The ti_ro_area member is the address of the ulwp_t structure for this thread. Since accessing the ulwp_t structure directly violates the encapsulation provided by libc_db, this member should generally not be used. However, it might be useful as a prototype for extensions.

The ti_state member is the state of the thread. The td_thr_state_e enumeration type can contain the following values:

TD_THR_ANY_STATE	This value is never returned by td_thr_get_info() but is used as a wildcard to select threads in td_ta_thr_iter().
TD_THR_UNKNOWN	The libc_db library cannot determine the state of the thread.
TD_THR_STOPPED	The thread has been stopped by a call to thr_suspend(3C).
TD_THR_RUN	The thread is runnable, but it is not currently assigned to an LWP.
TD_THR_ACTIVE	The thread is currently executing on an LWP.
TD_THR_ZOMBIE	The thread has exited, but it has not yet been deallocated by a call to thr_join(3C).
TD_THR_SLEEP	The thread is not currently runnable.
TD_THR_STOPPED_ASLEEP	The thread is both blocked by TD_THR_SLEEP and stopped by a call to td_thr_dbsuspend(3C_DB).

The ti_db_suspended member is non-zero if and only if this thread is currently suspended because the controlling process has called td_thr_dbsuspend on it.

The ti_type member is a type of thread. It is either TD_THR_USER for a user thread (one created by the application), or TD_THR_SYSTEM for one created by libc.

The `ti_pc` member is the value of the thread's program counter, provided that the thread's `ti_state` value is `TD_THR_SLEEP`, `TD_THR_STOPPED`, or `TD_THR_STOPPED_ASLEEP`. Otherwise, the value of this member is undefined.

The `ti_sp` member is the value of the thread's stack pointer, provided that the thread's `ti_state` value is `TD_THR_SLEEP`, `TD_THR_STOPPED`, or `TD_THR_STOPPED_ASLEEP`. Otherwise, the value of this member is undefined.

The `ti_flags` member is a set of special flags used by `libc`, currently of use only to those debugging `libc`.

The `ti_pri` member is the thread's priority as it would be returned by `thr_getprio(3C)`.

The `ti_lid` member is the ID of the LWP executing this thread, or the ID of the LWP that last executed this thread, if this thread is not currently assigned to anLWP.

The `ti_sigmask` member is this thread's signal mask. See `thr_sigsetmask(3C)`.

The `ti_traceme` member is non-zero if and only if event tracing for this thread is on.

The `ti_preemptflag` member is non-zero if and only if the thread was preempted the last time it was active.

The `ti_pirecflag` member is non-zero if and only if due to priority inheritance the thread is currently running at a priority other than its regular priority.

The `ti_events` member is the bitmap of events enabled for this thread.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTH`	An invalid thread handle was passed in.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_ERR`	The call did not complete successfully.

Attributes

See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also

`libc_db(3LIB)`, `td_ta_thr_iter(3C_DB)`, `td_thr_dbsuspend(3C_DB)`, `thr_create(3C)`, `thr_getprio(3C)`, `thr_join(3C)`, `thr_sigsetmask(3C)`, `thr_suspend(3C)`, `attributes(5)`, `threads(5)`

Name td_thr_lockowner – iterate over the set of locks owned by a thread

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_thr_lockowner(const td_thrhandle_t *th_p, td_sync_iter_f *cb,
     void *cb_data_p);
```

Description The `td_thr_lockowner()` function calls the iterator function *cb* once for every mutex that is held by the thread whose handle is *th_p*. The synchronization handle and the pointer *cb_data_p* are passed to the function. See `td_ta_thr_iter`(3C_DB) for a similarly structured function.

Iteration terminates early if the callback function *cb* returns a non-zero value.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTH`	An invalid thread handle was passed in.
`TD_BADPH`	There is a `NULL` external process handle associated with this internal process handle.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_ERR`	A `libc_db` internal error occurred.

Attributes See `attributes`(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db`(3LIB), `td_ta_thr_iter`(3C_DB), `attributes`(5)

Name td_thr_setprio – set the priority of a thread

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_thr_setprio(const td_thrhandle_t *th_p,
     const int new_prio);
```

Description The `td_thr_setprio()` function is obsolete. It always fails and returns `TD_NOCAPAB`.

Return Values

`TD_NOCAPAB` Capability not available.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `thr_setprio(3C)`, `attributes(5)`

Name td_thr_setsigpending, td_thr_sigsetmask – manage thread signals for libc_db

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_thr_setsigpending(const td_thrhandle_t * th_p, const uchar_ t ti_sigpending_flag,
     const sigset_t ti_sigmask;);

td_err_e td_thr_sigsetmask(const td_thrhandle_t *th_p, const sigset_t ti_sigmask);
```

Description The `td_thr_setsigpending()` and `td_thr_setsigmask()` functions affect the signal state of the thread identified by *th_p*.

The `td_thr_setsigpending()` function sets the set of pending signals for thread *th_p* to *ti_sigpending*. The value of the `libc`-internal field that indicates whether a thread has any signal pending is set to *ti_sigpending_flag*. To be consistent, *ti_sigpending_flag* should be 0 if and only if all of the bits in *ti_sigpending* are 0.

The `td_thr_sigsetmask()` function sets the signal mask of the thread *th_p* as if the thread had set its own signal mask with `thr_sigsetmask(3C)`. The new signal mask is the value of `ti_sigmask`.

There is no equivalent to the `SIG_BLOCK` or `SIG_UNBLOCK` operations of `thr_sigsetmask(3C)`, which mask or unmask specific signals without affecting the mask state of other signals. To block or unblock specific signals,

1. stop either the entire process or the thread with `td_thr_dbsuspend()`,
2. determine the thread's existing signal mask by calling `td_thr_get_info(3C_DB)`,
3. modify the *ti_sigmask* member of the `td_thrinfo_t` structure as desired, and
4. set the new signal mask with`td_thr_sigsetmask()`.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTH`	An invalid thread handle was passed in.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_ERR`	A `libc_db` internal error occurred.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `td_thr_dbsuspend(3C_DB)`, `td_thr_get_info(3C_DB)`, `attributes(5)`

Name td_thr_sleepinfo – return the synchronization handle for the object on which a thread is blocked

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_thr_sleepinfo(const td_thrhandle_t *th_p, td_synchandle_t *sh_p);
```

Description The `td_thr_sleepinfo()` function returns in **sh_p* the handle of the synchronization object on which a sleeping thread is blocked.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTH`	An invalid thread handle was passed in.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_ERR`	The thread *th_p* is not blocked on a synchronization object, or a `libc_db` internal error occurred.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `attributes(5)`

Name td_thr_tsd – get a thread's thread-specific data for libc_db library of interfaces

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_thr_tsd(const td_thrhandle_t, const thread_key_t key, void *data_pp);
```

Description The `td_thr_tsd()` function returns in **data_pp* the thread-specific data pointer for the thread identified by *th_p* and the thread-specific data key *key*. This is the same value that the thread *th_p* would obtain if it called `thr_getspecific(3C)`.

To find all the thread-specific data keys in use in a given target process, call `td_ta_tsd_iter(3C_DB)`.

Return Values

`TD_OK`	The call completed successfully.
`TD_BADTH`	An invalid thread handle was passed in.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_ERR`	A `libc_db` internal error occurred.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `td_ta_tsd_iter(3C_DB)`, `thr_getspecific(3C)`, `attributes(5)`

Name td_thr_validate – test a thread handle for validity

Synopsis

```
cc [ flag... ] file... -lc_db [ library... ]
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_thr_validate(const td_thrhandle_t *th_p);
```

Description The `td_thr_validate()` function tests whether *th_p* is a valid thread handle. A valid thread handle can become invalid if its thread exits.

Return Values

`TD_OK`	The call completed successfully. *th_p* is a valid thread handle.
`TD_BADTH`	*th_p* was `NULL`.
`TD_DBERR`	A call to one of the imported interface routines failed.
`TD_NOTHR`	*th_p* is not a valid thread handle.
`TD_ERR`	A `libc_db` internal error occurred.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libc_db(3LIB)`, `attributes(5)`

Name tell – return a file offset for a file descriptor

Synopsis

```
#include <unistd.h>

off_t tell(int fd);
```

Description The `tell()` function obtains the current value of the file-position indicator for the file descriptor *fd*.

Return Values Upon successful completion, `tell()` returns the current value of the file-position indicator for *fd* measured in bytes from the beginning of the file.

Otherwise, it returns −1 and sets `errno` to indicate the error.

Errors The `tell()` function will fail if:

`EBADF` The file descriptor *fd* is not an open file descriptor.

`EOVERFLOW` The current file offset cannot be represented correctly in an object of type `off_t`.

`ESPIPE` The file descriptor *fd* is associated with a pipe or FIFO.

Usage The `tell()` function is equivalent to `lseek(`*fd*`, 0, SEEK_CUR)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `lseek(2)`, `attributes(5)`

Name telldir – current location of a named directory stream

Synopsis

```
#include <dirent.h>

long int telldir(DIR *dirp);
```

Description The `telldir()` function obtains the current location associated with the directory stream specified by *dirp*.

If the most recent operation on the directory stream was a `seekdir(3C)`, the directory position returned from the `telldir()` is the same as that supplied as a *loc* argument for `seekdir()`.

Return Values Upon successful completion, `telldir()` returns the current location of the specified directory stream.

Errors The `telldir()` function will fail if:

`EOVERFLOW` The current location of the directory cannot be stored in an object of type `long`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `opendir(3C)`, `readdir(3C)`, `seekdir(3C)`, `attributes(5)`, `standards(5)`

Name termios – general terminal interface

Synopsis

```
#include <termios.h>

int tcgetattr(int fildes, struct termios *termios_p);

int tcsetattr(int fildes, int optional_actions,
     const struct termios *termios_p);

int tcsendbreak(int fildes, int duration);

int tcdrain(int fildes);

int tcflush(int fildes, int queue_selector);

int tcflow(int fildes, int action);

speed_t cfgetospeed(const struct termios *termios_p);

int cfsetospeed(struct termios *termios_p, speed_t speed);

speed_t cfgetispeed(const struct termios *termios_p);

int cfsetispeed(struct termios *termios_p, speed_t speed);

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

pid_t tcgetpgrp(int fildes);

int tcsetpgrp(int fildes, pid_t pgid);

pid_t tcgetsid(int fildes);
```

Description These functions describe a general terminal interface for controlling asynchronous communications ports. A more detailed overview of the terminal interface can be found in `termio(7I)`, which also describes an `ioctl(2)` interface that provides the same functionality. However, the function interface described by these functions is the preferred user interface.

Each of these functions is now described on a separate manual page.

See Also `ioctl(2)`, `cfgetispeed(3C)`, `cfgetospeed(3C)`, `cfsetispeed(3C)`, `cfsetospeed(3C)`, `tcdrain(3C)`, `tcflow(3C)`, `tcflush(3C)`, `tcgetattr(3C)`, `tcgetpgrp(3C)`, `tcgetsid(3C)`, `tcsendbreak(3C)`, `tcsetattr(3C)`, `tcgetpgrp(3C)`, `tcsendbreak(3C)`, `termio(7I)`

Name thr_create – create a thread

Synopsis

```
cc –mt [ flag... ] file...[ library... ]
#include <thread.h>

int thr_create(void *stack_base, size_t stack_size,
     void *(*start_func) (void*), void *arg, long flags,
     thread_t *new_thread_ID);
```

Description Thread creation adds a new thread of control to the current process. The procedure `main()` is a single thread of control. Each thread executes concurrently with all other threads within the calling process and with other threads from other active processes.

Although a newly created thread shares all of the calling process's global data with the other threads in the process, it has its own set of attributes and private execution stack. The new thread inherits the calling thread's signal mask and scheduling priority. Pending signals for a new thread are not inherited and will be empty.

The call to create a thread takes the address of a user-defined function, specified by *start_func*, as one of its arguments. This function is the complete execution routine for the new thread.

The lifetime of a thread begins with the successful return from `thr_create()`, which calls *start_func*( ) and ends with one of the following:

- the normal completion of *start_func*( ),
- an explicit call to `thr_exit(3C)`, or
- the conclusion of the calling process (see `exit(2)`).

The new thread performs by calling the function defined by *start_func* with only one argument, *arg*. If more than one argument needs to be passed to *start_func*, the arguments can be packed into a structure, the address of which can be passed to *arg*.

If *start_func* returns, the thread terminates with the exit status set to the *start_func* return value (see `thr_exit(3C)`).

When the thread from which `main()` originated returns, the effect is the same as if an implicit call to `exit()` were made using the return value of `main()` as the exit status. This behavior differs from a *start_func* return. If `main()` calls `thr_exit(3C)`, only the `main` thread exits, not the entire process.

If the thread creation fails, a new thread is not created and the contents of the location referenced by the pointer to the new thread are undefined.

The *flags* argument specifies which attributes are modifiable for the created thread. The value in *flags* is determined by the bitwise inclusive-OR of the following:

`THR_BOUND` This flag is obsolete and is maintained for compatibility.

THR_DETACHED — This flag affects the detachstate attribute of the thread. The new thread is created detached. The exit status of a detached thread is not accessible to other threads. Its thread ID and other resources may be re-used as soon as the thread terminates. thr_join(3C) will not wait for a detached thread.

THR_NEW_LWP — This flag is obsolete and is maintained for compatibility.

THR_SUSPENDED — This flag affects the suspended attribute of the thread. The new thread is created suspended and will not execute *start_func* until it is started by thr_continue().

THR_DAEMON — This flag affects the daemon attribute of the thread. In addition to being created detached (THR_DAEMON implies THR_DETACHED), the thread is marked as a daemon. Daemon threads do not interfere with the exit conditions for a process. A process will terminate when the last non-daemon thread exits or the process calls exit(2). Also, a thread that is waiting in thr_join(3C) for any thread to terminate will return EDEADLK when all remaining threads in the process are either daemon threads or other threads waiting in thr_join(). Daemon threads are most useful in libraries that want to use threads.

Default thread creation:

```
thread_t tid;
void *start_func(void *), *arg;
thr_create(NULL, 0, start_func, arg, 0, &tid);
```

Create a detached thread whose thread ID we do not care about:

```
thr_create(NULL, 0, start_func, arg, THR_DETACHED, NULL);
```

If *stack_base* is not NULL, the new thread uses the stack beginning at the address specified by *stack_base* and continuing for *stack_size* bytes, where *stack_size* must be greater than or equal to THR_MIN_STACK. If *stack_base* is NULL, thr_create() allocates a stack for the new thread with at least *stack_size* bytes. If *stack_size* is 0, a default size is used. If *stack_size* is not 0, it must be greater than or equal to THR_MIN_STACK. See NOTES.

When *new_thread_ID* is not NULL, it points to a location where the ID of the new thread is stored if thr_create() is successful. The ID is only valid within the calling process.

Return Values If successful, the thr_create() function returns 0. Otherwise, an error value is returned to indicate the error.

Errors EAGAIN — A resource control limit on the total number of threads in a process, task, project, or zone has been exceeded or some system resource has been exceeded.

EINVAL The *stack_base* argument is not NULL and *stack_size* is less than THR_MIN_STACK, or the *stack_base* argument is NULL and *stack_size* is not 0 and is less than THR_MIN_STACK.

ENOMEM The system cannot allocate stack for the thread.

The thr_create() function may use mmap() to allocate thread stacks from MAP_PRIVATE, MAP_NORESERVE, and MAP_ANON memory mappings if *stack_base* is NULL, and consequently may return upon failure the relevant error values returned by mmap(). See the mmap(2) manual page for these error values.

Examples

The following is an example of concurrency with multithreading. Since POSIX threads and Solaris threads are fully compatible even within the same process, this example uses pthread_create() if you execute a.out 0, or thr_create() if you execute a.out 1.

Five threads are created that simultaneously perform a time-consuming function, sleep(10). If the execution of this process is timed, the results will show that all five individual calls to sleep for ten-seconds completed in about ten seconds, even on a uniprocessor. If a single-threaded process calls sleep(10) five times, the execution time will be about 50-seconds.

The command-line to time this process is:

```
/usr/bin/time a.out 0 (for POSIX threading)
```

or

```
/usr/bin/time a.out 1 (for Solaris threading)
```

EXAMPLE 1 An example of concurrency with multithreading.

```
#define _REENTRANT    /* basic 3-lines for threads */
#include <pthread.h>
#include <thread.h>
#define NUM_THREADS 5
#define SLEEP_TIME 10

void *sleeping(void *);   /* thread routine */
int i;
thread_t tid[NUM_THREADS];      /* array of thread IDs */

int
main(int argc, char *argv[])
{
    if (argc == 1)  {
        printf("use 0 as arg1 to use pthread_create( )\n");
        printf("or use 1 as arg1 to use thr_create( )\n");
        return (1);
```

EXAMPLE 1 An example of concurrency with multithreading. *(Continued)*

```
    }

    switch (*argv[1])  {
    case '0':  /* POSIX */
        for ( i = 0; i < NUM_THREADS; i++)
                pthread_create(&tid[i], NULL, sleeping,
                    (void *)SLEEP_TIME);
        for ( i = 0; i < NUM_THREADS; i++)
                    pthread_join(tid[i], NULL);
        break;

    case '1':  /* Solaris */
        for ( i = 0; i < NUM_THREADS; i++)
            thr_create(NULL, 0, sleeping, (void *)SLEEP_TIME, 0,
                &tid[i]);
        while (thr_join(0, NULL, NULL) == 0)
                continue;
        break;
    }  /* switch */
    printf("main( ) reporting that all %d threads have
        terminated\n", i);
    return (0);
}  /* main */

void *
sleeping(void *arg)
{
    int sleep_time = (int)arg;
    printf("thread %d sleeping %d seconds ...\n", thr_self( ),
        sleep_time);
    sleep(sleep_time);
    printf("\nthread %d awakening\n", thr_self( ));
    return (NULL);
}
```

Had `main()` not waited for the completion of the other threads (using `pthread_join(3C)` or `thr_join(3C)`), it would have continued to process concurrently until it reached the end of its routine and the entire process would have exited prematurely (see `exit(2)`).

EXAMPLE 2 Creating a default thread with a new signal mask.

The following example demonstrates how to create a default thread with a new signal mask. The *new_mask* argument is assumed to have a value different from the creator's signal mask (*orig_mask*). The *new_mask* argument is set to block all signals except for `SIGINT`. The

EXAMPLE 2 Creating a default thread with a new signal mask. *(Continued)*

creator's signal mask is changed so that the new thread inherits a different mask, and is restored to its original value after `thr_create()` returns.

This example assumes that `SIGINT` is also unmasked in the creator. If it is masked by the creator, then unmasking the signal opens the creator to this signal. The other alternative is to have the new thread set its own signal mask in its start routine.

```
thread_t tid;
sigset_t new_mask, orig_mask;
int error;

(void)sigfillset(&new_mask);
(void)sigdelset(&new_mask, SIGINT);
(void)thr_sigsetmask(SIG_SETMASK, &new_mask, &orig_mask);
error = thr_create(NULL, 0, do_func, NULL, 0, &tid);
(void)thr_sigsetmask(SIG_SETMASK, &orig_mask, NULL);
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also

`exit(2)`, `getrlimit(2)`, `mmap(2)`, `exit(3C)`, `sleep(3C)`, `thr_exit(3C)`, `thr_join(3C)`, `thr_min_stack(3C)`, `thr_setconcurrency(3C)`, `thr_suspend(3C)`, `attributes(5)`, `standards(5)`, `threads(5)`

Notes

Since multithreaded-application threads execute independently of each other, their relative behavior is unpredictable. It is therefore possible for the thread executing `main()` to finish before all other user-application threads.

Using `thr_join(3C)` in the following syntax,

```
while (thr_join(0, NULL, NULL) == 0)
    continue;
```

will cause the invoking thread (which may be `main()`) to wait for the termination of all non-daemon threads, excluding threads that are themselves waiting in `thr_join()`; however, the second and third arguments to `thr_join()` need not necessarily be `NULL`.

A thread has not terminated until `thr_exit()` has finished. The only way to determine this is by `thr_join()`. When `thr_join()` returns a departed thread, it means that this thread has terminated and its resources are reclaimable. For instance, if a user specified a stack to `thr_create()`, this stack can only be reclaimed after `thr_join()` has reported this thread as a departed thread. It is not possible to determine when a *detached* thread has terminated. A detached thread disappears without leaving a trace.

Typically, thread stacks allocated by `thr_create()` begin on page boundaries and any specified (a red-zone) size is rounded up to the next page boundary. A page with no access permission is appended to the top of the stack so that most stack overflows will result in a `SIGSEGV` signal being sent to the offending thread. Thread stacks allocated by the caller are used as is.

Using a default stack size for the new thread, instead of passing a user-specified stack size, results in much better `thr_create()` performance. The default stack size for a user-thread is 1 megabyte in a 32-bit process and 2 megabyte in a 64-bit process.

A user-specified stack size must be greater than or equal to `THR_MIN_STACK`. A minimum stack size may not accommodate the stack frame for the user thread function *start_func*. If a stack size is specified, it must accommodate *start_func* requirements and the functions that it may call in turn, in addition to the minimum requirement.

It is usually very difficult to determine the runtime stack requirements for a thread. `THR_MIN_STACK` specifies how much stack storage is required to execute a trivial *start_func*. The total runtime requirements for stack storage are dependent on the storage required to do runtime linking, the amount of storage required by library runtimes (like `printf()`) that your thread calls. Since these storage parameters are not known before the program runs, it is best to use default stacks. If you know your runtime requirements or decide to use stacks that are larger than the default, then it makes sense to specify your own stacks.

Name thr_exit – terminate the calling thread

Synopsis

```
cc –mt [ flag... ] file...[ library... ]
#include <thread.h>

void thr_exit(void *status);
```

Description The `thr_exit()` function terminates the calling thread, in a similar way that `exit(3C)` terminates the calling process. If the calling thread is not detached, then the thread's ID and the exit status specified by *status* are retained. The value *status* is then made available to any successful join with the terminating thread (see `thr_join(3C)`); otherwise, *status* is disregarded allowing the thread's ID to be reclaimed immediately.

Any cancellation cleanup handlers that have been pushed and not yet popped are popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions will be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process level cleanup actions, including, but not limited to, calling any `atexit()` routines that might exist.

An exiting thread runs with all signals blocked. All thread termination functions, including cancellation cleanup handlers and thread-specific data destructor functions, are called with all signals blocked.

If any thread, including the `main()` thread, calls `thr_exit()`, only that thread will exit.

If `main()` returns or exits (either implicitly or explicitly), or any thread explicitly calls `exit()`, the entire process will exit.

The behavior of `thr_exit()` is undefined if called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to `thr_exit()`.

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the `thr_exit()` *status* parameter value.

If any thread (except the `main()` thread) implicitly or explicitly returns, the result is the same as if the thread called `thr_exit()` and it will return the value of *status* as the exit code.

The process will terminate with an exit status of `0` after the last non-daemon thread has terminated (including the `main()` thread). This behavior is the same as if the application had called `exit()` with a `0` argument at thread termination time.

Return Values The `thr_exit()` function cannot return to its caller.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `exit(3C)`, `thr_create(3C)`, `thr_join(3C)`, `thr_keycreate(3C)`, `attributes(5)`, `standards(5)`

Notes Although only POSIX implements cancellation, cancellation can be used with Solaris threads, due to their interoperability.

The *status* argument should not reference any variables local to the calling thread.

Name thr_getconcurrency, thr_setconcurrency – get or set thread concurrency level

Synopsis

```
cc –mt [ flag... ] file...[ library... ]
#include <thread.h>

int thr_setconcurrency(int new_level);

int thr_getconcurrency(void);
```

Description These functions are obsolete and maintained for compatibility only. The `thr_setconcurrency()` function updates the desired concurrency level that `libthread` maintains for the calling process. This value does not affect the behavior of the calling process.

The `thr_getconcurrency()` function returns the current value for the desired concurrency level.

Return Values The `thr_getconcurrency()` function always returns the current value for the desired concurrency level.

If successful, the `thr_setconcurrency()` function returns `0`. Otherwise, a non-zero value is returned to indicate the error.

Errors The `thr_setconcurrency()` function will fail if:

`EAGAIN` The specified concurrency level would cause a system resource to be exceeded.

`EINVAL` The value for *new_level* is negative.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `thr_create(3C)`, `attributes(5)`, `standards(5)`

Name thr_getprio, thr_setprio – access dynamic thread scheduling

Synopsis

```
cc –mt [ flag... ] file...[ library... ]
#include <thread.h>

int thr_setprio(thread_t target_thread, int priority);

int thr_getprio(thread_t target_thread, int *priority);
```

Description The `thr_setprio()` function sets the scheduling priority for the thread specified by target_thread within the current process to the value given by priority.

The `thr_getprio()` function stores the current priority for the thread specified by *target_thread* in the location pointed to by *priority*.

If the `thr_setprio()` function fails, the scheduling priority of the target thread is not changed.

See `priocntl(2)`, `pthread_setschedprio(3C)`, and `sched_setparam(3C)`.

Return Values If successful, the `thr_getprio()` and `thr_setprio()` functions return `0`. Otherwise, an error number is returned to indicate the error.

Errors The `thr_getprio()` and `thr_setprio()` functions will fail if:

`ESRCH` The value specified by *target_thread* does not refer to an existing thread.

The `thr_setprio()` function will fail if:

`EINVAL` The value of *priority* is invalid for the scheduling policy of the specified thread.

`EPERM` The caller does not have the appropriate permission to set the priority to the value specified.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `priocntl(2)`, `pthread_setschedprio(3C)`, `sched_setparam(3C)`, `thr_create(3C)`, `thr_suspend(3C)`, `thr_yield(3C)`, `attributes(5)`, `standards(5)`

Name thr_join – wait for thread termination

Synopsis

```
cc –mt [ flag... ] file...[ library... ]
#include <thread.h>

int thr_join(thread_t thread, thread_t *departed, void **status);
```

Description The `thr_join()` function suspends processing of the calling thread until the target *thread* completes. The *thread* argument must be a member of the current process and cannot be a detached thread. See `thr_create(3C)`.

If two or more threads wait for the same thread to complete, all will suspend processing until the thread has terminated, and then one thread will return successfully and the others will return with an error of `ESRCH`. The `thr_join()` function will not block processing of the calling thread if the target *thread* has already terminated.

If a `thr_join()` call returns successfully with a non-null *status* argument, the value passed to `thr_exit(3C)` by the terminating thread will be placed in the location referenced by *status*.

If the target *thread* ID is `0`, `thr_join()` finds and returns the status of a terminated undetached thread in the process. If no such thread exists, it suspends processing of the calling thread until a thread for which no other thread is waiting enters that state, at which time it returns successfully, or until all other threads in the process are either daemon threads or threads waiting in `thr_join()`, in which case it returns `EDEADLK`. See `NOTES`.

If *departed* is not `NULL`, it points to a location that is set to the ID of the terminated thread if `thr_join()` returns successfully.

Return Values If successful, `thr_join()` returns `0`. Otherwise, an error number is returned to indicate the error.

Errors

`EDEADLK` A joining deadlock would occur, such as when a thread attempts to wait for itself, or the calling thread is waiting for any thread to exit and only daemon threads or waiting threads exist in the process.

`ESRCH` No undetached thread could be found corresponding to the given thread ID.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `thr_create(3C)`, `thr_exit(3C)`, `wait(3C)`, `attributes(5)`, `standards(5)`

Notes Using `thr_join(3C)` in the following syntax,

```
while (thr_join(0, NULL, NULL) == 0);
```

will wait for the termination of all non-daemon threads, excluding threads that are themselves waiting in `thr_join()`.

Name thr_keycreate, thr_keycreate_once, thr_setspecific, thr_getspecific – thread-specific data functions

Synopsis

```
cc –mt [ flag... ] file... [ library... ]
#include <thread.h>

int thr_keycreate(thread_key_t *keyp,
     void (*destructor)(void *));

int thr_keycreate_once(thread_key_t *keyp,
     void (*destructor)(void *));

int thr_setspecific(thread_key_t key, void *value);

int thr_getspecific(thread_key_t key, void **valuep);
```

Description

Create Key

In general, thread key creation allocates a key that locates data specific to each thread in the process. The key is global to all threads in the process, which allows each thread to bind a value to the key once the key has been created. The key independently maintains specific values for each binding thread. The `thr_keycreate()` function allocates a global *key* namespace, pointed to by *keyp*, that is visible to all threads in the process. Each thread is initially bound to a private element of this *key*, which allows access to its thread-specific data.

Upon key creation, a new key is assigned the value `NULL` for all active threads. Additionally, upon thread creation, all previously created keys in the new thread are assigned the value `NULL`.

Optionally, a destructor function *destructor* can be associated with each *key*. Upon thread exit, if a *key* has a non-null *destructor* function and the thread has a non-null *value* associated with that *key*, the *destructor* function is called with the current associated *value*. If more than one *destructor* exists for a thread when it exits, the order of destructor calls is unspecified.

An exiting thread runs with all signals blocked. All thread termination functions, including thread-specific data destructor functions, are called with all signals blocked.

The `thr_keycreate_once()` function is identical to the `thr_keycreate()` function except that the key pointed to by *keyp* must be statically initialized with the value `THR_ONCE_KEY` before calling `thr_keycreate_once()` and the key will be created exactly once. This is equivalent to using `pthread_once()` to call a onetime initialization function that calls `thr_keycreate()` to create the data key.

Set Value

Once a key has been created, each thread can bind a new *value* to the key using `thr_setspecific()`. The values are unique to the binding thread and are individually maintained. These values continue for the life of the calling thread.

Proper synchronization of *key* storage and access must be ensured by the caller. The *value* argument to `thr_setspecific()` is generally a pointer to a block of dynamically allocated memory reserved by the calling thread for its own use. See `EXAMPLES` below.

At thread exit, the *destructor* function, which is associated at time of creation, is called and it uses the specific key value as its sole argument.

Get Value `thr_getspecific()` stores the current value bound to *key* for the calling thread into the location pointed to by *valuep*.

Return Values If successful, `thr_keycreate()`, `thr_keycreate_once()`, `thr_setspecific()` and `thr_getspecific()` return 0. Otherwise, an error number is returned to indicate the error.

Errors If the following conditions occur, `thr_keycreate()` and `thr_keycreate_once()` return the corresponding error number:

`EAGAIN` The system lacked the necessary resources to create another thread-specific data key.

`ENOMEM` Insufficient memory exists to create the key.

If the following conditions occur, `thr_setspecific()` returns the corresponding error number:

`ENOMEM` Insufficient memory exists to associate the value with the key.

The `thr_setspecific()` function returns the corresponding error number:

`EINVAL` The *key* value is invalid.

Examples **EXAMPLE 1** Call the thread-specific data from more than one thread without special initialization.

In this example, the thread-specific data in this function can be called from more than one thread without special initialization. For each argument passed to the executable, a thread is created and privately bound to the string-value of that argument.

```
/* cc -mt thisfile.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <thread.h>

void *thread_specific_data(void *);
void cleanup(void*);
#define MAX_ARGC 20
thread_t tid[MAX_ARGC];
int num_threads;

int
main(int argc, char *argv[]) {
  int i;
  num_threads = argc - 1;
  for (i = 0; i < num_threads; i++)
```

EXAMPLE 1 Call the thread-specific data from more than one thread without special initialization. *(Continued)*

```
     thr_create(NULL, 0, thread_specific_data, argv[i+1], 0, &tid[i]);
  for (i = 0; i < num_threads; i++)
     thr_join(tid[i], NULL, NULL);
  return (0);
} /* end main */

void *
thread_specific_data(void *arg) {
  static thread_key_t key = THR_ONCE_KEY;
  char *private_data = arg;
  void *tsd = NULL;
  void *data;

  thr_keycreate_once(&key, cleanup);
  thr_getspecific(key, &tsd);
  if (tsd == NULL) {
       data = malloc(strlen(private_data) + 1);
       strcpy(data, private_data);
       thr_setspecific(key, data);
       thr_getspecific(key, &tsd);
  }
  printf("tsd for %d = %s\n", thr_self(), (char *)tsd);
  thr_getspecific(key, &tsd);
  printf("tsd for %d remains %s\n", thr_self(), (char *)tsd);
  return (NULL);
} /* end thread_specific_data */

void
cleanup(void *v) {
  /* application-specific clean-up function */
  free(v);
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

pthread_once(3C), thr_exit(3C), attributes(5), standards(5)

Warnings The `thr_getspecific()` and `thr_setspecific()` functions can be called either explicitly or implicitly from a thread-specific data destructor function. Calling `thr_setspecific()` from a destructor can result in lost storage or infinite loops.

Name thr_kill – send a signal to a thread

Synopsis

```
cc –mt [ flag... ] file... [ library... ]
#include <signal.h>
#include <thread.h>

int thr_kill(thread_t thread, int sig);
```

Description The `thr_kill()` function sends the *sig* signal to the thread designated by *thread*. The *thread* argument must be a member of the same process as the calling thread. The *sig* argument must be one of the signals listed in `signal.h`(3HEAD), with the exception of `SIGCANCEL` being reserved and off limits to `thr_kill()`. If *sig* is `0`, a validity check is done for the existence of the target thread; no signal is sent.

Return Values Upon successful completion, `thr_kill()` returns `0`. Otherwise, an error number is returned. In the event of failure, no signal is sent.

Errors The `thr_kill()` function will fail if:

`EINVAL` The *sig* argument value is not zero and is an invalid or an unsupported signal number.

`ESRCH` No thread was found that corresponded to the thread designated by *thread* ID.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also `kill`(2), `sigaction`(2), `raise`(3C), `signal.h`(3HEAD), `thr_self`(3C), `attributes`(5), `standards`(5)

Name thr_main – identify the main thread

Synopsis

```
cc –mt [ flag... ] file... [ library... ]
#include <thread.h>

int thr_main(void);
```

Description The `thr_main()` function returns one of the following:

`1` if the calling thread is the main thread

`0` if the calling thread is not the main thread

`-1` if `libthread` is not linked in or thread initialization has not completed

Files `/lib/libthread`

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `thr_self(3C)`, `attributes(5)`

Name thr_min_stack – return the minimum-allowable size for a thread's stack

Synopsis

```
cc –mt [ flag... ] file...[ library... ]
#include <thread.h>

size_t thr_min_stack(void);
```

Description When a thread is created with a user-supplied stack, the user must reserve enough space to run this thread. In a dynamically linked execution environment, it is very hard to know what the minimum stack requirments are for a thread. The function `thr_min_stack()` returns the amount of space needed to execute a null thread. This is a thread that was created to execute a null procedure. A thread that does something useful should have a stack size that is `thr_min_stack()` + <*some increment*>.

Most users should not be creating threads with user-supplied stacks. This functionality was provided to support applications that wanted complete control over their execution environment.

Typically, users should let the threads library manage stack allocation. The threads library provides default stacks which should meet the requirements of any created thread.

`thr_min_stack()` will return the unsigned int `THR_MIN_STACK`, which is the minimum-allowable size for a thread's stack.

In this implementation the default size for a user-thread's stack is one mega-byte. If the second argument to `thr_create(3C)` is NULL, then the default stack size for the newly-created thread will be used. Otherwise, you may specify a stack-size that is at least `THR_MIN_STACK`, yet less than the size of your machine's virtual memory.

It is recommended that the default stack size be used.

To determine the smallest-allowable size for a thread's stack, execute the following:

```
/* cc thisfile.c -lthread */
#define _REENTRANT
#include <thread.h>
#include <stdio.h>
main( )  {
        printf("thr_min_stack( ) returns %u\n",thr_min_stack( ));
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also attributes(5), standards(5)

Name thr_self – get calling thread's ID

Synopsis

```
cc –mt [ flag... ] file...[ library... ]
#include <thread.h>

thread_t thr_self(void);

typedef(unsigned int thread_t);
```

Description `thr_self()` returns the thread ID of the calling thread.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `thr_create(3C)`, `attributes(5)`, `standards(5)`

Name thr_sigsetmask – change or examine calling thread's signal mask

Synopsis

```
cc –mt [ flag... ] file... [ library... ]
#include <thread.h>
#include <signal.h>

int thr_sigsetmask(int how, const sigset_t *set, sigset_t *oset);
```

Description The `thr_sigsetmask()` function changes or examines a calling thread's signal mask. Each thread has its own signal mask. A new thread inherits the calling thread's signal mask and priority; however, pending signals are not inherited. Signals pending for a new thread will be empty.

If the value of the argument *set* is not `NULL`, `set` points to a set of signals that can modify the currently blocked set. If the value of *set* is `NULL`, the value of *how* is insignificant and the thread's signal mask is unmodified; thus, `thr_sigsetmask()` can be used to inquire about the currently blocked signals.

The value of the argument *how* specifies the method in which the set is changed and takes one of the following values:

`SIG_BLOCK` *set* corresponds to a set of signals to block. They are added to the current signal mask.

`SIG_UNBLOCK` *set* corresponds to a set of signals to unblock. These signals are deleted from the current signal mask.

`SIG_SETMASK` *set* corresponds to the new signal mask. The current signal mask is replaced by `set`.

If the value of *oset* is not `NULL`, it points to the location where the previous signal mask is stored.

Return Values Upon successful completion, the `thr_sigsetmask()` function returns `0`. Otherwise, it returns a non-zero value.

Errors The `thr_sigsetmask()` function will fail if:

`EINVAL` The value of *how* is not defined and *oset* is `NULL`.

Examples **EXAMPLE 1** Create a default thread that can serve as a signal catcher/handler with its own signal mask.

The following example shows how to create a default thread that can serve as a signal catcher/handler with its own signal mask. `new` will have a different value from the creator's signal mask.

As POSIX threads and Solaris threads are fully compatible even within the same process, this example uses `pthread_create(3C)` if you execute `a.out 0`, or `thr_create(3C)` if you execute `a.out 1`.

EXAMPLE 1 Create a default thread that can serve as a signal catcher/handler with its own signal mask. *(Continued)*

In this example:

- The `sigemptyset(3C)` function initializes a null signal set, new. The `sigaddset(3C)` function packs the signal, `SIGINT`, into that new set.
- Either `pthread_sigmask()` or `thr_sigsetmask()` is used to mask the signal, `SIGINT` (CTRL-C), from the calling thread, which is `main()`. The signal is masked to guarantee that only the new thread will receive this signal.
- `pthread_create()` or `thr_create()` creates the signal-handling thread.
- Using `pthread_join(3C)` or `thr_join(3C)`, `main()` then waits for the termination of that signal-handling thread, whose ID number is `user_threadID`. Then `main()` will `sleep(3C)` for 2 seconds, after which the program terminates.
- The signal-handling thread, `handler`:
  - Assigns the handler `interrupt()` to handle the signal `SIGINT` by the call to `sigaction(2)`.
  - Resets its own signal set to *not block* the signal, `SIGINT`.
  - Sleeps for 8 seconds to allow time for the user to deliver the signal `SIGINT` by pressing the CTRL-C.

```
/* cc thisfile.c -lthread -lpthread */
#define _REENTRANT    /* basic first 3-lines for threads */
#include <pthread.h>
#include <thread.h>

thread_t user_threadID;
sigset_t new;
void *handler( ), interrupt( );

int
main( int argc, char *argv[ ] ){
   test_argv(argv[1]);

   sigemptyset(&new);
   sigaddset(&new, SIGINT);
   switch(*argv[1])  {

     case '0':   /* POSIX */
       pthread_sigmask(SIG_BLOCK, &new, NULL);
       pthread_create(&user_threadID, NULL, handler, argv[1]);
       pthread_join(user_threadID, NULL);
       break;
```

EXAMPLE 1 Create a default thread that can serve as a signal catcher/handler with its own signal mask. *(Continued)*

```
     case '1':   /* Solaris */
       thr_sigsetmask(SIG_BLOCK, &new, NULL);
       thr_create(NULL, 0, handler, argv[1], 0, &user_threadID);
       thr_join(user_threadID, NULL, NULL);
       break;
}  /* switch */

   printf("thread handler, # %d, has exited\n",user_threadID);
       sleep(2);
       printf("main thread, # %d is done\n", thr_self( ));
       return (0)
} /* end main */

struct sigaction act;

void *
handler(char *argv1)
 {
        act.sa_handler = interrupt;
        sigaction(SIGINT, &act, NULL);
        switch(*argv1){
          case '0':     /* POSIX */
            pthread_sigmask(SIG_UNBLOCK, &new, NULL);
            break;
          case '1':   /* Solaris */
            thr_sigsetmask(SIG_UNBLOCK, &new, NULL);
            break;
  }
  printf("\n Press CTRL-C to deliver SIGINT signal to the process\n");
  sleep(8);  /* give user time to hit CTRL-C */
  return (NULL)
}

void
interrupt(int sig)
{
printf("thread %d caught signal %d\n", thr_self( ), sig);
}

void test_argv(char argv1[ ])    {
  if(argv1 == NULL)  {
     printf("use 0 as arg1 to use thr_create( );\n \
     or use 1 as arg1 to use pthread_create( )\n");
     exit(NULL);
```

EXAMPLE 1 Create a default thread that can serve as a signal catcher/handler with its own signal mask. *(Continued)*

```
  }
}
```

In the last example, the `handler` thread served as a signal-handler while also taking care of activity of its own (in this case, sleeping, although it could have been some other activity). A thread could be completely dedicated to signal-handling simply by waiting for the delivery of a selected signal by blocking with `sigwait(2)`. The two subroutines in the previous example, `handler()` and `interrupt()`, could have been replaced with the following routine:

```
void *
handler(void *ignore)
{ int signal;
  printf("thread %d waiting for you to press the CTRL-C keys\n",
          thr_self( ));
  sigwait(&new, &signal);
  printf("thread %d has received the signal %d \n", thr_self( ), signal);
}
/*pthread_create( ) and thr_create( ) would use NULL instead of
  argv[1] for the arg passed to handler( ) */
```

In this routine, one thread is dedicated to catching and handling the signal specified by the set `new`, which allows `main()` and all of its other sub-threads, created *after* `pthread_sigmask()` or `thr_sigsetmask()` masked that signal, to continue uninterrupted. Any use of `sigwait(2)` should be such that all threads block the signals passed to `sigwait(2)` at all times. Only the thread that calls `sigwait()` will get the signals. The call to `sigwait(2)` takes two arguments.

For this type of background dedicated signal-handling routine, a Solaris daemon thread can be used by passing the argument `THR_DAEMON` to `thr_create()`.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe and Async-Signal-Safe

See Also

`sigaction(2)`, `sigprocmask(2)`, `sigwait(2)`, `cond_wait(3C)`, `pthread_cancel(3C)`, `pthread_create(3C)`, `pthread_join(3C)`, `pthread_self(3C)`, `sigaddset(3C)`, `sigemptyset(3C)`, `sigsetops(3C)`, `sleep(3C)`, `attributes(5)`, `cancellation(5)`, `standards(5)`

Notes

It is not possible to block signals that cannot be caught or ignored (see `sigaction(2)`). It is also not possible to block or unblock `SIGCANCEL`, as `SIGCANCEL` is reserved for the implementation of POSIX thread cancellation (see `pthread_cancel(3C)` and `cancellation(5)`). This restriction is quietly enforced by the standard C library.

Using sigwait(2) in a dedicated thread allows asynchronously generated signals to be managed synchronously; however, sigwait(2) should never be used to manage synchronously generated signals.

Synchronously generated signals are exceptions that are generated by a thread and are directed at the thread causing the exception. Since `sigwait()` blocks waiting for signals, the blocking thread cannot receive a synchronously generated signal.

Calling thesigprocmask(2) function will be the same as if `thr_sigsetmask()` or `pthread_sigmask()` has been called. POSIX leaves the semantics of the call to sigprocmask(2) unspecified in a multi-threaded process, so programs that care about POSIX portability should not depend on this semantic.

If a signal is delivered while a thread is waiting on a condition variable, the cond_wait(3C) function will be interrupted and the handler will be executed. The state of the lock protecting the condition variable is undefined while the thread is executing the signal handler.

Signals that are generated synchronously should not be masked. If such a signal is blocked and delivered, the receiving process is killed.

Name thr_stksegment – get thread stack address and size

Synopsis

```
cc –mt [ flag... ] file...[ library... ]
#include <thread.h>
#include <signal.h>

int thr_stksegment(stack_t *ss);
```

Description The `thr_stksegment()` function returns, in its `stack_t` argument, the address and size of the calling thread's stack.

The `stack_t` structure includes the following members:

```
void   *ss_sp
size_t ss_size
int    ss_flags
```

On successful return from `thr_stksegment()`, `ss_sp` contains the high address of the caller's stack and `ss_size` contains the size of the stack in bytes. The `ss_flags` member is always 0. Note that the meaning of `ss_sp` is reversed from other uses of `stack_t` such as `sigaltstack(2)` where `ss_sp` is the low address.

The stack information provided by `thr_stksegment()` is typically used by debuggers, garbage collectors, and similar applications. Most applications should not require such information.

Return Values The `thr_stksegment()` function returns `0` if the thread stack address and size were successfully retrieved. Otherwise, it returns a non-zero error value.

Errors The `thr_stksegment()` function will fail if:

`EAGAIN` The stack information for the thread is not available because the thread's initialization is not yet complete, or the thread is an internal thread.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `sigaltstack(2)`, `thr_create(3C)`, `attributes(5)`

Name thr_suspend, thr_continue – suspend or continue thread execution

Synopsis

```
cc –mt [ flag... ] file...[ library... ]
#include <thread.h>

int thr_suspend(thread_t target_thread);

int thr_continue(thread_t target_thread);
```

Description The `thr_suspend()` function immediately suspends the execution of the thread specified by *target_thread*. On successful return from `thr_suspend()`, the suspended thread is no longer executing. Once a thread is suspended, subsequent calls to `thr_suspend()` have no effect.

The `thr_continue()` function resumes the execution of a suspended thread. Once a suspended thread is continued, subsequent calls to `thr_continue()` have no effect.

A suspended thread will not be awakened by any mechanism other than a call to `thr_continue()`. Signals and the effect of calls to`mutex_unlock(3C)`, `rw_unlock(3C)`, `sema_post(3C)`, `cond_signal(3C)`, and `cond_broadcast(3C)` remain pending until the execution of the thread is resumed by `thr_continue()`.

Return Values If successful, the `thr_suspend()` and `thr_continue()` functions return `0`. Otherwise, a non-zero value is returned to indicate the error.

Errors The `thr_suspend()` and `thr_continue()` functions will fail if:

`ESRCH` The *target_thread* cannot be found in the current process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `thr_create(3C)`, `thr_join(3C)`, `attributes(5)`, `standards(5)`

Warnings The `thr_suspend()` function is extremely difficult to use safely because it suspends the target thread with no concern for the target thread's state. The target thread could be holding locks, waiting for a lock, or waiting on a condition variable when it is unconditionally suspended. The thread will not run until `thr_continue()` is applied, regardless of any calls to `mutex_unlock()`, `cond_signal()`, or `cond_broadcast()` by other threads. Its existence on a sleep queue can interfere with the waking up of other threads that are on the same sleep queue.

The `thr_suspend()` and `thr_continue()` functions should be avoided. Mechanisms that involve the cooperation of the targeted thread, such as mutex locks and condition variables, should be employed instead.

Name thr_yield – yield to another thread

Synopsis

```
cc –mt [ flag... ] file...[ library... ]
#include <thread.h>

void thr_yield(void);
```

Description The `thr_yield()` function causes the current thread to yield its execution in favor of another thread with the same or greater priority.

Return Values The `thr_yield()` function returns nothing and does not set `errno`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `thr_setprio(3C)`, `attributes(5)`, `standards(5)`

Name timeradd, timersub, timerclear, timerisset, timercmp – operations on timeval structures

Synopsis

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

void timeradd(struct timeval *a, struct timeval *b,
     struct timeval *res);

void timerclear(struct timeval *tvp);

int timercmp(struct timeval *a, struct timeval *b, CMP);

int timerisset(struct timeval *tvp);

void timersub(struct timeval *a, struct timeval *b,
     struct timeval *res);
```

Description These macros are provided for manipulating `timeval` structures for use with `gettimeofday(3C)` and `settimeofday(3C)` operands. The structure is defined in `<sys/time.h>` as:

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

The `timeradd()` macro adds the time information stored in *a* to *b* and stores the resulting `timeval` in *res*. The results are simplified such that the value of *res*→*tv_usec* is always less than 1,000,000 (1 second).

The `timersub()` macro subtracts the time information stored in *b* from *a* and stores the resulting `timeval` in *res*.

The `timerclear()` macro initializes *tvp* to midnight (0 hour) January 1st, 1970 (the Epoch).

The `timerisset()` macro returns true if *tvp* is set to any time value other than the Epoch.

The `timercmp()` macro compares *a* to *b* using the form a *CMP b*, where *CMP* is one of `<`, `<=`, `==`, `!=`, `>=`, or `>`.

Usage These macros are not available in function form. All of these macros evaluate their arguments more than once. If parameters passed to these macros are expressions with side effects, the results are undefined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe with Exceptions

See Also gettimeofday(3C), attributes(5)

Name timer_create – create a timer

Synopsis

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

int timer_create(clockid_t clock_id,
     struct sigevent *restrict evp, timer_t *restrict timerid);
```

Description The `timer_create()` function creates a timer using the specified clock, *clock_id*, as the timing base. The `timer_create()` function returns, in the location referenced by *timerid*, a timer ID of type `timer_t` used to identify the timer in timer requests. This timer ID will be unique within the calling process until the timer is deleted. The particular clock, *clock_id*, is defined in `<time.h>`. The timer whose ID is returned will be in a disarmed state upon return from `timer_create()`.

The *evp* argument, if non-null, points to a `sigevent` structure. This structure, allocated by the application, defines the asynchronous notification that will occur when the timer expires (see `signal.h(3HEAD)` for event notification details). If the *evp* argument is `NULL`, the effect is as if the *evp* argument pointed to a `sigevent` structure with the `sigev_notify` member having the value `SIGEV_SIGNAL`, the `sigev_signo` having the value `SIGALARM`, and the `sigev_value` member having the value of the timer ID.

The system defines a set of clocks that can be used as timing bases for per-process timers. The following values for *clock_id* are supported:

`CLOCK_REALTIME`	wall clock
`CLOCK_VIRTUAL`	user CPU usage clock
`CLOCK_PROF`	user and system CPU usage clock
`CLOCK_HIGHRES`	non-adjustable, high-resolution clock

For timers created with a *clock_id* of `CLOCK_HIGHRES`, the system will attempt to use an optimal hardware source. This may include, but is not limited to, per-CPU timer sources. The actual hardware source used is transparent to the user and may change over the lifetime of the timer. For example, if the caller that created the timer were to change its processor binding or its processor set, the system may elect to drive the timer with a hardware source that better reflects the new binding. Timers based on a *clock_id* of `CLOCK_HIGHRES` are ideally suited for interval timers that have minimal jitter tolerence.

Timers are not inherited by a child process across a `fork(2)` and are disarmed and deleted by a call to one of the `exec` functions (see `exec(2)`).

Return Values Upon successful completion, `timer_create()` returns `0` and updates the location referenced by *timerid* to a `timer_t`, which can be passed to the per-process timer calls. If an error occurs, the function returns `−1` and sets `errno` to indicate the error. The value of *timerid* is undefined if an error occurs.

Errors The `timer_create()` function will fail if:

`EAGAIN` The system lacks sufficient signal queuing resources to honor the request, or the calling process has already created all of the timers it is allowed by the system.

`EINVAL` The specified clock ID, *clock_id*, is not defined.

`EPERM` The specified clock ID, *clock_id*, is `CLOCK_HIGHRES` and the {`PRIV_PROC_CLOCK_HIGHRES`} is not asserted in the effective set of the calling process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `exec(2)`, `fork(2)`, `time(2)`, `clock_settime(3C)`, `signal(3C)`, `signal.h(3HEAD)`, `timer_delete(3C)`, `timer_settime(3C)`, `attributes(5)`, `privileges(5)`, `standards(5)`

Name timer_delete – delete a timer

Synopsis

```
#include <time.h>

int timer_delete(timer_t timerid);
```

Description The `timer_delete()` function deletes the specified timer, *timerid*, previously created by the `timer_create(3C)` function. If the timer is armed when `timer_delete()` is called, the behavior will be as if the timer is automatically disarmed before removal. The disposition of pending signals for the deleted timer is unspecified.

Return Values If successful, the function returns `0`. Otherwise, the function returns `−1` and sets `errno` to indicate the error.

Errors The `timer_delete()` function will fail if:

`EINVAL` The timer ID specified by *timerid* is not a valid timer ID.

`ENOSYS` The `timer_delete()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `timer_create(3C)`, `attributes(5)`, `standards(5)`

Name timer_settime, timer_gettime, timer_getoverrun – per-process timers

Synopsis

```
#include <time.h>

int timer_settime(timer_t timerid, int flags,
     const struct itimerspec *restrict value,
     struct itimerspec *restrict ovalue);

int timer_gettime(timer_t timerid, struct itimerspec *value);

int timer_getoverrun(timer_t timerid);
```

Description The `timer_settime()` function sets the time until the next expiration of the timer specified by *timerid* from the `it_value` member of the *value* argument and arm the timer if the `it_value` member of *value* is non-zero. If the specified timer was already armed when `timer_settime()` is called, this call resets the time until next expiration to the *value* specified. If the `it_value` member of *value* is 0, the timer is disarmed. The effect of disarming or resetting a timer on pending expiration notifications is unspecified.

If the flag `TIMER_ABSTIME` is not set in the argument *flags*, `timer_settime()` behaves as if the time until next expiration is set to be equal to the interval specified by the `it_value` member of *value*. That is, the timer expires in `it_value` nanoseconds from when the call is made. If the flag `TIMER_ABSTIME` is set in the argument *flags*, `timer_settime()` behaves as if the time until next expiration is set to be equal to the difference between the absolute time specified by the `it_value` member of *value* and the current value of the clock associated with *timerid*. That is, the timer expires when the clock reaches the value specified by the `it_value` member of *value*. If the specified time has already passed, the function succeeds and the expiration notification is made.

The reload value of the timer is set to the value specified by the `it_interval` member of *value*. When a timer is armed with a non-zero `it_interval`, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer will be rounded up to the larger multiple of the resolution. Quantization error will not cause the timer to expire earlier than the rounded time value.

If the argument *ovalue* is not `NULL`, the function `timer_settime()` stores, in the location referenced by *ovalue*, a value representing the previous amount of time before the timer would have expired or 0 if the timer was disarmed, together with the previous timer reload value. The members of *ovalue* are subject to the resolution of the timer, and they are the same values that would be returned by a `timer_gettime()` call at that point in time.

The `timer_gettime()` function stores the amount of time until the specified timer, *timerid*, expires and the reload value of the timer into the space pointed to by the *value* argument. The `it_value` member of this structure contains the amount of time before the timer expires, or 0 if the timer is disarmed. This value is returned as the interval until timer expiration, even if the timer was armed with absolute time. The `it_interval` member of *value* contains the reload value last set by `timer_settime()`.

Only a single signal will be queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal will be queued, and a timer overrun occurs. When a timer expiration signal is delivered to or accepted by a process, the `timer_getoverrun()` function returns the timer expiration overrun count for the specified timer. The overrun count returned contains the number of extra timer expirations that occurred between the time the signal was generated (queued) and when it was delivered or accepted, up to but not including an implementation-dependent maximum of `DELAYTIMER_MAX`. If the number of such extra expirations is greater than or equal to `DELAYTIMER_MAX`, then the overrun count will be set to `DELAYTIMER_MAX`. The value returned by `timer_getoverrun()` applies to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered for the timer, the meaning of the overrun count returned is undefined.

Return Values If the `timer_settime()` or `timer_gettime()` functions succeed, `0` is returned. If an error occurs for either of these functions, `−1` is returned, and `errno` is set to indicate the error. If the `timer_getoverrun()` function succeeds, it returns the timer expiration overrun count as explained above.

Errors The `timer_settime()`, `timer_gettime()` and `timer_getoverrun()` functions will fail if:

`EINVAL` The *timerid* argument does not correspond to a timer returned by `timer_create(3C)` but not yet deleted by `timer_delete(3C)`.

`ENOSYS` The `timer_settime()`, `timer_gettime()`, and `timer_getoverrun()` functions are not supported by the system. The `timer_settime()` function will fail if:

`EINVAL` A *value* structure specified a nanosecond value less than zero or greater than or equal to 1000 million.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `time.h(3HEAD)`, `clock_settime(3C)`, `timer_create(3C)`, `timer_delete(3C)`, `attributes(5)`, `standards(5)`

Name tmpfile – create a temporary file

Synopsis

```
#include <stdio.h>

FILE *tmpfile(void);
```

Description The `tmpfile()` function creates a temporary file and opens a corresponding stream. The file will automatically be deleted when all references to the file are closed. The file is opened as in `fopen(3C)` for update (`w+`).

The largest value that can be represented correctly in an object of type `off_t` will be established as the offset maximum in the open file description.

Return Values Upon successful completion, `tmpfile()` returns a pointer to the stream of the file that is created. Otherwise, it returns a null pointer and sets `errno` to indicate the error.

Errors The `tmpfile()` function will fail if:

`EINTR` A signal was caught during the execution of `tmpfile()`.

`EMFILE` There are `OPEN_MAX` file descriptors currently open in the calling process.

`ENFILE` The maximum allowable number of files is currently open in the system.

`ENOSPC` The directory or file system which would contain the new file cannot be expanded.

The `tmpfile()` function may fail if:

`EMFILE` There are `FOPEN_MAX` streams currently open in the calling process.

`ENOMEM` Insufficient storage space is available.

Usage The stream refers to a file which is unlinked. If the process is killed in the period between file creation and unlinking, a permanent file may be left behind.

The `tmpfile()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Standard	See `standards(5)`.

See Also `unlink(2)`, `fopen(3C)`, `mkstemp(3C)`, `mktemp(3C)`, `tmpnam(3C)`, `lf64(5)`, `standards(5)`

Name tmpnam, tmpnam_r, tempnam – create a name for a temporary file

Synopsis

```
#include <stdio.h>

char *tmpnam(char *s);

char *tmpnam_r(char *s);

char *tempnam(const char *dir, const char *pfx);
```

Description These functions generate file names that can be used safely for a temporary file.

`tmpnam()` The `tmpnam()` function always generates a file name using the path prefix defined as `P_tmpdir` in the <`stdio.h`> header. On Solaris systems, the default value for `P_tmpdir` is `/var/tmp`. If *s* is `NULL`, `tmpnam()` leaves its result in a thread–specific data area and returns a pointer to that area. The next call to `tmpnam()` by the same thread 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 through inclusion of <`stdio.h`>. The `tmpnam()` function places its result in that array and returns *s*.

`tmpnam_r()` The `tmpnam_r()` function has the same functionality as `tmpnam()` except that if *s* is a null pointer, the function returns `NULL`.

`tempnam()` The `tempnam()` function allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is `NULL` or points to a string that is not a name for an appropriate directory, the path prefix defined as `P_tmpdir` in the <`stdio.h`> header is used. If that directory is not accessible, `/tmp` is used. If, however, the `TMPDIR` environment variable is set in the user's environment, its value is used as the temporary-file directory.

Many applications prefer that temporary files have certain initial character sequences in their names. The *pfx* argument may be `NULL` or point to a string of up to five characters to be used as the initial characters of the temporary-file name.

Upon successful completion, `tempnam()` uses `malloc(3C)` to allocate space for a string, puts the generated pathname in that space, and returns a pointer to it. The pointer is suitable for use in a subsequent call to `free()`. If `tempnam()` cannot return the expected result for any reason (for example, `malloc()` failed), or if none of the above-mentioned attempts to find an appropriate directory was successful, a null pointer is returned and `errno` is set to indicate the error.

Errors The `tempnam()` function will fail if:

`ENOMEM` Insufficient storage space is available.

Usage These functions generate a different file name each time they are called.

Files created using these functions and either `fopen(3C)` or `creat(2)` are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to remove the file when its use is ended.

If called more than TMP_MAX (defined in <stdio.h>) times in a single process, these functions start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or mktemp(3C) and the file names are chosen to render duplication by other means unlikely.

The tmpnam() function is safe to use in multithreaded applications because it employs thread-specific data if it is passed a NULL pointer. However, its use is discouraged. The tempnam() function is safe in multithreaded applications and should be used instead.

When compiling multithreaded applications, the _REENTRANT flag must be defined on the compile line. This flag should be used only with multithreaded applications.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	tmpnam() and tempnam() are Standard.
MT-Level	Safe

See Also

creat(2), unlink(2), fopen(3C), free(3C), malloc(3C), mktemp(3C), mkstemp(3C), tmpfile(3C), attributes(5), standards(5)

Name toascii – translate integer to a 7-bit ASCII character

Synopsis

```
#include <ctype.h>

int toascii(int c);
```

Description The `toascii()` function converts its argument into a 7-bit ASCII character.

Return Values The `toascii()` function returns the value (*c* & `0x7f`).

Errors No errors are returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe

See Also `isascii(3C)`, `attributes(5)`, `standards(5)`

Name _tolower – transliterate upper-case characters to lower-case

Synopsis

```
#include <ctype.h>

int _tolower(int c);
```

Description The `_tolower()` macro is equivalent to `tolower(3C)` except that the argument *c* must be an upper-case letter.

Return Values On successful completion, `_tolower()` returns the lower-case letter corresponding to the argument passed.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe

See Also `isupper(3C)`, `tolower(3C)`, `attributes(5)`, `standards(5)`

Name tolower – transliterate upper-case characters to lower-case

Synopsis

```
#include <ctype.h>

int tolower(int c);
```

Description The `tolower()` function has as a domain a type `int`, the value of which is representable as an `unsigned char` or the value of `EOF`. If the argument has any other value, the argument is returned unchanged. If the argument of `tolower()` represents an upper-case letter, and there exists a corresponding lower-case letter (as defined by character type information in the program locale category `LC_CTYPE`), the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

Return Values On successful completion, `tolower()` returns the lower-case letter corresponding to the argument passed. Otherwise, it returns the argument unchanged.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe

See Also `_tolower(3C)`, `setlocale(3C)`, `attributes(5)`, `standards(5)`

Name _toupper – transliterate lower-case characters to upper-case

Synopsis

```
#include <ctype.h>

int _toupper(int c);
```

Description The `_toupper()` macro is equivalent to `toupper(3C)` except that the argument *c* must be a lower-case letter.

Return Values On successful completion, `_toupper()` returns the upper-case letter corresponding to the argument passed.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe

See Also `islower(3C)`, `toupper(3C)`, `attributes(5)`, `standards(5)`

Name toupper – transliterate lower-case characters to upper-case

Synopsis

```
#include <ctype.h>

int toupper(int c);
```

Description The `toupper()` function has as a domain a type `int`, the value of which is representable as an `unsigned char` or the value of `EOF`. If the argument has any other value, the argument is returned unchanged. If the argument of `toupper()` represents a lower-case letter, and there exists a corresponding upper-case letter (as defined by character type information in the program locale category `LC_CTYPE`), the result is the corresponding upper-case letter. All other arguments in the domain are returned unchanged.

Return Values On successful completion, `toupper()` returns the upper-case letter corresponding to the argument passed.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe

See Also `_toupper(3C)`, `setlocale(3C)`, `attributes(5)`, `standards(5)`

Name towctrans – wide-character mapping

Synopsis

```
#include <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);
```

Description The `towctrans()` function maps the wide character *wc* using the mapping described by *desc*. The current setting of the `LC_CTYPE` category shall be the same as during the call to `wctrans()` that returned the value *desc*.

The function call `towctrans(`*wc*`, wctrans("tolower"))` behaves the same as `towlower(`*wc*`)`.

The function call `towctrans(`*wc*`, wctrans("toupper"))` behaves the same as `towupper(`*wc*`)`.

Return Values The `towctrans()` function returns the mapped value of *wc*, using the mapping described by *desc*; otherwise, it returns *wc* unchanged.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `setlocale(3C)`, `wctrans(3C)`, `attributes(5)`, `standards(5)`

Name towlower – transliterate upper-case wide-character code to lower-case

Synopsis

```
#include <wchar.h>

wint_t towlower(wint_t wc);
```

Description The `towlower()` function has as a domain a type `wint_t`, the value of which must be a character representable as a `wchar_t`, and must be a wide-character code corresponding to a valid character in the current locale or the value of `WEOF`. If the argument has any other value, the argument is returned unchanged. If the argument of `towlower()` represents an upper-case wide-character code, and there exists a corresponding lower-case wide-character code (as defined by character type information in the program locale category `LC_CTYPE`), the result is the corresponding lower-case wide-character code. All other arguments in the domain are returned unchanged.

Return Values On successful completion, `towlower()` returns the lower-case letter corresponding to the argument passed. Otherwise, it returns the argument unchanged.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe

See Also `iswalpha(3C)`, `setlocale(3C)`, `towupper(3C)`, `attributes(5)`, `standards(5)`

Name towupper – transliterate lower-case wide-character code to upper-case

Synopsis

```
#include <wchar.h>

wint_t towupper(wint_t wc);
```

Description The `towupper()` function has as a domain a type `wint_t`, the value of which must be a character representable as a `wchar_t`, and must be a wide-character code corresponding to a valid character in the current locale or the value of `WEOF`. If the argument has any other value, the argument is returned unchanged. If the argument of `towupper()` represents a lower-case wide-character code (as defined by character type information in the program locale category `LC_CTYPE),` the result is the corresponding upper-case wide-character code. All other arguments in the domain are returned unchanged.

Return Values Upon successful completion, `towupper()` returns the upper-case letter corresponding to the argument passed. Otherwise, it returns the argument unchanged.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe

See Also `iswalpha(3C)`, `setlocale(3C)`, `towlower(3C)`, `attributes(5)`, `standards(5)`

Name truncate, ftruncate – set a file to a specified length

Synopsis

```
#include <unistd.h>

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

int ftruncate(int fildes, off_t length);
```

Description The `truncate()` function causes the regular file named by *path* to have a size equal to *length* bytes.

If the file previously was larger than *length*, the extra data is discarded. If the file was previously shorter than length, its size is increased, and the extended area appears as if it were zero-filled.

The application must ensure that the process has write permission for the file.

This function does not modify the file offset for any open file descriptions associated with the file.

The `ftruncate()` function causes the regular file referenced by *fildes* to be truncated to *length*. If the size of the file previously exceeded *length*, the extra data is no longer available to reads on the file. If the file previously was smaller than this size, `ftruncate()` increases the size of the file with the extended area appearing as if it were zero-filled. The value of the seek pointer is not modified by a call to `ftruncate()`.

The `ftruncate()` function works only with regular files and shared memory. If *fildes* refers to a shared memory object, `ftruncate()` sets the size of the shared memory object to *length*. If *fildes* refers to a directory or is not a valid file descriptor open for writing, `ftruncate()` fails.

If the effect of `ftruncate()` is to decrease the size of a shared memory object or memory mapped file and whole pages beyond the new end were previously mapped, then the whole pages beyond the new end shall be discarded.

If the effect of `ftruncate()` is to increase the size of a shared memory object, it is unspecified if the contents of any mapped pages between the old end-of-file and the new are flushed to the underlying object.

These functions do not modify the file offset for any open file descriptions associated with the file. On successful completion, if the file size is changed, these functions will mark for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the `S_ISUID` and `S_ISGID` bits of the file mode are left unchanged.

If the request would cause the file size to exceed the soft file size limit for the process, the request will fail and a `SIGXFSZ` signal will be generated for the process.

Return Values Upon successful completion, `ftruncate()` and `truncate()` return `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `ftruncate()` and `truncate()` functions will fail if:

`EINTR`	A signal was caught during execution.
`EINVAL`	The *length* argument was less than 0.
`EFBIG` or `EINVAL`	The *length* argument was greater than the maximum file size.
`EIO`	An I/O error occurred while reading from or writing to a file system.
`EROFS`	The named file resides on a read-only file system.

The `truncate()` function will fail if:

`EACCES`	A component of the path prefix denies search permission, or write permission is denied on the file.
`EFAULT`	The *path* argument points outside the process' allocated address space.
`EINVAL`	The *path* argument is not an ordinary file.
`EISDIR`	The named file is a directory.
`ELOOP`	Too many symbolic links were encountered in resolving *path*.
`EMFILE`	The maximum number of file descriptors available to the process has been reached.
`ENAMETOOLONG`	The length of the specified pathname exceeds {`PATH_MAX`} bytes, or the length of a component of the pathname exceeds {`NAME_MAX`} bytes.
`ENOENT`	A component of *path* does not name an existing file or *path* is an empty string.
`ENFILE`	Additional space could not be allocated for the system file table.
`ENOTDIR`	A component of the path prefix of *path* is not a directory.
`ENOLINK`	The *path* argument points to a remote machine and the link to that machine is no longer active.

The `ftruncate()` function will fail if:

`EAGAIN`	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see `chmod(2)`).
`EBADF` or `EINVAL`	The *fildes* argument is not a file descriptor open for writing.
`EFBIG`	The file is a regular file and *length* is greater than the offset maximum established in the open file description associated with *fildes*.
`EINVAL`	The *fildes* argument references a file that was opened without write permission.

EINVAL The *fildes* argument does not correspond to an ordinary file.

ENOLINK The *fildes* argument points to a remote machine and the link to that machine is no longer active.

The `truncate()` function may fail if:

ENAMETOOLONG Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

Usage The `truncate()` and `ftruncate()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `chmod(2)`, `fcntl(2)`, `open(2)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Name tsearch, tfind, tdelete, twalk – manage binary search trees

Synopsis

```
#include <search.h>

void *tsearch(const void *key, void **rootp,
     int (*compar)(const void *, const void *));

void *tfind(const void *key, void * const *rootp,
     int (*compar)(const void *, const void *));

void *tdelete(const void *restrict key, void **restrict rootp,
     int (*compar)(const void *, const void *));

void twalk(const void *root, void(*action) (void *, VISIT, int));
```

Description The `tsearch()`, `tfind()`, `tdelete()`, and `twalk()` functions are routines for manipulating binary search trees. They are generalized from *Knuth (6.2.2) Algorithms T and D.* All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The `tsearch()` function is used to build and access the tree. The *key* argument is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to **key* (the value pointed to by *key*), a pointer to this found datum is returned. Otherwise, **key* is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. The *rootp* argument points to a variable that points to the root of the tree. A null value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like `tsearch()`, `tfind()` will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, `tfind()` will return a null pointer. The arguments for `tfind()` are the same as for `tsearch()`.

The `tdelete()` function deletes a node from a binary search tree. The arguments are the same as for `tsearch()`. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. `tdelete()` returns a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The `twalk()` function traverses a binary search tree. The *root* argument is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type

```
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

(defined in <search.h>), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Return Values If the node is found, both `tsearch()` and `tfind()` return a pointer to it. If not, `tfind()` returns a null pointer, and `tsearch()` returns a pointer to the inserted item.

A null pointer is returned by `tsearch()` if there is not enough space available to create a new node.

A null pointer is returned by `tsearch()`, `tfind()` and `tdelete()` if *rootp* is a null pointer on entry.

The `tdelete()` function returns a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The `twalk()` function returns no value.

Errors No errors are defined.

Usage The *root* argument to `twalk()` is one level of indirection less than the *rootp* arguments to `tsearch()` and `tdelete()`.

There are two nomenclatures used to refer to the order in which tree nodes are visited. `tsearch()` uses preorder, postorder and endorder to refer respectively to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

If the calling function alters the pointer to the root, the results are unpredictable.

These functions safely allows concurrent access by multiple threads to disjoint data, such as overlapping subtrees or tables.

Examples **EXAMPLE 1** A sample program of using `tsearch()` function.

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <string.h>
#include <stdio.h>
#include <search.h>
struct node {
```

EXAMPLE 1 A sample program of using `tsearch()` function. *(Continued)*

```
        char *string;
        int length;
};
char string_space[10000];
struct node nodes[500];
void *root = NULL;

int node_compare(const void *node1, const void *node2) {
        return strcmp(((const struct node *) node1)->string,
                      ((const struct node *) node2)->string);
}

void print_node(const void *node, VISIT order, int level) {
        if (order == preorder || order == leaf) {
                printf("length=%d, string=%20s\n",
                (*(struct node **)node)->length,
                (*(struct node **)node)->string);
        }
}

main()
{
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        int i = 0;

        while (gets(strptr) != NULL && i++ < 500) {
                nodeptr->string = strptr;
                nodeptr->length = strlen(strptr);
                (void) tsearch((void *)nodeptr,
                        &root, node_compare);
                strptr += nodeptr->length + 1;
                nodeptr++;
        }
        twalk(root, print_node);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also bsearch(3C), hsearch(3C), lsearch(3C), attributes(5), standards(5)

Name ttyname, ttyname_r – find pathname of a terminal

Synopsis

```
#include <unistd.h>

char *ttyname(int fildes);

char *ttyname_r(int fildes, char *name, int namelen);
```

Standard conforming

```
cc [ flag...] file ... -D_POSIX_PTHREAD_SEMANTICS [ library ... ]

int ttyname_r(int fildes, char *name, size_t namesize);
```

Description The `ttyname()` function returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*. The return value points to thread–specific data whose content is overwritten by each call from the same thread.

The `ttyname_r()` function has the same functionality as `ttyname()` except that the caller must supply a buffer *name* with length *namelen* to store the result; this buffer must be at least `_POSIX_PATH_MAX` in size (defined in <limits.h>). The standard-conforming version (see `standards(5)`) of `ttyname_r()` takes a *namesize* parameter of type `size_t`.

Return Values Upon successful completion, `ttyname()` and `ttyname_r()` return a pointer to a string. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

The standard-conforming `ttyname_r()` returns 0 if successful or the error number upon failure.

Errors The `ttyname()` and `ttyname_r()` functions may fail if:

`EBADF` The *fildes* argument is not a valid file descriptor. This condition is reported.

`ENOTTY` The *fildes* argument does not refer to a terminal device. This condition is reported.

The `ttyname_r()` function may fail if:

`ERANGE` The value of *namesize* is smaller than the length of the string to be returned including the terminating null character.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also Intro(3), gettext(3C), setlocale(3C), attributes(5), standards(5)

Notes When compiling multithreaded applications, see Intro(3), *Notes On Multithreaded Applications*.

Messages printed from this function are in the native language specified by the LC_MESSAGES locale category. See setlocale(3C).

The return value of ttyname() points to thread–specific data whose content is overwritten by each call from the same thread. This function is safe to use in multithreaded applications, but its use is discouraged. The ttyname_r() function should used instead.

Solaris 2.4 and earlier releases provided definitions of the ttyname_r() interface as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface as described above. Support for the Draft 6 interface is provided for compatibility only and might not be supported in future releases. New applications and libraries should use the standard-conforming interface.

Name ttyslot – find the slot of the current user in the user accounting database

Synopsis

```
#include <stdlib.h>

int ttyslot(void);
```

Description The `ttyslot()` function returns the index of the current user's entry in the user accounting database, `/var/adm/utmpx`. The current user's entry is an entry for which the `utline` member matches the name of a terminal device associated with any of the process's file descriptors 0, 1 or 2. The index is an ordinal number representing the record number in the database of the current user's entry. The first entry in the database is represented by the return value 0.

Return Values Upon successful completion, `ttyslot()` returns the index of the current user's entry in the user accounting database. If an error was encountered while searching for the terminal name or if none of the above file descriptors are associated with a terminal device, −1 is returned.

Files `/var/adm/utmpx` user access and accounting information

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `getutent(3C)`, `ttyname(3C)`, `utmpx(4)`, `attributes(5)`

Name u8_strcmp – UTF-8 string comparison function

Synopsis

```
#include <sys/u8_textprep.h>

int u8_strcmp(const char *s1, const char *s2, size_t n,
     int flag, size_t version, int *errnum);
```

Parameters

s1, *s2* Pointers to null-terminated UTF-8 strings

n The maximum number of bytes to be compared. If 0, the comparison is performed until either or both of the strings are examined to the string terminating null byte.

flag The possible comparison options constructed by a bit-wise-inclusive-OR of the following values:

`U8_STRCMP_CS`
Perform case-sensitive string comparison. This is the default.

`U8_STRCMP_CI_UPPER`
Perform case-insensitive string comparison based on Unicode upper case converted results of *s1* and *s2*.

`U8_STRCMP_CI_LOWER`
Perform case-insensitive string comparison based on Unicode lower case converted results of *s1* and *s2*.

`U8_STRCMP_NFD`
Perform string comparison after *s1* and *s2* have been normalized by using Unicode Normalization Form D.

`U8_STRCMP_NFC`
Perform string comparison after *s1* and *s2* have been normalized by using Unicode Normalization Form C.

`U8_STRCMP_NFKD`
Perform string comparison after *s1* and *s2* have been normalized by using Unicode Normalization Form KD.

`U8_STRCMP_NFKC`
Perform string comparison after *s1* and *s2* have been normalized by using Unicode Normalization Form KC.

Only one case-sensitive or case-insensitive option is allowed. Only one Unicode Normalization option is allowed.

version The version of Unicode data that should be used during comparison. The following values are supported:

`U8_UNICODE_320`
Use Unicode 3.2.0 data during comparison.

`U8_UNICODE_500`
Use Unicode 5.0.0 data during comparison.

`U8_UNICODE_LATEST`
Use the latest Unicode version data available, which is Unicode 5.0.0.

errnum A non-zero value indicates that an error has occurred during comparison. The following values are supported:

`EBADF` The specified option values are conflicting and cannot be supported.

`EILSEQ` There was an illegal character at *s1*, *s2*, or both.

`EINVAL` There was an incomplete character at *s1*, *s2*, or both.

`ERANGE` The specified Unicode version value is not supported.

Description The `u8_stcmp()` function internally processes UTF-8 strings pointed to by *s1* and *s2* based on the corresponding version of the Unicode Standard and other input arguments and compares the result strings in byte-by-byte, machine ordering.

When multiple comparison options are specified, Unicode Normalization is performed after case-sensitive or case-insensitive processing is performed.

Return Values The `u8_strcmp()` function returns an integer greater than, equal to, or less than 0 if the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*, respectively.

When `u8_strcmp()` detects an illegal or incomplete character, such character causes the function to set *errnum* to indicate the error. Afterward, the comparison is still performed on the resultant strings and a value based on byte-by-byte comparison is always returned.

Examples **EXAMPLE 1** Perform simple default string comparison.

```
#include <sys/u8_textprep.h>

int
docmp_default(const char *u1, const char *u2) {
    int result;
    int errnum;

    result = u8_strcmp(u1, u2, 0, 0, U8_UNICODE_LATEST, &errnum);
    if (errnum == EILSEQ)
        return (-1);
    if (errnum == EINVAL)
        return (-2);
    if (errnum == EBADF)
        return (-3);
    if (errnum == ERANGE)
        return (-4);
```

EXAMPLE 2 Perform upper case based case-insensitive comparison with Unicode 3.2.0 date.

```
#include <sys/u8_textprep.h>

int
docmp_caseinsensitive_u320(const char *u1, const char *u2) {
    int result;
    int errnum;

    result = u8_strcmp(u1, u2, 0, U8_STRCMP_CI_UPPER,
        U8_UNICODE_320, &errnum);
    if (errnum == EILSEQ)
        return (-1);
    if (errnum == EINVAL)
        return (-2);
    if (errnum == EBADF)
        return (-3);
    if (errnum == ERANGE)
        return (-4);

    return (result);
}
```

EXAMPLE 3 Perform Unicode Normalization Form D.

Perform Unicode Normalization Form D and upper case based case-insensitive comparison with Unicode 3.2.0 date.

```
#include <sys/u8_textprep.h>

int
docmp_nfd_caseinsensitive_u320(const char *u1, const char *u2) {
    int result;
    int errnum;

    result = u8_strcmp(u1, u2, 0,
        (U8_STRCMP_NFD|U8_STRCMP_CI_UPPER), U8_UNICODE_320,
        &errnum);
    if (errnum == EILSEQ)
        return (-1);
    if (errnum == EINVAL)
        return (-2);
    if (errnum == EBADF)
        return (-3);
    if (errnum == ERANGE)
        return (-4);

    return (result);
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also u8_textprep_str(3C), u8_validate(3C), attributes(5), u8_strcmp(9F), u8_textprep_str(9F), u8_validate(9F)

The Unicode Standard (http://www.unicode.org)

Name u8_textprep_str – string-based UTF-8 text preparation function

Synopsis

```
#include <sys/u8_textprep.h>

size_t u8_textprep_str(char *inarray, size_t *inlen,
     char *outarray, size_t *outlen, int flag,
     size_t unicode_version, int *errnum);
```

Parameters

inarray A pointer to a byte array containing a sequence of UTF-8 character bytes to be prepared.

inlen As input argument, the number of bytes to be prepared in *inarray*. As output argument, the number of bytes in *inarray* still not consumed.

outarray A pointer to a byte array where prepared UTF-8 character bytes can be saved.

outlen As input argument, the number of available bytes at *outarray* where prepared character bytes can be saved. As output argument, after the conversion, the number of bytes still available at *outarray*.

flag The possible preparation options constructed by a bitwise-inclusive-OR of the following values:

`U8_TEXTPREP_IGNORE_NULL`
Normally `u8_textprep_str()` stops the preparation if it encounters null byte even if the current *inlen* is pointing to a value bigger than zero.

With this option, null byte does not stop the preparation and the preparation continues until *inlen* specified amount of *inarray* bytes are all consumed for preparation or an error happened.

`U8_TEXTPREP_IGNORE_INVALID`
Normally `u8_textprep_str()` stops the preparation if it encounters illegal or incomplete characters with corresponding *errnum* values.

When this option is set, `u8_textprep_str()` does not stop the preparation and instead treats such characters as no need to do any preparation.

`U8_TEXTPREP_TOUPPER`
Map lowercase characters to uppercase characters if applicable.

`U8_TEXTPREP_TOLOWER`
Map uppercase characters to lowercase characters if applicable.

`U8_TEXTPREP_NFD`
Apply Unicode Normalization Form D.

`U8_TEXTPREP_NFC`
Apply Unicode Normalization Form C.

`U8_TEXTPREP_NFKD`
Apply Unicode Normalization Form KD.

`U8_TEXTPREP_NFKC`
Apply Unicode Normalization Form KC.

Only one case folding option is allowed. Only one Unicode Normalization option is allowed.

When a case folding option and a Unicode Normalization option are specified together, UTF-8 text preparation is done by doing case folding first and then Unicode Normalization.

If no option is specified, no processing occurs except the simple copying of bytes from input to output.

unicode_version The version of Unicode data that should be used during UTF-8 text preparation. The following values are supported:

`U8_UNICODE_320`
Use Unicode 3.2.0 data during comparison.

`U8_UNICODE_500`
Use Unicode 5.0.0 data during comparison.

`U8_UNICODE_LATEST`
Use the latest Unicode version data available which is Unicode 5.0.0 currently.

errnum The error value when preparation is not completed or fails. The following values are supported:

`E2BIG`	Text preparation stopped due to lack of space in the output array.
`EBADF`	Specified option values are conflicting and cannot be supported.
`EILSEQ`	Text preparation stopped due to an input byte that does not belong to UTF-8.
`EINVAL`	Text preparation stopped due to an incomplete UTF-8 character at the end of the input array.
`ERANGE`	The specified Unicode version value is not a supported version.

Description The `u8_textprep_str()` function prepares the sequence of UTF-8 characters in the array specified by *inarray* into a sequence of corresponding UTF-8 characters prepared in the array specified by *outarray*. The *inarray* argument points to a character byte array to the first character in the input array and *inlen* indicates the number of bytes to the end of the array to be converted. The *outarray* argument points to a character byte array to the first available byte in the output array and *outlen* indicates the number of the available bytes to the end of the array. Unless *flag* is `U8_TEXTPREP_IGNORE_NULL`, `u8_textprep_str()` normally stops when it encounters a null byte from the input array regardless of the current *inlen* value.

If *flag* is `U8_TEXTPREP_IGNORE_INVALID` and a sequence of input bytes does not form a valid UTF-8 character, preparation stops after the previous successfully prepared character. If *flag* is `U8_TEXTPREP_IGNORE_INVALID` and the input array ends with an incomplete UTF-8 character, preparation stops after the previous successfully prepared bytes. If the output array is not large enough to hold the entire prepared text, preparation stops just prior to the input bytes that would cause the output array to overflow. The value pointed to by *inlen* is decremented to reflect the number of bytes still not prepared in the input array. The value pointed to by *outlen* is decremented to reflect the number of bytes still available in the output array.

Return Values The `u8_textprep_str()` function updates the values pointed to by *inlen* and *outlen* arguments to reflect the extent of the preparation. When `U8_TEXTPREP_IGNORE_INVALID` is specified, `u8_textprep_str()` returns the number of illegal or incomplete characters found during the text preparation. When `U8_TEXTPREP_IGNORE_INVALID` is not specified and the text preparation is entirely successful, the function returns 0. If the entire string in the input array is prepared, the value pointed to by *inlen* will be 0. If the text preparation is stopped due to any conditions mentioned above, the value pointed to by *inlen* will be non-zero and *errnum* is set to indicate the error. If such and any other error occurs, `u8_textprep_str()` returns (`size_t`)-1 and sets *errnum* to indicate the error.

Examples **EXAMPLE 1** Simple UTF-8 text preparation

```
#include <sys/u8_textprep.h>
.
.
.
size_t ret;
char ib[MAXPATHLEN];
char ob[MAXPATHLEN];
size_t il, ol;
int err;
.
.
.
/*
 * We got a UTF-8 pathname from somewhere.
 *
 * Calculate the length of input string including the terminating
```

EXAMPLE 1 Simple UTF-8 text preparation *(Continued)*

```
 * NULL byte and prepare other arguments.
 */
(void) strlcpy(ib, pathname, MAXPATHLEN);
il = strlen(ib) + 1;
ol = MAXPATHLEN;

/*
 * Do toupper case folding, apply Unicode Normalization Form D,
 * ignore NULL byte, and ignore any illegal/incomplete characters.
 */
ret = u8_textprep_str(ib, &il, ob, &ol,
    (U8_TEXTPREP_IGNORE_NULL|U8_TEXTPREP_IGNORE_INVALID|
    U8_TEXTPREP_TOUPPER|U8_TEXTPREP_NFD), U8_UNICODE_LATEST, &err);
if (ret == (size_t)-1) {
    if (err == E2BIG)
        return (-1);
    if (err == EBADF)
        return (-2);
    if (err == ERANGE)
        return (-3);
    return (-4);
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

u8_strcmp(3C), u8_validate(3C), attributes(5), u8_strcmp(9F), u8_textprep_str(9F), u8_validate(9F)

The Unicode Standard (http://www.unicode.org)

Notes

After the text preparation, the number of prepared UTF-8 characters and the total number bytes may decrease or increase when you compare the numbers with the input buffer.

Case conversions are performed using Unicode data of the corresponding version. There are no locale-specific case conversions that can be performed.

Name u8_validate – validate UTF-8 characters and calculate the byte length

Synopsis

```
#include <sys/u8_textprep.h>

int u8_validate(char *u8str, size_t n, char **list, int flag,
     int *errnum);
```

Parameters

u8str The UTF-8 string to be validated.

n The maximum number of bytes in *u8str* that can be examined and validated.

list A list of null-terminated character strings in UTF-8 that must be additionally checked against as invalid characters. The last string in *list* must be null to indicate there is no further string.

flag Possible validation options constructed by a bitwise-inclusive-OR of the following values:

`U8_VALIDATE_ENTIRE`
By default, `u8_validate()` looks at the first character or up to *n* bytes, whichever is smaller in terms of the number of bytes to be consumed, and returns with the result.

When this option is used, `u8_validate()` will check up to *n* bytes from *u8str* and possibly more than a character before returning the result.

`U8_VALIDATE_CHECK_ADDITIONAL`
By default, `u8_validate()` does not use list supplied.

When this option is supplied with a list of character strings, `u8_validate()` additionally validates *u8str* against the character strings supplied with *list* and returns EBADF in *errnum* if *u8str* has any one of the character strings in *list*.

`U8_VALIDATE_UCS2_RANGE`
By default, `u8_validate()` uses the entire Unicode coding space of U+0000 to U+10FFFF.

When this option is specified, the valid Unicode coding space is smaller to U+0000 to U+FFFF.

errnum An error occurred during validation. The following values are supported:

`EBADF` Validation failed because list-specified characters were found in the string pointed to by *u8str*.

`EILSEQ` Validation failed because an illegal byte was found in the string pointed to by *u8str*.

`EINVAL` Validation failed because an incomplete byte was found in the string pointed to by *u8str*.

ERANGE Validation failed because character bytes were encountered that are outside the range of the Unicode coding space.

Description The `u8_validate()` function validates *u8str* in UTF-8 and determines the number of bytes constituting the character(s) pointed to by *u8str*.

Return Values If *u8str* is a null pointer, `u8_validate()` returns 0. Otherwise, `u8_validate()` returns either the number of bytes that constitute the characters if the next *n* or fewer bytes form valid characters, or -1 if there is an validation failure, in which case it may set *errnum* to indicate the error.

Examples **EXAMPLE 1** Determine the length of the first UTF-8 character.

```
#include <sys/u8_textprep.h>

char u8[MAXPATHLEN];
int errnum;
.
.
.
len = u8_validate(u8, 4, (char **)NULL, 0, &errnum);
if (len == -1) {
    switch (errnum) {
        case EILSEQ:
        case EINVAL:
            return (MYFS4_ERR_INVAL);
        case EBADF:
            return (MYFS4_ERR_BADNAME);
        case ERANGE:
            return (MYFS4_ERR_BADCHAR);
        default:
            return (-10);
    }
}
```

EXAMPLE 2 Check if there are any invalid characters in the entire string.

```
#include <sys/u8_textprep.h>

char u8[MAXPATHLEN];
int n;
int errnum;
.
.
.
n = strlen(u8);
len = u8_validate(u8, n, (char **)NULL, U8_VALIDATE_ENTIRE, &errnum);
if (len == -1) {
```

EXAMPLE 2 Check if there are any invalid characters in the entire string. *(Continued)*

```
    switch (errnum) {
        case EILSEQ:
        case EINVAL:
            return (MYFS4_ERR_INVAL);
        case EBADF:
            return (MYFS4_ERR_BADNAME);
        case ERANGE:
            return (MYFS4_ERR_BADCHAR);
        default:
            return (-10);
    }
}
```

EXAMPLE 3 Check if there is any invalid character, including prohibited characters, in the entire string.

```
#include <sys/u8_textprep.h>

char u8[MAXPATHLEN];
int n;
int errnum;
char *prohibited[4] = {
    ".", "..", "\\", NULL
};
.
.
.
n = strlen(u8);
len = u8_validate(u8, n, prohibited,
    (U8_VALIDATE_ENTIRE|U8_VALIDATE_CHECK_ADDITIONAL), &errnum);
if (len == -1) {
    switch (errnum) {
        case EILSEQ:
        case EINVAL:
            return (MYFS4_ERR_INVAL);
        case EBADF:
            return (MYFS4_ERR_BADNAME);
        case ERANGE:
            return (MYFS4_ERR_BADCHAR);
        default:
            return (-10);
    }
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also u8_strcmp(3C), u8_textprep_str(3C), attributes(5), u8_strcmp(9F), u8_textprep_str(9F), u8_validate(9F)

The Unicode Standard (http://www.unicode.org)

Name ualarm – schedule signal after interval in microseconds

Synopsis

```
#include <unistd.h>

useconds_t ualarm(useconds_t useconds, useconds_t interval);
```

Description The `ualarm()` function causes the `SIGALRM` signal to be generated for the calling process after the number of real-time microseconds specified by the *useconds* argument has elapsed. When the *interval* argument is non-zero, repeated timeout notification occurs with a period in microseconds specified by the *interval* argument. If the notification signal, `SIGALRM`, is not caught or ignored, the calling process is terminated.

Because of scheduling delays, resumption of execution when the signal is caught may be delayed an arbitrary amount of time.

Interactions between `ualarm()` and either `alarm(2)` or `sleep(3C)` are unspecified.

Return Values The `ualarm()` function returns the number of microseconds remaining from the previous `ualarm()` call. If no timeouts are pending or if `ualarm()` has not previously been called, `ualarm()` returns `0`.

Errors No errors are defined.

Usage The `ualarm()` function is a simplified interface to `setitimer(2)`, and uses the `ITIMER_REAL` interval timer.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Standard	See `standards(5)`.

See Also `alarm(2)`, `setitimer(2)`, `sighold(3C)`, `signal(3C)`, `sleep(3C)`, `usleep(3C)`, `attributes(5)`, `standards(5)`

Name uconv_u16tou32, uconv_u16tou8, uconv_u32tou16, uconv_u32tou8, uconv_u8tou16, uconv_u8tou32 – Unicode encoding conversion functions

Synopsis

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/u8_textprep.h>

int uconv_u16tou32(const uint16_t *utf16str, size_t *utf16len,
     uint32_t *utf32str, size_t *utf32len, int flag);

int uconv_u16tou8(const uint16_t *utf16str, size_t *utf16len,
     uchar_t *utf8str, size_t *utf8len, int flag);

int uconv_u32tou16(const uint32_t *utf32str, size_t *utf32len,
     uint16_t *utf16str, size_t *utf16len, int flag);

int uconv_u32tou8(const uint32_t *utf32str, size_t *utf32len,
     uchar_t *utf8str, size_t *utf8len, int flag);

int uconv_u8tou16(const uchar_t *utf8str, size_t *utf8len,
     uint16_t *utf16str, size_t *utf16len, int flag);

int uconv_u8tou32(const uchar_t *utf8str, size_t *utf8len,
     uint32_t *utf32str, size_t *utf32len, int flag);
```

Parameters

utf16str A pointer to a UTF-16 character string.

utf16len As an input parameter, the number of 16-bit unsigned integers in *utf16str* as UTF-16 characters to be converted or saved.

As an output parameter, the number of 16-bit unsigned integers in *utf16str* consumed or saved during conversion.

utf32str A pointer to a UTF-32 character string.

utf32len As an input parameter, the number of 32-bit unsigned integers in *utf32str* as UTF-32 characters to be converted or saved.

As an output parameter, the number of 32-bit unsigned integers in *utf32str* consumed or saved during conversion.

utf8str A pointer to a UTF-8 character string.

utf8len As an input parameter, the number of bytes in *utf8str* as UTF-8 characters to be converted or saved.

As an output parameter, the number of bytes in *utf8str* consumed or saved during conversion.

flag The possible conversion options that are constructed by a bitwise-inclusive-OR of the following values:

`UCONV_IN_BIG_ENDIAN`
The input parameter is in big endian byte ordering.

`UCONV_OUT_BIG_ENDIAN`
The output parameter should be in big endian byte ordering.

`UCONV_IN_SYSTEM_ENDIAN`
The input parameter is in the default byte ordering of the current system.

`UCONV_OUT_SYSTEM_ENDIAN`
The output parameter should be in the default byte ordering of the current system.

`UCONV_IN_LITTLE_ENDIAN`
The input parameter is in little endian byte ordering.

`UCONV_OUT_LITTLE_ENDIAN`
The output parameter should be in little endian byte ordering.

`UCONV_IGNORE_NULL`
The null or `U+0000` character should not stop the conversion.

`UCONV_IN_ACCEPT_BOM`
If the Byte Order Mark (BOM, `U+FEFF`) character exists as the first character of the input parameter, interpret it as the BOM character.

`UCONV_OUT_EMIT_BOM`
Start the output parameter with Byte Order Mark (BOM, `U+FEFF`) character to indicate the byte ordering if the output parameter is in `UTF-16` or `UTF-32`.

Description The `uconv_u16tou32()` function reads the given *utf16str* in `UTF-16` until `U+0000` (zero) in *utf16str* is encountered as a character or until the number of 16-bit unsigned integers specified in *utf16len* is read. The `UTF-16` characters that are read are converted into `UTF-32` and the result is saved at *utf32str*. After the successful conversion, *utf32len* contains the number of 32-bit unsigned integers saved at *utf32str* as `UTF-32` characters.

The `uconv_u16tou8()` function reads the given *utf16str* in `UTF-16` until `U+0000` (zero) in *utf16str* is encountered as a character or until the number of 16-bit unsigned integers specified in *utf16len* is read. The `UTF-16` characters that are read are converted into `UTF-8` and the result is saved at *utf8str*. After the successful conversion, *utf8len* contains the number of bytes saved at *utf8str* as `UTF-8` characters.

The `uconv_u32tou16()` function reads the given *utf32str* in `UTF-32` until `U+0000` (zero) in *utf32str* is encountered as a character or until the number of 32-bit unsigned integers specified in *utf32len* is read. The `UTF-32` characters that are read are converted into `UTF-16` and the result is saved at *utf16str*. After the successful conversion, *utf16len* contains the number of 16-bit unsigned integers saved at *utf16str* as `UTF-16` characters.

The `uconv_u32tou8()` function reads the given *utf32str* in `UTF-32` until `U+0000` (zero) in *utf32str* is encountered as a character or until the number of 32-bit unsigned integers specified

in *utf32len* is read. The UTF-32 characters that are read are converted into UTF-8 and the result is saved at *utf8str*. After the successful conversion, *utf8len* contains the number of bytes saved at *utf8str* as UTF-8 characters.

The uconv_u8tou16() function reads the given *utf8str* in UTF-8 until the null ('\0') byte in *utf8str* is encountered or until the number of bytes specified in *utf8len* is read. The UTF-8 characters that are read are converted into UTF-16 and the result is saved at *utf16str*. After the successful conversion, *utf16len* contains the number of 16-bit unsigned integers saved at *utf16str* as UTF-16 characters.

The uconv_u8tou32() function reads the given *utf8str* in UTF-8 until the null ('\0') byte in *utf8str* is encountered or until the number of bytes specified in *utf8len* is read. The UTF-8 characters that are read are converted into UTF-32 and the result is saved at *utf32str*. After the successful conversion, *utf32len* contains the number of 32-bit unsigned integers saved at *utf32str* as UTF-32 characters.

During the conversion, the input and the output parameters are treated with byte orderings specified in the *flag* parameter. When not specified, the default byte ordering of the system is used. The byte ordering *flag* value that is specified for UTF-8 is ignored.

When UCONV_IN_ACCEPT_BOM is specified as the *flag* and the first character of the string pointed to by the input parameter is the BOM character, the value of the BOM character dictates the byte ordering of the subsequent characters in the string pointed to by the input parameter, regardless of the supplied input parameter byte ordering option *flag* values. If the UCONV_IN_ACCEPT_BOM is not specified, the BOM as the first character is treated as a regular Unicode character: Zero Width No Break Space (ZWNBSP) character.

When UCONV_IGNORE_NULL is specified, regardless of whether the input parameter contains U+0000 or null byte, the conversion continues until the specified number of input parameter elements at *utf16len*, *utf32len*, or *utf8len* are entirely consumed during the conversion.

As output parameters, *utf16len*, *utf32len*, and *utf8len* are not changed if conversion fails for any reason.

Return Values Upon successful conversion, the functions return 0. Upon failure, the functions return one of the following errno values:

EILSEQ The conversion detected an illegal or out of bound character value in the input parameter.

E2BIG The conversion cannot finish because the size specified in the output parameter is too small.

EINVAL The conversion stops due to an incomplete character at the end of the input string.

EBADF Conflicting byte-ordering option *flag* values are detected.

Examples

EXAMPLE 1 Convert a UTF-16 string in little-endian byte ordering into UTF-8 string.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/u8_textprep.h>
.
.
.
uint16_t u16s[MAXNAMELEN + 1];
uchar_t u8s[MAXNAMELEN + 1];
size_t u16len, u8len;
int ret;
.
.
.
u16len = u8len = MAXNAMELEN;
ret = uconv_u16tou8(u16s, &u16len, u8s, &u8len,
    UCONV_IN_LITTLE_ENDIAN);
if (ret != 0) {
     /* Conversion error occurred. */
     return (ret);
}
.
.
.
```

EXAMPLE 2 Convert a UTF-32 string in big endian byte ordering into little endian UTF-16.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/u8_textprep.h>
.
.
.
/*
  * An UTF-32 character can be mapped to an UTF-16 character with
  * two 16-bit integer entities as a "surrogate pair."
  */
uint32_t u32s[101];
uint16_t u16s[101];
int ret;
size_t u32len, u16len;
.
.
.
u32len = u16len = 100;
ret = uconv_u32tou16(u32s, &u32len, u16s, &u16len,
    UCONV_IN_BIG_ENDIAN | UCONV_OUT_LITTLE_ENDIAN);
```

EXAMPLE 2 Convert a UTF-32 string in big endian byte ordering into little endian UTF-16. *(Continued)*

```
if (ret == 0) {
     return (0);
} else if (ret == E2BIG) {
     /* Use bigger output parameter and try just one more time. */
     uint16_t u16s2[201];

     u16len = 200;
     ret = uconv_u32tou16(u32s, &u32len, u16s2, &u16len,
        UCONV_IN_BIG_ENDIAN | UCONV_OUT_LITTLE_ENDIAN);
     if (ret == 0)
          return (0);
}

/* Otherwise, return -1 to indicate an error condition. */
return (-1);
```

EXAMPLE 3 Convert a UTF-8 string into UTF-16 in little-endian byte ordering.

Convert a UTF-8 string into UTF-16 in little-endian byte ordering with a Byte Order Mark (BOM) character at the beginning of the output parameter.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/u8_textprep>
.
.
.
uchar_t u8s[MAXNAMELEN + 1];
uint16_t u16s[MAXNAMELEN + 1];
size_t u8len, u16len;
int ret;
.
.
.
u8len = u16len = MAXNAMELEN;
ret = uconv_u8tou16(u8s, &u8len, u16s, &u16len,
    UCONV_IN_LITTLE_ENDIAN | UCONV_EMIT_BOM);
if (ret != 0) {
     /* Conversion error occurred. */
     return (ret);
}
.
.
.
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also attributes(5), uconv_u16tou32(9F)

The Unicode Standard (http://www.unicode.org)

Notes Each UTF-16 or UTF-32 character maps to an UTF-8 character that might need one to maximum of four bytes.

One UTF-32 or UTF-8 character can yield two 16-bit unsigned integers as a UTF-16 character, which is a surrogate pair if the Unicode scalar value is bigger than U+FFFF.

Ill-formed UTF-16 surrogate pairs are seen as illegal characters during the conversion.

Name ucred_get, ucred_free, ucred_geteuid, ucred_getruid, ucred_getsuid, ucred_getegid, ucred_getrgid, ucred_getsgid, ucred_getgroups, ucred_getprivset, ucred_getpid, ucred_getprojid, ucred_getzoneid, ucred_getpflags, ucred_getlabel, ucred_size – user credential functions

Synopsis

```
#include <ucred.h>

ucred_t *ucred_get(pid_t pid);

void ucred_free(ucred_t *uc);

uid_t ucred_geteuid(const ucred_t *uc);

uid_t ucred_getruid(const ucred_t *uc);

uid_t ucred_getsuid(const ucred_t *uc);

gid_t ucred_getegid(const ucred_t *uc);

gid_t ucred_getrgid(const ucred_t *uc);

gid_t ucred_getsgid(const ucred_t *uc);

int ucred_getgroups(const ucred_t *uc, const gid_t **groups);

const priv_set_t *ucred_getprivset(const ucred_t *uc,
     priv_ptype_t set);

pid_t ucred_getpid(const ucred_t *uc);

projid_t ucred_getprojid(const ucred_t *uc);

zoneid_t ucred_getzoneid(const ucred_t *uc);

uint_t ucred_getpflags(const ucred_t *uc, uint_t flags);

m_label_t *ucred_getlabel(const ucred_t *uc);

size_t ucred_size(void);
```

Description These functions return or act on a user credential, `ucred_t`. User credentials are returned by various functions and describe the credentials of a process. Information about the process can then be obtained by calling the access functions. Access functions can fail if the underlying mechanism did not return sufficient information.

The `ucred_get()` function returns the user credential of the specified *pid* or `NULL` if none can be obtained. A *pid* value of `P_MYID` returns information about the calling process. The return value is dynamically allocated and must be freed using `ucred_free()`.

The `ucred_geteuid()`, `ucred_getruid()`, `ucred_getsuid()`, `ucred_getegid()`, `ucred_getrgid()`, and `ucred_getsgid()` functions return the effective UID, real UID, saved UID, effective GID, real GID, saved GID, respectively, or -1 if the user credential does not contain sufficient information.

The `ucred_getgroups()` function stores a pointer to the group list in the `gid_t *` pointed to by the second argument and returns the number of groups in the list. It returns -1 if the information is not available. The returned group list is valid until `ucred_free()` is called on the user credential given as argument.

The `ucred_getpid()` function returns the process ID of the process or -1 if the process ID is not available. The process ID returned in a user credential is only guaranteed to be correct in a very limited number of cases when returned by `door_ucred(3C)` and `ucred_get()`. In all other cases, the process in question might have handed of the file descriptor, the process might have exited or executed another program, or the process ID might have been reused by a completely unrelated process after the original program exited.

The `ucred_getprojid()` function returns the project ID of the process or -1 if the project ID is not available.

The `ucred_getzoneid()` function returns the zone ID of the process or –1 if the zone ID is not available.

The `ucred_getprivset()` function returns the specified privilege set specified as second argument, or `NULL` if either the requested information is not available or the privilege set name is invalid. The returned privilege set is valid until `ucred_free()` is called on the specified user credential.

The `ucred_getpflags()` function returns the value of the specified privilege flags from the `ucred` structure, or `(uint_t)-1` if none was present.

The `ucred_getlabel()` function returns the value of the label, or `NULL` if the label is not available. The returned label is valid until `ucred_free()` is called on the specified user credential. This function is available only if the system is configured with Trusted Extensions.

The `ucred_free()` function frees the memory allocated for the specified user credential.

The `ucred_size()` function returns `sizeof(ucred_t)`. This value is constant only until the next boot, at which time it could change. The `ucred_size()` function can be used to determine the size of the buffer needed to receive a credential option with `SO_RECVUCRED`. See `socket.h(3HEAD)`.

Return Values See DESCRIPTION.

Errors The `ucred_get()` function will fail if:

`EAGAIN` There is not enough memory available to allocate sufficient memory to hold a user credential. The application can try again later.

`EACCES` The caller does not have sufficient privileges to examine the target process.

`EMFILE`
`ENFILE` The calling process cannot open any more files.

ENOMEM The physical limits of the system are exceeded by the memory allocation needed to hold a user credential.

ESRCH The target process does not exist.

The `ucred_getprivset()` function will fail if:

EINVAL The privilege set argument is invalid.

The `ucred_getlabel()` function will fail if:

EINVAL The label is not present.

The `ucred_geteuid()`, `ucred_getruid()`, `ucred_getsuid()`, `ucred_getegid()`, `ucred_getrgid()`, `ucred_getsgid()`, `ucred_getgroups()`, `ucred_getpflags()`, `ucred_getprivset()`, `ucred_getprojid()`, `ucred_getpid()`, and `ucred_getlabel()` functions will fail if:

EINVAL The requested user credential attribute is not available in the specified user credential.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

`getpflags(2)`, `getppriv(2)`, `door_ucred(3C)`, `getpeerucred(3C)`, `priv_set(3C)`, `socket.h(3HEAD)`, `attributes(5)`, `labels(5)`, `privileges(5)`

Name umem_alloc, umem_zalloc, umem_free, umem_nofail_callback – fast, scalable memory allocation

Synopsis

```
cc [ flag ... ] file... -lumem [ library ... ]
#include <umem.h>

void *umem_alloc(size_t size, int  flags);

void *umem_zalloc(size_t size, int  flags);

void umem_free(void *buf, size_t size);

void umem_nofail_callback((int (*callback)(void));

void *malloc(size_t size);

void *calloc(size_t nelem, size_t elsize);

void free(void *ptr);

void *memalign(size_t alignment, size_t size);

void *realloc(void *ptr, size_t size);

void *valloc(size_t size);
```

Description The `umem_alloc()` function returns a pointer to a block of *size* bytes suitably aligned for any variable type. The initial contents of memory allocated using `umem_alloc()` is undefined. The *flags* argument determines the behavior of `umem_alloc()` if it is unable to fulfill the request. The *flags* argument can take the following values:

`UMEM_DEFAULT` Return `NULL` on failure.

`UMEM_NOFAIL` Call an optional *callback* (set with `umem_nofail_callback()`) on failure. The *callback* takes no arguments and can finish by:

- returning `UMEM_CALLBACK_RETRY`, in which case the allocation will be retried. If the allocation fails, the callback will be invoked again.
- returning `UMEM_CALLBACK_EXIT`(*status*), in which case `exit(2)` is invoked with *status* as its argument. The `exit()` function is called only once. If multiple threads return from the `UMEM_NOFAIL` callback with `UMEM_CALLBACK_EXIT`(*status*), one will call `exit()` while the other blocks until `exit()` terminates the program.
- invoking a context-changing function (`setcontext(2)`) or a non-local jump (`longjmp(3C)` or `siglongjmp(3C)`, or ending the current thread of control (`thr_exit(3C)` or `pthread_exit(3C)`. The application is responsible for any necessary cleanup. The state of `libumem` remains consistent.

If no callback has been set or the callback has been set to `NULL`, `umem_alloc`(..., `UMEM_NOFAIL`) behaves as though the callback returned `UMEM_CALLBACK_EXIT`(255).

The libumem library can call callbacks from any place that a UMEM_NOFAIL allocation is issued. In multithreaded applications, callbacks are expected to perform their own concurrency management.

The function call umem_alloc(0, *flag*) always returns NULL. The function call umem_free(NULL, 0) is allowed.

The umem_zalloc() function has the same semantics as umem_alloc(), but the block of memory is initialized to zeros before it is returned.

The umem_free() function frees blocks previously allocated using umem_alloc() and umem_zalloc(). The buffer address and size must exactly match the original allocation. Memory must not be returned piecemeal.

The umem_nofail_callback() function sets the process-wide UMEM_NOFAIL callback. See the description of UMEM_NOFAIL for more information.

The malloc(), calloc(), free(), memalign(), realloc(), and valloc() functions are as described in malloc(3C). The libumem library provides these functions for backwards-compatibility with the standard functions.

Environment Variables

See umem_debug(3MALLOC) for environment variables that effect the debugging features of the libumem library.

UMEM_OPTIONS — Contains a list of comma-separated options. Unrecognized options are ignored. The options that are supported are:

backend=sbrk
backend=mmap — Set the underlying function used to allocate memory. This option can be set to sbrk (the default) for an sbrk(2)–based source or mmap for an mmap(2)–based source. If set to a value that is not supported, sbrk will be used.

Examples

EXAMPLE 1 Using the umem_alloc() function.

```
#include <stdio.h>
#include <umem.h>
...
char *buf = umem_alloc(1024, UMEM_DEFAULT);

if (buf == NULL) {
     fprintf(stderr, "out of memory\n");
          return (1);
}
/* cannot assume anything about buf's contents */
...
umem_free(buf, 1024);
```

EXAMPLE 1 Using the `umem_alloc()` function. *(Continued)*

```
...
```

EXAMPLE 2 Using the `umem_zalloc()` function

```
#include <stdio.h>
#include <umem.h>
...
char *buf = umem_zalloc(1024, UMEM_DEFAULT);

if (buf == NULL) {
    fprintf(stderr, "out of memory\n");
         return (1);
}
/* buf contains zeros */
...
umem_free(buf, 1024);
...
```

EXAMPLE 3 Using UMEM_NOFAIL

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

/*
 * Note that the allocation code below does not have to
 * check for umem_alloc() returning NULL
 */
int
my_failure_handler(void)
{
         (void) fprintf(stderr, "out of memory\n");
         return (UMEM_CALLBACK_EXIT(255));
}
...
umem_nofail_callback(my_failure_handler);
...
int i;
char *buf[100];

for (i = 0; i < 100; i++)
         buf[i] = umem_alloc(1024 * 1024, UMEM_NOFAIL);
...
for (i = 0; i < 100; i++)
    umem_free(buf[i], 1024 * 1024);
...
```

EXAMPLE 4 Using UMEM_NOFAIL in a multithreaded application

```
#define _REENTRANT
#include <thread.h>
#include <stdio.h>
#include <umem.h>

void *
start_func(void *the_arg)
{
          int *info = (int *)the_arg;
          char *buf = umem_alloc(1024 * 1024, UMEM_NOFAIL);

          /* does not need to check for buf == NULL */
          buf[0] = 0;
          ...
          /*
           * if there were other UMEM_NOFAIL allocations,
           * we would need to arrange for buf to be
           * umem_free()ed upon failure.
           */
          ...
          umem_free(buf, 1024 * 1024);
          return (the_arg);
}
...
int
my_failure_handler(void)
{
         /* terminate the current thread with status NULL */
         thr_exit(NULL);
}
...
umem_nofail_callback(my_failure_handler);
...
int my_arg;

thread_t tid;
void *status;

(void) thr_create(NULL, NULL, start_func, &my_arg, 0,
    NULL);
...
while (thr_join(0, &tid, &status) != 0)
          ;

if (status == NULL) {
    (void) fprintf(stderr, "thread %d ran out of memory\n",
```

EXAMPLE 4 Using UMEM_NOFAIL in a multithreaded application *(Continued)*

```
            tid);
}
...
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See below.

For `malloc()`, `calloc()`, `free()`, `realloc()`, and `valloc()`, see `standards(5)`.

See Also `exit(2)`, `mmap(2)`, `sbrk(2)`, `bsdmalloc(3MALLOC)`, `libumem(3LIB)`, `longjmp(3C)`, `malloc(3C)`, `malloc(3MALLOC)`, `mapmalloc(3MALLOC)`, `pthread_exit(3C)`, `thr_exit(3C)`, `umem_cache_create(3MALLOC)`, `umem_debug(3MALLOC)`, `watchmalloc(3MALLOC)`, `attributes(5)`, `standards(5)`

Oracle Solaris Modular Debugger Guide

Warnings Any of the following can cause undefined results:

- Passing a pointer returned from `umem_alloc()` or `umem_zalloc()` to `free()` or `realloc()`.
- Passing a pointer returned from `malloc()`, `calloc()`, `valloc()`, `memalign()`, or `realloc()` to `umem_free()`.
- Writing past the end of a buffer allocated using `umem_alloc()` or `umem_zalloc()`
- Performing `UMEM_NOFAIL` allocations from an `atexit(3C)` handler.

If the `UMEM_NOFAIL` callback performs `UMEM_NOFAIL` allocations, infinite recursion can occur.

Notes The following list compares the features of the `malloc(3C)`, `bsdmalloc(3MALLOC)`, `malloc(3MALLOC)`, `mtmalloc(3MALLOC)` , and the `libumem` functions.

- The `malloc(3C)`, `bsdmalloc(3MALLOC)`, and `malloc(3MALLOC)` functions have no support for concurrency. The `libumem` and `mtmalloc(3MALLOC)` functions support concurrent allocations.
- The `bsdmalloc(3MALLOC)` functions afford better performance but are space-inefficient.
- The `malloc(3MALLOC)` functions are space-efficient but have slower performance.
- The standard, fully SCD-compliant `malloc(3C)` functions are a trade-off between performance and space-efficiency.

- The `mtmalloc(3MALLOC)` functions provide fast, concurrent `malloc()` implementations that are not space-efficient.
- The `libumem` functions provide a fast, concurrent allocation implementation that in most cases is more space-efficient than `mtmalloc(3MALLOC)`.

Name umem_cache_create, umem_cache_destroy, umem_cache_alloc, umem_cache_free – allocation cache manipulation

Synopsis

```
cc [ flag ... ] file... -lumem [ library ... ]
#include <umem.h>

umem_cache_t *umem_cache_create(char *debug_name, size_t bufsize,
     size_t align, umem_constructor_t *constructor,
     umem_destructor_t *destructor, umem_reclaim_t *reclaim,
     void *callback_data, vmem_t *source, int cflags);

void umem_cache_destroy(umem_cache_t *cache);

void *umem_cache_alloc(umem_cache_t *cache, int flags);

void umem_cache_free(umem_cache_t *cache, void *buffer);
```

Description These functions create, destroy, and use an "object cache" An object cache is a collection of buffers of a single size, with optional content caching enabled by the use of callbacks (see `Cache Callbacks`). Object caches are MT-Safe. Multiple allocations and freeing of memory from different threads can proceed simultaneously. Object caches are faster and use less space per buffer than `malloc(3MALLOC)` and `umem_alloc(3MALLOC)`. For more information about object caching, see "The Slab Allocator: An Object-Caching Kernel Memory Allocator" and "Magazines and vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources".

The `umem_cache_create()` function creates object caches. Once a cache has been created, objects can be requested from and returned to the cache using `umem_cache_alloc()` and `umem_cache_free()`, respectively. A cache with no outstanding buffers can be destroyed with `umem_cache_destroy()`.

Creating and Destroying Caches

The `umem_cache_create()` function creates a cache of objects and takes as arguments the following:

debug_name	A human-readable name for debugging purposes.
bufsize	The size, in bytes, of the buffers in this cache.
align	The minimum alignment required for buffers in this cache. This parameter must be a power of 2. If 0, it is replaced with the minimum required alignment for the current architecture.
constructor	The callback to construct an object.
destructor	The callback to destroy an object.
reclaim	The callback to reclaim objects.
callback_data	An opaque pointer passed to the callbacks.
source	This parameter must be `NULL`.

cflags This parameter must be either 0 or `UMC_NODEBUG`. If `UMC_NODEBUG`, all debugging features are disabled for this cache. See `umem_debug`(3MALLOC).

Each cache can have up to three associated callbacks:

```
int constructor(void *buffer, void *callback_data, int flags);
void destructor(void *buffer, void *callback_data);
void reclaim(void *callback_data);
```

The *callback_data* argument is always equal to the value passed to `umem_cache_create()`, thereby allowing a client to use the same callback functions for multiple caches, but with customized behavior.

The reclaim callback is called when the umem function is requesting more memory from the operating system. This callback can be used by clients who retain objects longer than they are strictly needed (for example, caching non-active state). A typical reclaim callback might return to the cache ten per cent of the unneeded buffers.

The constructor and destructor callbacks enable the management of buffers with the constructed state. The constructor takes as arguments a buffer with undefined contents, some callback data, and the flags to use for any allocations. This callback should transform the buffer into the constructed state.

The destructor callback takes as an argument a constructed object and prepares it for return to the general pool of memory. The destructor should undo any state that the constructor created. For debugging, the destructor can also check that the buffer is in the constructed state, to catch incorrectly freed buffers. See `umem_debug`(3MALLOC) for further information on debugging support.

The `umem_cache_destroy()` function destroys an object cache. If the cache has any outstanding allocations, the behavior is undefined.

Allocating Objects

The `umem_cache_alloc()` function takes as arguments:

cache a cache pointer

flags flags that determine the behavior if `umem_cache_alloc()` is unable to fulfill the allocation request

If successful, `umem_cache_alloc()` returns a pointer to the beginning of an object of *bufsize* length.

There are three cases to consider:

- A new buffer needed to be allocated. If the cache was created with a constructor, it is applied to the buffer and the resulting object is returned.

- The object cache was able to use a previously freed buffer. If the cache was created with a constructor, the object is returned unchanged from when it was freed.
- The allocation of a new buffer failed. The *flags* argument determines the behavior:

  `UMEM_DEFAULT` The `umem_cache_alloc()` function returns `NULL` if the allocation fails.

  `UMEM_NOFAIL` The `umem_cache_alloc()` function cannot return `NULL`. A callback is used to determine what action occurs. See `umem_alloc`(3MALLOC) for more information.

Freeing Objects

The `umem_cache_free()` function takes as arguments:

cache a cache pointer

buf a pointer previously returned from `umem_cache_alloc()`. This argument must not be `NULL`.

If the cache was created with a constructor callback, the object must be returned to the constructed state before it is freed.

Undefined behavior results if an object is freed multiple times, if an object is modified after it is freed, or if an object is freed to a cache other than the one from which it was allocated.

Caches with Constructors

When a constructor callback is in use, there is essentially a contract between the cache and its clients. The cache guarantees that all objects returned from `umem_cache_alloc()` will be in the constructed state, and the client guarantees that it will return the object to the constructed state before handing it to `umem_cache_free()`.

Return Values

Upon failure, the `umem_cache_create()` function returns a null pointer.

Errors

The `umem_cache_create()` function will fail if:

`EAGAIN` There is not enough memory available to allocate the cache data structure.

`EINVAL` The *debug_name* argument is `NULL`, the *align* argument is not a power of two or is larger than the system pagesize, or the *bufsize* argument is 0.

`ENOMEM` The `libumem` library could not be initialized, or the *bufsize* argument is too large and its use would cause integer overflow to occur.

Examples

EXAMPLE 1 Use a fixed-size structure with no constructor callback.

```
#include <umem.h>

typedef struct my_obj {
     long my_data1;
} my_obj_t;

/*
```

EXAMPLE 1 Use a fixed-size structure with no constructor callback. *(Continued)*

```
 * my_objs can be freed at any time.  The contents of
 * my_data1 is undefined at allocation time.
 */

umem_cache_t *my_obj_cache;

...
my_obj_cache = umem_cache_create("my_obj", sizeof (my_obj_t),
    0, NULL, NULL, NULL, NULL, NULL, 0);
...
my_obj_t *cur = umem_cache_alloc(my_obj_cache, UMEM_DEFAULT);
...
/* use cur */
...
umem_cache_free(my_obj_cache, cur);
...
```

EXAMPLE 2 Use an object with a mutex.

```
#define _REENTRANT
#include <synch.h>
#include <umem.h>

typedef struct my_obj {
          mutex_t my_mutex;
          long my_data;
} my_obj_t;

/*
 * my_objs can only be freed when my_mutex is unlocked.
 */
int
my_obj_constructor(void *buf, void *ignored, int flags)
{
          my_obj_t *myobj = buf;

          (void) mutex_init(&my_obj->my_mutex, USYNC_THREAD, NULL);

          return (0);
}

void
my_obj_destructor(void *buf, void *ignored)
{
          my_obj_t *myobj = buf;
```

EXAMPLE 2 Use an object with a mutex. *(Continued)*

```
          (void) mutex_destroy(&my_obj->my_mutex);
}

umem_cache_t *my_obj_cache;

...
my_obj_cache = umem_cache_create("my_obj", sizeof (my_obj_t),
    0, my_obj_constructor, my_obj_destructor, NULL, NULL,
         NULL, 0);
...
my_obj_t *cur = umem_cache_alloc(my_obj_cache, UMEM_DEFAULT);
cur->my_data = 0;       /* cannot assume anything about my_data */
...
umem_cache_free(my_obj_cache, cur);
...
```

EXAMPLE 3 Use a more complex object with a mutex.

```
#define _REENTRANT
#include <assert.h>
#include <synch.h>
#include <umem.h>

typedef struct my_obj {
          mutex_t my_mutex;
          cond_t my_cv;
          struct bar *my_barlist;
          unsigned my_refcount;
} my_obj_t;

/*
 * my_objs can only be freed when my_barlist == NULL,
 * my_refcount == 0, there are no waiters on my_cv, and
 * my_mutex is unlocked.
 */

int
my_obj_constructor(void *buf, void *ignored, int flags)
{
          my_obj_t *myobj = buf;

          (void) mutex_init(&my_obj->my_mutex, USYNC_THREAD, NULL);
          (void) cond_init(&my_obj->my_cv, USYNC_THREAD, NULL);
          myobj->my_barlist = NULL;
          myobj->my_refcount = 0;
```

EXAMPLE 3 Use a more complex object with a mutex. *(Continued)*

```
          return (0);
}

void
my_obj_destructor(void *buf, void *ignored)
{
          my_obj_t *myobj = buf;

          assert(myobj->my_refcount == 0);
          assert(myobj->my_barlist == NULL);
          (void) cond_destroy(&my_obj->my_cv);
          (void) mutex_destroy(&my_obj->my_mutex);
}

umem_cache_t *my_obj_cache;

...
my_obj_cache = umem_cache_create("my_obj", sizeof (my_obj_t),
    0, my_obj_constructor, my_obj_destructor, NULL, NULL,
         NULL, 0);
...
my_obj_t *cur = umem_cache_alloc(my_obj_cache, UMEM_DEFAULT);
...
/* use cur */
...
umem_cache_free(my_obj_cache, cur);
...
```

EXAMPLE 4 Use objects with a subordinate buffer while reusing callbacks.

```
#include assert.h>
#include umem.h>

typedef struct my_obj {
          char *my_buffer;
          size_t my_size;
} my_obj_t;

/*
 * my_size and the my_buffer pointer should never be changed
 */

int
my_obj_constructor(void *buf, void *arg, int flags)
{
          size_t sz = (size_t)arg;
```

EXAMPLE 4 Use objects with a subordinate buffer while reusing callbacks. *(Continued)*

```
          my_obj_t *myobj = buf;

          if ((myobj->my_buffer = umem_alloc(sz, flags)) == NULL)
                return (1);

          my_size = sz;

          return (0);
}

void
my_obj_destructor(void *buf, void *arg)
{
          size_t sz = (size_t)arg;

          my_obj_t *myobj = buf;

          assert(sz == buf->my_size);
          umem_free(myobj->my_buffer, sz);
}

...
umem_cache_t *my_obj_4k_cache;
umem_cache_t *my_obj_8k_cache;
...
my_obj_cache_4k = umem_cache_create("my_obj_4k", sizeof (my_obj_t),
         0, my_obj_constructor, my_obj_destructor, NULL,
         (void *)4096, NULL, 0);

my_obj_cache_8k = umem_cache_create("my_obj_8k", sizeof (my_obj_t),
         0, my_obj_constructor, my_obj_destructor, NULL,
         (void *)8192, NULL, 0);
...
my_obj_t *my_obj_4k = umem_cache_alloc(my_obj_4k_cache,
         UMEM_DEFAULT);
my_obj_t *my_obj_8k = umem_cache_alloc(my_obj_8k_cache,
         UMEM_DEFAULT);
/* no assumptions should be made about the contents
of the buffers */
...
/* make sure to return them to the correct cache */
umem_cache_free(my_obj_4k_cache, my_obj_4k);
umem_cache_free(my_obj_8k_cache, my_obj_8k);
...
```

See the EXAMPLES section of umem_alloc(3MALLOC) for examples involving the UMEM_NOFAIL flag.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also setcontext(2), atexit(3C), libumem(3LIB), longjmp(3C), swapcontext(3C), thr_exit(3C), umem_alloc(3MALLOC), umem_debug(3MALLOC), attributes(5)

Bonwick, Jeff, "The Slab Allocator: An Object-Caching Kernel Memory Allocator", Proceedings of the Summer 1994 Usenix Conference.

Bonwick, Jeff and Jonathan Adams, "Magazines and vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources", Proceedings of the Summer 2001 Usenix Conference.

Warnings Any of the following can cause undefined results:

- Destroying a cache that has outstanding allocated buffers.
- Using a cache after it has been destroyed.
- Calling umem_cache_free() on the same buffer multiple times.
- Passing a NULL pointer to umem_cache_free().
- Writing past the end of a buffer.
- Reading from or writing to a buffer after it has been freed.
- Performing UMEM_NOFAIL allocations from an atexit(3C) handler.

Per-cache callbacks can be called from a variety of contexts. The use of functions that modify the active context, such as setcontext(2), swapcontext(3C), and thr_exit(3C), or functions that are unsafe for use in multithreaded applications, such as longjmp(3C) and siglongjmp(3C), result in undefined behavior.

A constructor callback that performs allocations must pass its *flags* argument unchanged to umem_alloc(3MALLOC) and umem_cache_alloc(). Any allocations made with a different flags argument results in undefined behavior. The constructor must correctly handle the failure of any allocations it makes.

Notes Object caches make the following guarantees about objects:

- If the cache has a constructor callback, it is applied to every object before it is returned from umem_cache_alloc() for the first time.
- If the cache has a constructor callback, an object passed to umem_cache_free() and later returned from umem_cache_alloc() is not modified between the two events.
- If the cache has a destructor, it is applied to all objects before their underlying storage is returned.

No other guarantees are made. In particular, even if there are buffers recently freed to the cache, `umem_cache_alloc()` can fail.

Name umem_debug – debugging features of the umem library

Synopsis

```
cc [ flag... ] file... -lumem [ library... ]
#include <umem.h>
```

Description The `libumem` library provides debugging features that detect memory leaks, buffer overruns, multiple frees, use of uninitialized data, use of freed data, and many other common programming errors. The activation of the run-time debugging features is controlled by environment variables.

When the library detects an error, it writes a description of the error to an internal buffer that is readable with the `::umem_status` `mdb(1)` *dcmd* and then calls `abort(3C)`.

Environment Variables

`UMEM_DEBUG` This variable contains a list of comma-separated options. Unrecognized options are ignored. Possible options include:

`audit`[=*frames*] This option enables the recording of auditing information, including thread ID, high-resolution time stamp, and stack trace for the last action (allocation or free) on every allocation. If transaction logging (see `UMEM_LOGGING`) is enabled, this auditing information is also logged.

The *frames* parameter sets the number of stack frames recorded in the auditing structure. The upper bound for frames is implementation-defined. If a larger value is requested, the upper bound is used instead.

If *frames* is not specified or is not an integer, the default value of 15 is used.

This option also enables the `guards` option.

`contents`[=*count*] If auditing and contents logging (see `UMEM_LOGGING`) are enabled, the first *count* bytes of each buffer are logged when they are freed. If a buffer is shorter than *count* bytes, it is logged in its entirety.

If *count* is not specified or is not an integer, the default value of 256 is used.

`default` This option is equivalent to `audit,contents,guards`.

`guards` This option enables filling allocated and freed buffers with special patterns to help detect the use

of uninitialized data and previously freed buffers. It also enables an 8-byte redzone after each buffer that contains `0xfeedfacefeedfaceULL`.

When an object is freed, it is filled with `0xdeadbeef`. When an object is allocated, the `0xdeadbeef` pattern is verified and replaced with `0xbaddcafe`. The redzone is checked every time a buffer is allocated or freed.

For caches with either constructors or destructors, or both, `umem_cache_alloc(3MALLOC)` and `umem_cache_free(3MALLOC)` apply the cache's constructor and destructor, respectively, instead of caching constructed objects. The presence of `assert(3C)`s in the destructor verifying that the buffer is in the constructed state can be used to detect any objects returned in an improper state. See `umem_cache_create(3MALLOC)` for details.

`verbose` — The library writes error descriptions to standard error before aborting. These messages are not localized.

`UMEM_LOGGING` — To be enabled, this variable should be set to a comma-separated list of in-memory logs. The logs available are:

`transaction`[=*size*] — If the `audit` debugging option is set (see `UMEM_DEBUG`), the audit structures from previous transactions are entered into this log.

`contents`[=*size*] — If the `audit` debugging option is set, the contents of objects are recorded in this log as they are freed.

If the "contents" debugging option was not set, 256 bytes of each freed buffer are saved.

`fail`[=*size*] — Records are entered into this log for every failed allocation.

For any of these options, if *size* is not specified, the default value of 64k is used. The *size* parameter must be an integer that can be qualified with K, M, G, or T to specify kilobytes, megabytes, gigabytes, or terabytes, respectively.

Logs that are not listed or that have either a size of 0 or an invalid size are disabled.

The log is disabled if during initialization the requested amount of storage cannot be allocated.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted
MT-Level	MT-Safe

See Also `mdb(1)`, `abort(3C)`, `signal(3C)`, `umem_cache_create(3MALLOC)`, `attributes(5)`

Oracle Solaris Modular Debugger Guide

Warnings When `libumem` aborts the process using `abort(3C)`, any existing signal handler for `SIGABRT` is called. If the signal handler performs allocations, undefined behavior can result.

Notes Some of the debugging features work only for allocations smaller than 16 kilobytes in size. Allocations larger than 16 kilobytes could have reduced support.

Activating any of the library's debugging features could significantly increase the library's memory footprint and decrease its performance.

Name ungetc – push byte back into input stream

Synopsis

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

Description The `ungetc()` function pushes the byte specified by *c* (converted to an `unsigned char`) back onto the input stream pointed to by *stream*. The pushed-back bytes will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function (`fseek`(3C), `fsetpos`(3C) or `rewind`(3C)) discards any pushed-back bytes for the stream. The external storage corresponding to the stream is unchanged.

Four bytes of push-back are guaranteed. If `ungetc()` is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro `EOF`, the operation fails and the input stream is unchanged.

A successful call to `ungetc()` clears the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back bytes will be the same as it was before the bytes were pushed back. The file-position indicator is decremented by each successful call to `ungetc()`; if its value was 0 before a call, its value is indeterminate after the call.

Return Values Upon successful completion, `ungetc()` returns the byte pushed back after conversion. Otherwise it returns `EOF`.

Errors No errors are defined.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards`(5).

See Also `read`(2), `Intro`(3), `__fsetlocking`(3C), `fseek`(3C), `fsetpos`(3C), `getc`(3C), `setbuf`(3C), `stdio`(3C), `attributes`(5), `standards`(5)

Name ungetwc – push wide-character code back into input stream

Synopsis

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

wint_t ungetwc(wint_t wc, FILE *stream);
```

Description The `ungetwc()` function pushes the character corresponding to the wide character code specified by *wc* back onto the input stream pointed to by *stream*. The pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function (`fseek`(3C), `fsetpos`(3C) or `rewind`(3C)) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of push-back is guaranteed. If `ungetwc()` is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

If the value of *wc* equals that of the macro `WEOF`, the operation fails and the input stream is unchanged.

A successful call to `ungetwc()` clears the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back characters will be the same as it was before the characters were pushed back. The file-position indicator is decremented (by one or more) by each successful call to `ungetwc()`; if its value was 0 before a call, its value is indeterminate after the call.

Return Values Upon successful completion, `ungetwc()` returns the wide-character code corresponding to the pushed-back character. Otherwise it returns `WEOF`.

Errors The `ungetwc()` function may fail if:

`EILSEQ` An invalid character sequence is detected, or a wide-character code does not correspond to a valid character.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `read`(2), `fseek`(3C), `fsetpos`(3C), `rewind`(3C), `setbuf`(3C), `attributes`(5), `standards`(5)

Name unlockpt – unlock a pseudo-terminal master/slave pair

Synopsis

```
#include <stdlib.h>

int unlockpt(int fildes);
```

Description The `unlockpt()` function unlocks the slave pseudo-terminal device associated with the master to which *fildes* refers.

Portable applications must call `unlockpt()` before opening the slave side of a pseudo-terminal device.

Return Values Upon successful completion, `unlockpt()` returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `unlockpt()` function may fail if:

`EBADF` The *fildes* argument is not a file descriptor open for writing.

`EINVAL` The *fildes* argument is not associated with a master pseudo-terminal device.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `open(2)`, `grantpt(3C)`, `ptsname(3C)`, `attributes(5)`, `standards(5)`

STREAMS Programming Guide

Name unsetenv – remove an environment variable

Synopsis

```
#include <stdlib.h>

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

Description The `unsetenv()` function removes an environment variable from the environment of the calling process. The *name* argument points to a string that is the name of the variable to be removed. The named argument cannot contain an '=' character. If the named variable does not exist in the current environment, the environment is unchanged and the function is considered to have completed successfully.

If the application modifies *environ* or the pointers to which it points, the behavior of `unsetenv()` is undefined. The `unsetenv()` function updates the list of pointers to which *environ* points.

Return Values Upon successful completion, 0 is returned. Otherwise, -1 is returned, `errno` set to indicate the error, and the environment is left unchanged.

Errors The `unsetenv()` function will fail if:

`EINVAL` The *name* argument is a null pointer, points to an empty string, or points to a string containing an '=' character.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `getenv(3C)`, `setenv(3C)`, `attributes(5)`, `standards(5)`

Name usleep – suspend execution for interval in microseconds

Synopsis

```
#include <unistd.h>

int usleep(useconds_t useconds);
```

Description The `usleep()` function suspends the caller from execution for the number of microseconds specified by the *useconds* argument. The actual suspension time might be less than requested because any caught signal will terminate `usleep()` following execution of that signal's catching routine. The suspension time might be longer than requested by an arbitrary amount because of the scheduling of other activity in the system.

If the value of *useconds* is 0, then the call has no effect.

The use of the `usleep()` function has no effect on the action or blockage of any signal. In a multithreaded process, only the invoking thread is suspended from execution.

Return Values On completion, `usleep()` returns `0`. There are no error returns.

Errors No errors are returned.

Usage The `usleep()` function is included for its historical usage. The `nanosleep(3C)` function is preferred over this function.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See `standards(5)`.

See Also `nanosleep(3C)`, `sleep(3C)`, `attributes(5)`, `standards(5)`

Name vfwprintf, vswprintf, vwprintf – wide-character formatted output of a stdarg argument list

Synopsis

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

int vfwprintf(FILE *restrict stream, const wchar_t *restrict format,
     va_list arg);

int vswprintf(wchar_t *restrict s, size_t n,
     const wchar_t *restrict format, va_list arg);

int vwprintf(const wchar_t *restrict format, va_list arg);
```

Description The `vwprintf()`, `vfwprintf()`, and `vswprintf()` functions are the same as `wprintf()`, `fwprintf()`, and `swprintf()` respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by <`stdarg.h`>.

These functions do not invoke the `va_end()` macro. However, as these functions do invoke the `va_arg()` macro, the value of *ap* after the return is indeterminate.

Return Values Refer to `fwprintf`(3C).

Errors Refer to `fwprintf`(3C).

Usage Applications using these functions should call `va_end(`*ap*`)` afterwards to clean up.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards`(5).

See Also `fwprintf`(3C), `setlocale`(3C), `attributes`(5), `standards`(5)

Notes The `vwprintf()`, `vfwprintf()`, and `vswprintf()` functions can be used safely in multithreaded applications, as long as `setlocale`(3C) is not being called to change the locale.

Name vlfmt – display error message in standard format and pass to logging and monitoring services

Synopsis

```
#include <pfmt.h>
#include <stdarg.h>

int vlfmt(FILE *stream, long flag, const char *format, va_list ap);
```

Description The `vlfmt()` function is identical to `lfmt(3C)`, except that it is called with an argument list as defined by `<stdarg.h>`.

The `<stdarg.h>` header defines the type `va_list` and a set of macros for advancing through a list of arguments whose number and types may vary. The *ap* argument is of type `va_list`. This argument is used with the `<stdarg.h>` macros `va_start()`, `va_arg()`, and `va_end()`. See `stdarg(3EXT)`. The example in the `EXAMPLES` section below demonstrates their use with `vlfmt()`.

Return Values Upon successful completion, `vlfmt()` returns the number of bytes transmitted. Otherwise, `−1` is returned if there was a write error to *stream*, or `−2` is returned if unable to log and/or display at console.

Examples **EXAMPLE 1** Use of `vlfmt()` to write an `errlog()` routine.

The following example demonstrates how `vlfmt()` could be used to write an `errlog()` routine. The `va_alist()` macro is used as the parameter list in a function definition. The `va_start(`*ap*`, . . .)` call, where *ap* is of type `va_list`, must be invoked before any attempt to traverse and access unnamed arguments. Calls to `va_arg(`*ap*`, `*atype*`)` traverse the argument list. Each execution of `va_arg()` expands to an expression with the value and type of the next argument in the list *ap*, which is the same object initialized by `va_start()`. The *atype* argument is the type that the returned argument is expected to be. The `va_end(`*ap*`)` macro must be invoked when all desired arguments have been accessed. The argument list in *ap* can be traversed again if `va_start()` is called again after `va_end()`.) In the example below, `va_arg()` is executed first to retrieve the format string passed to `errlog()`. The remaining `errlog()` arguments (*arg1*, *arg2*, ...) are passed to `vlfmt()` in the argument *ap*.

```
#include <pfmt.h>
#include <stdarg.h>
/*
 *   errlog should be called like
 *         errlog(log_info, format, arg1, ...);
 */
void errlog(long log_info, ...)
{
        va_list ap;
        char *format;
        va_start(ap, );
        format = va_arg(ap, char *);
        (void) vlfmt(stderr, log_info|MM_ERROR, format, ap);
        va_end(ap);
```

EXAMPLE 1 Use of `vlfmt()` to write an `errlog()` routine. *(Continued)*

```
        (void) abort();
}
```

Usage Since `vlfmt()` uses `gettxt(3C)`, it is recommended that `vlfmt()` not be used.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `gettxt(3C)`, `lfmt(3C)`, `attributes(5)`, `stdarg(3EXT)`

Name vpfmt – display error message in standard format and pass to logging and monitoring services

Synopsis

```
#include <pfmt.h>
#include <stdarg.h>

int vpfmt(FILE *stream, long flag, const char *format, va_list ap);
```

Description The `vpfmt()` function is identical to `pfmt(3C)`, except that it is called with an argument list as defined by `<stdarg.h>`.

The `<stdarg.h>` header defines the type `va_list` and a set of macros for advancing through a list of arguments whose number and types may vary. The *ap* argument is of type `va_list`. This argument is used with the `<stdarg.h>` macros `va_start()`, `va_arg()`, and `va_end()`. See `stdarg(3EXT)`. The example in the `EXAMPLES` section below demonstrates their use with `vpfmt()`.

Return Values Upon successful completion, `vpfmt()` returns the number of bytes transmitted. Otherwise, `−1` is returned if there was a write error to *stream*.

Examples **EXAMPLE 1** Use of `vpfmt()` to write an error routine.

The following example demonstrates how `vpfmt()` could be used to write an `error()` routine. The `va_alist()` macro is used as the parameter list in a function definition. The `va_start(`*ap*`, . . .)` call, where *ap* is of type `va_list`, must be invoked before any attempt to traverse and access unnamed arguments. Calls to `va_arg(`*ap*`,` *atype*`)` traverse the argument list. Each execution of `va_arg()` expands to an expression with the value and type of the next argument in the list *ap*, which is the same object initialized by `va_start()`. The *atype* argument is the type that the returned argument is expected to be. The `va_end(`*ap*`)` macro must be invoked when all desired arguments have been accessed. The argument list in *ap* can be traversed again if `va_start()` is called again after `va_end()`. In the example below, `va_arg()` is executed first to retrieve the format string passed to `error()`. The remaining `error()` arguments (*arg1*, *arg2*, ...) are passed to `vpfmt()` in the argument *ap*.

```
#include <pfmt.h>
#include <stdarg.h>
/*
 *   error should be called like
 *         error(format, arg1, ...);
 */
void error(...)
{
        va_list ap;
        char *format;
        va_start(ap, );
        format = va_arg(ap, char *);
        (void) vpfmt(stderr, MM_ERROR, format, ap);
        va_end(ap);
        (void) abort();
```

EXAMPLE 1 Use of `vpfmt()` to write an error routine. *(Continued)*

```
}
```

Usage

Since `vpfmt()` uses `gettxt(3C)`, it is recommended that `vpfmt()` not be used.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also

`gettxt(3C)`, `pfmt(3C)`, `attributes(5)`, `stdarg(3EXT)`

Name vprintf, vfprintf, vsprintf, vsnprintf, vasprintf – print formatted output of a variable argument list

Synopsis

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

int vprintf(const char *format, va_list ap);

int vfprintf(FILE *stream, const char *format, va_list ap);

int vsprintf(char *s, const char *format, va_list ap);

int vsnprintf(char *s, size_t n, const char *format, va_list ap);

int vasprintf(char **ret, const char *format, va_list ap);
```

Description The `vprintf()`, `vfprintf()`, `vsprintf()`, `vsnprintf()`, and `vasprintf()` functions are the same as `printf()`, `fprintf()`, `sprintf()`, `snprintf()`, and `asprintf()`, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the `<stdarg.h>` header. See `printf(3C)`.

The `<stdarg.h>` header defines the type `va_list` and a set of macros for advancing through a list of arguments whose number and types may vary. The argument *ap* to the `vprint` family of functions is of type `va_list`. This argument is used with the `<stdarg.h>` header file macros `va_start()`, `va_arg()`, and `va_end()` (see `stdarg(3EXT)`). The `EXAMPLES` section below demonstrates the use of `va_start()` and `va_end()` with `vprintf()`.

The macro `va_alist()` is used as the parameter list in a function definition, as in the function called `error()` in the example below. The macro `va_start(`*ap*, *name*`)`, where *ap* is of type `va_list` and *name* is the rightmost parameter (just before `...`), must be called before any attempt to traverse and access unnamed arguments is made. The `va_end(`*ap*`)` macro must be invoked when all desired arguments have been accessed. The argument list in *ap* can be traversed again if `va_start()` is called again after `va_end()`. In the example below, the `error()` arguments (*arg1*, *arg2*, ...) are passed to `vfprintf()` in the argument *ap*.

Return Values Refer to `printf(3C)`.

Errors The `vprintf()` and `vfprintf()` functions will fail if either the *stream* is unbuffered or the *stream*'s buffer needed to be flushed and:

`EFBIG` The file is a regular file and an attempt was made to write at or beyond the offset maximum.

Examples **EXAMPLE 1** Using `vprintf()` to write an error routine.

The following demonstrates how `vfprintf()` could be used to write an error routine:

```
#include <stdio.h>
#include <stdarg.h>
. . .
/*
```

EXAMPLE 1 Using `vprintf()` to write an error routine. *(Continued)*

```
 *   error should be called like
 *        error(function_name, format, arg1, ...);
 */
void error(char *function_name, char *format, ...)
{
        va_list ap;
        va_start(ap, format);
        /* print out name of function causing error */
        (void) fprintf(stderr, "ERR in %s: ", function_name);
        /* print out remainder of message */
        (void) vfprintf(stderr, format, ap);
        va_end(ap);
        (void) abort();
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See below.
Standard	See below.

All of these functions can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

See `standards(5)` for the standards conformance of `vprintf()`, `vfprintf()`, `vsprintf()`, and `vsnprintf()`. The `vasprintf()` function is modeled on the one that appears in the FreeBSD, NetBSD, and GNU C libraries.

See Also

`printf(3C)`, `attributes(5)`, `stdarg(3EXT)`, `attributes(5)`, `standards(5)`

Notes

The `vsnprintf()` return value when $n = 0$ was changed in the Solaris 10 release. The change was based on the SUSv3 specification. The previous behavior was based on the initial SUSv2 specification, where `vsnprintf()` when $n = 0$ returns an unspecified value less than 1.

Name vsyslog – log message with a stdarg argument list

Synopsis

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog(int priority, const char *message, va_list ap);
```

Description The `vsyslog()` function is identical to `syslog(3C)`, except that it is called with an argument list as defined by `<stdarg.h>` rather than with a variable number of arguments.

Examples **EXAMPLE 1** Use `vsyslog()` to write an error routine.

The following example demonstrates the use of `vsyslog()` in writing an error routine.

```
#include <syslog.h>
#include <stdarg.h>

/*
 * error should be called like:
 *   error(pri, function_name, format, arg1, arg2...);
 */

void
error(int pri, char *function_name, char *format, ...)
{
        va_list args;

        va_start(args, format);
        /* log name of function causing error */
        (void) syslog(pri, "ERROR in %s.", function_name);
        /* log remainder of message */
        (void) vsyslog(pri, format, args);
        va_end(args);
        (void) abort( );
}

main()
{
        error(LOG_ERR, "main", "process %d is dying", getpid());
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also syslog(3C), attributes(5)

Name wait3, wait4 – wait for process to terminate or stop

Synopsis

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

pid_t wait3(int *statusp, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *statusp, int options, struct rusage *rusage);
```

Description The `wait3()` function delays its caller until a signal is received or one of its child processes terminates or stops due to tracing. If any child process has died or stopped due to tracing and this has not already been reported, return is immediate, returning the process ID and status of one of those children. If that child process has died, it is discarded. If there are no children, −1 is returned immediately. If there are only running or stopped but reported children, the calling process is blocked.

If *statusp* is not a null pointer, then on return from a successful `wait3()` call, the status of the child process is stored in the integer pointed to by *statusp*. **statusp* indicates the cause of termination and other information about the terminated process in the following manner:

- If the low-order 8 bits of **statusp* are equal to 0177, the child process has stopped; the 8 bits higher up from the low-order 8 bits of **statusp* contain the number of the signal that caused the process to stop. See `signal.h`(3HEAD).
- If the low-order 8 bits of **statusp* are non-zero and are not equal to 0177, the child process terminated due to a signal; the low-order 7 bits of **statusp* contain the number of the signal that terminated the process. In addition, if the low-order seventh bit of **statusp* (that is, bit 0200) is set, a "core image" of the process was produced; see `signal.h`(3HEAD).
- Otherwise, the child process terminated due to an `exit()` call; the 8 bits higher up from the low-order 8 bits of **statusp* contain the low-order 8 bits of the argument that the child process passed to `exit()`; see `exit`(2).

The *options* argument is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in <`sys/wait.h`>:

`WNOHANG` Execution of the calling process is not suspended if status is not immediately available for any child process.

`WUNTRACED` The status of any child processes that are stopped, and whose status has not yet been reported since they stopped, are also reported to the requesting process.

If *rusage* is not a null pointer, a summary of the resources used by the terminated process and all its children is returned. Only the user time used and the system time used are currently available. They are returned in the `ru_utime` and `ru_stime`, members of the rusage structure, respectively.

When the WNOHANG option is specified and no processes have status to report, wait3() returns 0. The WNOHANG and WUNTRACED options may be combined by the bitwise OR operation of the two values.

The wait4() function is an extended interface. If *pid* is 0, wait4() is equivalent to wait3(). If *pid* has a nonzero value, wait4() returns status only for the indicated process ID, but not for any other child processes. If *pid* has a negative value, wait4() return status only for child processes whose process group ID is equal to the absolute value of *pid*. The status can be evaluated using the macros defined by wait.h(3HEAD).

Return Values If wait3() or wait4() returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, −1 is returned and errno is set to indicate the error.

If wait3() or wait4() return due to the delivery of a signal to the calling process, −1 is returned and errno is set to EINTR. If WNOHANG was set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

The wait3() and wait4() functions return 0 if WNOHANG is specified and there are no stopped or exited children, and return the process ID of the child process if they return due to a stopped or terminated child process. Otherwise, they return −1 and set errno to indicate the error.

Errors The wait3() and wait4() functions will fail and return immediately if:

ECHILD The calling process has no existing unwaited-for child processes.

EFAULT The *statusp* or *rusage* arguments point to an illegal address.

EINTR The function was interrupted by a signal. The value of the location pointed to by *statusp* is undefined.

EINVAL The value of *options* is not valid.

The wait4() function may fail if:

ECHILD The process specified by *pid* does not exist or is not a child of the calling process.

The wait3()and wait4() functions will terminate prematurely, return −1, and set errno to EINTR upon the arrival of a signal whose SA_RESTART bit in its flags field is not set (see sigaction(2)).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also kill(1), exit(2), waitid(2), waitpid(3C), getrusage(3C), signal(3C), signal.h(3HEAD), wait(3C), wait.h(3HEAD), proc(4), attributes(5)

Notes If a parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

The `wait3()` and `wait4()` functions are automatically restarted when a process receives a signal while awaiting termination of a child process, unless the `SA_RESTART` bit is not set in the flags for that signal.

Name wait – wait for child process to stop or terminate

Synopsis

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

pid_t wait(int *stat_loc);
```

Description The `wait()` function will suspend execution of the calling thread until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in `wait()`, `waitpid(3C)`, or `waitid(2)` awaiting termination of the same process, exactly one thread will return the process status at the time of the target process termination. If status information is available prior to the call to `wait()`, return will be immediate.

If `wait()` returns because the status of a child process is available, it returns the process ID of the child process. If the calling process specified a non-zero value for *stat_loc*, the status of the child process is stored in the location pointed to by *stat_loc*. That status can be evaluated with the macros described on the `wait.h(3HEAD)` manual page.

In the following, *status* is the object pointed to by *stat_loc*:

- If the child process terminated due to an `_exit()` call, the low order 8 bits of *status* will be 0 and the high order 8 bits will contain the low order 7 bits of the argument that the child process passed to `_exit()`; see `exit(2)`.
- If the child process terminated due to a signal, the high order 8 bits of *status* will be 0 and the low order 7bits will contain the number of the signal that caused the termination. In addition, if `WCOREFLG` is set, a "core image" will have been produced; see `signal.h(3HEAD)` and `wait.h(3HEAD)`.

One instance of a `SIGCHLD` signal is queued for each child process whose status has changed. If `wait()` returns because the status of a child process is available, any pending `SIGCHLD` signal associated with the process ID of that child process is discarded. Any other pending `SIGCHLD` signals remain pending.

If the calling process has `SA_NOCLDWAIT` set or has `SIGCHLD` set to `SIG_IGN`, and the process has no unwaited children that were transformed into zombie processes, it will block until all of its children terminate, and `wait()` will fail and set `errno` to `ECHILD`.

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1, with the initialization process inheriting the child processes; see `Intro(2)`.

Return Values When `wait()` returns due to a terminated child process, the process ID of the child is returned to the calling process. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `wait()` function will fail if:

`ECHILD` The calling process has no existing unwaited-for child processes.

`EINTR` The function was interrupted by a signal.

Usage Since `wait()` blocks on a stopped child, a calling process wanting to see the return results of such a call should use `waitpid(3C)` or `waitid(2)` instead of `wait()`. The `wait()` function is implemented as a call to `waitpid(-1, stat_loc, 0)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `Intro(2)`, `exec(2)`, `exit(2)`, `fork(2)`, `pause(2)`, `waitid(2)`, `ptrace(3C)`, `signal(3C)`, `signal.h(3HEAD)`, `waitpid(3C)`, `wait.h(3HEAD)`, `attributes(5)`

Name waitpid – wait for child process to change state

Synopsis

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Description The `waitpid()` function will suspend execution of the calling thread until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in `waitpid()`, `wait(3C)`, or `waitid(2)` awaiting termination of the same process, exactly one thread will return the process status at the time of the target process termination. If status information is available prior to the call to `waitpid()`, return will be immediate.

The *pid* argument specifies a set of child processes for which status is requested, as follows:

- If *pid* is less than `(pid_t)−1`, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.
- If *pid* is equal to `(pid_t)−1`, status is requested for any child process.
- If *pid* is equal to `(pid_t)0` status is requested for any child process whose process group ID is equal to that of the calling process.
- If *pid* is greater than `(pid_t)0`, it specifies the process ID of the child process for which status is requested.

One instance of a `SIGCHLD` signal is queued for each child process whose status has changed. If `waitpid()` returns because the status of a child process is available, and `WNOWAIT` was not specified in *options*, any pending `SIGCHLD` signal associated with the process ID of that child process is discarded. Any other pending `SIGCHLD` signals remain pending.

If the calling process has `SA_NOCLDWAIT` set or has `SIGCHLD` set to `SIG_IGN` and the process has no unwaited children that were transformed into zombie processes, it will block until all of its children terminate, and `waitpid()` will fail and set `errno` to `ECHILD`.

If `waitpid()` returns because the status of a child process is available, then that status may be evaluated with the macros defined by `wait.h(3HEAD)` If the calling process had specified a non-zero value of *stat_loc*, the status of the child process will be stored in the location pointed to by *stat_loc*.

The *options* argument is constructed from the bitwise-inclusive OR of zero or more of the following flags, defined in the header `<sys/wait.h>`:

`WCONTINUED` The status of any continued child process specified by *pid*, whose status has not been reported since it continued, is also reported to the calling process.

`WNOHANG` The `waitpid()` function will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.

`WNOWAIT` Keep the process whose status is returned in *stat_loc* in a waitable state. The process may be waited for again with identical results.

`WUNTRACED` The status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, is also reported to the calling process. `WSTOPPED` is a synonym for `WUNTRACED`.

Return Values If `waitpid()` returns because the status of a child process is available, it returns a value equal to the process ID of the child process for which status is reported. If `waitpid()` returns due to the delivery of a signal to the calling process, `−1` is returned and `errno` is set to `EINTR`. If `waitpid()` was invoked with `WNOHANG` set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, then `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `waitpid()` function will fail if:

`ECHILD` The process or process group specified by *pid* does not exist or is not a child of the calling process or can never be in the states specified by *options*.

`EINTR` The `waitpid()` function was interrupted due to the receipt of a signal sent by the calling process.

`EINVAL` An invalid value was specified for *options*.

Usage With *options* equal to `0` and *pid* equal to `(pid_t)−1`, `waitpid()` is identical to `wait(3C)`. The `waitpid()` function is implemented as a call to the more general `waitid(2)` function.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `Intro(2)`, `exec(2)`, `exit(2)`, `fork(2)`, `pause(2)`, `sigaction(2)`, `ptrace(3C)`, `signal(3C)`, `siginfo.h(3HEAD)`, `wait(3C)`, `wait.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name walkcontext, addrtosymstr, printstack, backtrace, backtrace_symbols, backtrace_symbols_fd – walk stack pointed to by ucontext

Synopsis

```
#include <ucontext.h>

int walkcontext(const ucontext_t *uptr,
     int (*operate_func)(uintptr_t, int, void *), void *usrarg);

int addrtosymstr(uintptr_t addr, char *buffer, int len);

int printstack(int fd);

#include <execinfo.h>

int backtrace(void **buffer, int size);

char **backtrace_symbols(void *const *buffer, int size);

void backtrace_symbols_fd(void *const *buffer, int size, int fd);
```

Description The `walkcontext()` function walks the call stack pointed to by *uptr*, which can be obtained by a call to `getcontext(2)` or from a signal handler installed with the `SA_SIGINFO` flag. The `walkcontext()` function calls the user-supplied function *operate_func* for each routine found on the call stack and each signal handler invoked. The user function is passed three arguments: the PC at which the call or signal occurred, the signal number that occurred at this PC (0 if no signal occurred), and the third argument passed to `walkcontext()`. If the user function returns a non-zero value, `walkcontext()` returns without completing the callstack walk.

The `addrtosymstr()` function attempts to convert a PC into a symbolic representation of the form

objname`'`*funcname*`+0x`*offset*`[0xPC]`

where *objname* is the module in which the PC is located, *funcname* is the name of the function, and *offset* is the offset from the beginning of the function. The *objname*, *funcname*, and *offset* values are obtained from `dladdr(3C)` and might not always be present. The resulting string is written to the user-supplied buffer. Should the length of the string be larger than the user-supplied buffer, only the portion of the string that will fit is written and null-terminated.

The `printstack()` function uses `walkcontext()` to print a symbolic stack trace to the specified file descriptor. This is useful for reporting errors from signal handlers. The `printstack()` function uses `dladdr1()` (see `dladdr(3C)`) to obtain symbolic symbol names. As a result, only global symbols are reported as symbol names by `printstack()`.

The `backtrace()` function uses `walkcontext()` to generate a stack's program counter values for the calling thread and place these values into the array specified by the buffer argument. The *size* argument specifies the maximum number of program counters that will be recorded. This function is provided for compatibility with the GNU `libc` used on Linux systems, `glibc`.

The `backtrace_symbols()` function translates the numerical program counter values previously recorded by a call to `backtrace()` in the *buffer* argument, and converts, where possible, each PC to a string indicating the module, function and offset of each call site. The number of symbols present in the array must be passed in with the *size* argument.

The set of strings is returned in an array obtained from a call to `malloc(3C)`. It is the responsibility of the caller to pass the returned pointer to `free()`. The individual strings must not be freed. Since `malloc()` is used to obtain the needed space, this function is MT-Safe rather than Async-Signal-Safe and cannot be used reliably from a signal handler. This function is provided for `glibc` compatibility.

The `backtrace_symbols_fd()` function translates the numerical program counter values previously recorded by a call to `backtrace()` in the buffer argument, and converts, where possible, each PC to a string indicating the module, function, and offset of each call site. These strings are written to the file descriptor specified in the *fd* argument. This function is provided for `glibc` compatibility.

Return Values Upon successful completion, `walkcontext()` and `printstack()` return 0. If `walkcontext()` cannot read the stack or the stack trace appears corrupted, both functions return -1.

The `addrtosymstr()` function returns the number of characters necessary to hold the entire string representation of the passed in address, irrespective of the size of the user-supplied buffer.

The `backtrace()` function returns the number of stack frames captured.

The `backtrace_symbols()` function returns a pointer to an array containing string representations of the calling sequence.

Errors No error values are defined.

Usage The `walkcontext()` function is typically used to obtain information about the call stack for error reporting, performance analysis, or diagnostic purposes. Many library functions are not Async-Signal-Safe and should not be used from a signal handler. If `walkcontext()` is to be called from a signal handler, careful programming is required. In particular, `stdio(3C)` and `malloc(3C)` cannot be used.

The `walkstack()`, `addrtosymstr()`, `printstack()`, `backtrace()`, and `backtrace_symbols_fd()` functions are Async-Signal-Safe and can be called from a signal handler. The string representation generated by `addrtosymstr()` and displayed by `printstack()`, `backtrace_symbols()` and `backtrace_symbols_fd()` is unstable and will change depending on the information available in the modules that comprise the stack trace.

Tail-call optimizations on SPARC eliminate stack frames that would otherwise be present. For example, if the code is of the form

```
#include <stdio.h>

main()
```

```
{
        bar();
        exit(0);
}

bar()
{
        int a;
        a = foo(fileno(stdout));
        return (a);
}

foo(int file)
{
        printstack(file);
}
```

compiling without optimization will yield a stack trace of the form

```
/tmp/q:foo+0x8
/tmp/q:bar+0x14
/tmp/q:main+0x4
/tmp/q:_start+0xb8
```

whereas with higher levels of optimization the output is

```
/tmp/q:main+0x10
/tmp/q:_start+0xb8
```

since both the call to `foo()` in main and the call to `bar()` in `foo()` are handled as tail calls that perform a return or restore in the delay slot. For further information, see *The SPARC Architecture Manual*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See below.

The `backtrace_symbols()` function is MT-Safe. The remaining functions are Async-Signal-Safe.

See Also `Intro(2)`, `getcontext(2)`, `sigaction(2)`, `dladdr(3C)`, `siginfo.h(3HEAD)`, `attributes(5)`

Weaver, David L. and Tom Germond, eds. *The SPARC Architecture Manual*, Version 9. Santa Clara: Prentice Hall, 2000.

Name watchmalloc – debugging memory allocator

Synopsis

```
#include <stdlib.h>

void *malloc(size_t size);

void free(void *ptr);

void *realloc(void *ptr, size_t size);

void *memalign(size_t alignment, size_t size);

void *valloc(size_t size);

void *calloc(size_t nelem, size_t elsize);

#include <malloc.h>

int mallopt(int cmd, int value);

struct mallinfo mallinfo(void);
```

Description The collection of `malloc()` functions in this shared object are an optional replacement for the standard versions of the same functions in the system C library. See `malloc(3C)`. They provide a more strict interface than the standard versions and enable enforcement of the interface through the watchpoint facility of `/proc`. See `proc(4)`.

Any dynamically linked application can be run with these functions in place of the standard functions if the following string is present in the environment (see `ld.so.1(1)`):

```
LD_PRELOAD=watchmalloc.so.1
```

The individual function interfaces are identical to the standard ones as described in `malloc(3C)`. However, laxities provided in the standard versions are not permitted when the watchpoint facility is enabled (see `WATCHPOINTS` below):

- Memory may not be freed more than once.
- A pointer to freed memory may not be used in a call to `realloc()`.
- A call to `malloc()` immediately following a call to `free()` will not return the same space.
- Any reference to memory that has been freed yields undefined results.

To enforce these restrictions partially, without great loss in speed as compared to the watchpoint facility described below, a freed block of memory is overwritten with the pattern `0xdeadbeef` before returning from `free()`. The `malloc()` function returns with the allocated memory filled with the pattern `0xbaddcafe` as a precaution against applications incorrectly expecting to receive back unmodified memory from the last `free()`. The `calloc()` function always returns with the memory zero-filled.

Entry points for `mallopt()` and `mallinfo()` are provided as empty routines, and are present only because some `malloc()` implementations provide them.

Watchpoints The watchpoint facility of `/proc` can be applied by a process to itself. The functions in `watchmalloc.so.1` use this feature if the following string is present in the environment:

MALLOC_DEBUG=WATCH

This causes every block of freed memory to be covered with `WA_WRITE` watched areas. If the application attempts to write any part of freed memory, it will trigger a watchpoint trap, resulting in a `SIGTRAP` signal, which normally produces an application core dump.

A header is maintained before each block of allocated memory. Each header is covered with a watched area, thereby providing a red zone before and after each block of allocated memory (the header for the subsequent memory block serves as the trailing red zone for its preceding memory block). Writing just before or just after a memory block returned by `malloc()` will trigger a watchpoint trap.

Watchpoints incur a large performance penalty. Requesting `MALLOC_DEBUG=WATCH` can cause the application to run 10 to 100 times slower, depending on the use made of allocated memory.

Further options are enabled by specifying a comma-separated string of options:

MALLOC_DEBUG=WATCH,RW,STOP

`WATCH` Enables `WA_WRITE` watched areas as described above.

`RW` Enables both `WA_READ` and `WA_WRITE` watched areas. An attempt either to read or write freed memory or the red zones will trigger a watchpoint trap. This incurs even more overhead and can cause the application to run up to 1000 times slower.

`STOP` The process will stop showing a `FLTWATCH` machine fault if it triggers a watchpoint trap, rather than dumping core with a `SIGTRAP` signal. This allows a debugger to be attached to the live process at the point where it underwent the watchpoint trap. Also, the various `/proc` tools described in `proc(1)` can be used to examine the stopped process.

One of `WATCH` or `RW` must be specified, else the watchpoint facility is not engaged. `RW` overrides `WATCH`. Unrecognized options are silently ignored.

Limitations Sizes of memory blocks allocated by `malloc()` are rounded up to the worst-case alignment size, 8 bytes for 32-bit processes and 16 bytes for 64-bit processes. Accessing the extra space allocated for a memory block is technically a memory violation but is in fact innocuous. Such accesses are not detected by the watchpoint facility of `watchmalloc`.

Interposition of `watchmalloc.so.1` fails innocuously if the target application is statically linked with respect to its `malloc()` functions.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also proc(1), bsdmalloc(3MALLOC), calloc(3C), free(3C), malloc(3C), malloc(3MALLOC), mapmalloc(3MALLOC), memalign(3C), realloc(3C), valloc(3C), libmapmalloc(3LIB), proc(4), attributes(5)

Name wcrtomb – convert a wide-character code to a character (restartable)

Synopsis

```
#include <stdio.h>

size_t wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);
```

Description If *s* is a null pointer, the `wcrtomb()` function is equivalent to the call:

wcrtomb(buf, L'\0', ps)

where *buf* is an internal buffer.

If *s* is not a null pointer, the `wcrtomb()` function determines the number of bytes needed to represent the character that corresponds to the wide-character given by *wc* (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At most `MB_CUR_MAX` bytes are stored. If *wc* is a null wide-character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state. The resulting state described is the initial conversion state.

If *ps* is a null pointer, the `wcrtomb()` function uses its own internal `mbstate_t` object, which is initialized at program startup to the initial conversion state. Otherwise, the `mbstate_t` object pointed to by *ps* is used to completely describe the current conversion state of the associated character sequence. Solaris will behave as if no function defined in the Solaris Reference Manual calls `wcrtomb()`.

The behavior of this function is affected by the `LC_CTYPE` category of the current locale. See `environ(5)`.

Return Values The `wcrtomb()` function returns the number of bytes stored in the array object (including any shift sequences). When *wc* is not a valid wide-character, an encoding error occurs. In this case, the function stores the value of the macros `EILSEQ` in `errno` and returns `(size_t)−1`; the conversion state is undefined.

Errors The `wcrtomb()` function may fail if:

`EINVAL` The *ps* argument points to an object that contains an invalid conversion state.

`EILSEQ` Invalid wide-character code is detected.

Usage If *ps* is not a null pointer, `wcrtomb()` uses the `mbstate_t` object pointed to by *ps* and the function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale. If *ps* is a null pointer, `wcrtomb()` uses its internal `mbstate_t` object and the function is Unsafe in multithreaded applications.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	See NOTES below
Standard	See `standards(5)`.

See Also `mbsinit(3C)`, `setlocale(3C)`, `attributes(5)`, `standards(5)`, `environ(5)`

Name wcscoll, wscoll – wide character string comparison using collating information

Synopsis

```
#include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);

int wscoll(const wchar_t *ws1, const wchar_t *ws2);
```

Description The `wcscoll()` and `wscoll()` functions compare the wide character string pointed to by *ws1* to the wide character string pointed to by *ws2*, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

The `wcscoll()` and `wscoll()` functions do not change the setting of `errno` if successful.

An application wanting to check for error situations should set `errno` to 0 before calling `wcscoll()` or `wscoll()`. If `errno` is non-zero on return, an error has occurred.

Return Values Upon successful completion, `wcscoll()` and `wscoll()` return an integer greater than, equal to, or less than 0, depending upon whether the wide character string pointed to by *ws1* is greater than, equal to, or less than the wide character string pointed to by *ws2*, when both are interpreted as appropriate to the current locale. On error, `wcscoll()` and `wscoll()` may set `errno`, but no return value is reserved to indicate an error.

Errors The `wcscoll()` and `wscoll()` functions may fail if:

`EINVAL` The *ws1* or *ws2* arguments contain wide character codes outside the domain of the collating sequence.

Usage The `wcsxfrm(3C)` and `wcscmp(3C)` functions should be used for sorting large lists.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	`wcscoll()` is Standard
MT-Level	MT-Safe with exceptions

The `wcscoll()` and `wscoll()` functions can be used safely in multithreaded applications as long as `setlocale(3C)` is not being called to change the locale.

See Also `setlocale(3C)`, `wcscmp(3C)`, `wcsxfrm(3C)`, `attributes(5)`, `standards(5)`

Name wcsftime – convert date and time to wide character string

Synopsis

```
#include <wchar.h>
```

XPG4 and SUS

```
size_t wcsftime(wchar_t *wcs, size_t maxsize, const char *format,
     const struct tm *timptr);
```

Default and other standards

```
size_t wcsftime(wchar_t *restrict wcs, size_t maxsize,
     const wchar_t *restrict format,
     const struct tm *restrict timptr);
```

Description The `wcsftime()` function is equivalent to the `strftime(3C)` function, except that:

- The argument *wcs* points to the initial element of an array of wide-characters into which the generated output is to be placed.
- The argument *maxsize* indicates the maximum number of wide-characters to be placed in the output array.
- The argument *format* is a wide-character string and the conversion specifications are replaced by corresponding sequences of wide-characters.
- The return value indicates the number of wide-characters placed in the output array.

If copying takes place between objects that overlap, the behavior is undefined.

Return Values If the total number of resulting wide character codes (including the terminating null wide-character code) is no more than *maxsize*, `wcsftime()` returns the number of wide-character codes placed into the array pointed to by *wcs*, not including the terminating null wide-character code. Otherwise, `0` is returned and the contents of the array are indeterminate.

The `wcfstime()` function uses `malloc(3C)` and should `malloc()` fail, `errno` will be set by `malloc()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `malloc(3C)`, `setlocale(3C)`, `strftime(3C)`, `attributes(5)`, `standards(5)`

Notes The `wcsftime()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Name wcsrtombs – convert a wide-character string to a character string (restartable)

Synopsis

```
#include <wchar.h>

size_t wcsrtombs(char *restrict dst,
     const wchar_t **restrict src, size_t len,
     mbstate_t *restrict ps);
```

Description The `wcsrtombs()` function converts a sequence of wide-characters from the array indirectly pointed to by *src* into a sequence of corresponding characters, beginning in the conversion state described by the object pointed to by *ps*. If *dst* is not a null pointer, the converted characters are then stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null wide-character, which is also stored. Conversion stops earlier in the following cases:

- When a code is reached that does not correspond to a valid character.
- When the next character would exceed the limit of *len* total bytes to be stored in the array pointed to by *dst* (and *dst* is not a null pointer).

Each conversion takes place as if by a call to the `wcrtomb()` function.

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide-character) or the address just past the last wide-character converted (if any). If conversion stopped due to reaching a terminating null wide-character, the resulting state described is the initial conversion state.

If *ps* is a null pointer, the `wcsrtombs()` function uses its own internal `mbstate_t` object, which is initialized at program startup to the initial conversion state. Otherwise, the `mbstate_t` object pointed to by *ps* is used to completely describe the current conversion state of the associated character sequence. Solaris will behave as if no function defined in the Solaris Reference Manual calls `wcsrtombs()`.

The behavior of this function is affected by the `LC_CTYPE` category of the current locale. See `environ(5)`.

Return Values If conversion stops because a code is reached that does not correspond to a valid character, an encoding error occurs. In this case, the `wcsrtombs()` function stores the value of the macro `EILSEQ` in `errno` and returns `(size_t)−1`; the conversion state is undefined. Otherwise, it returns the number of bytes in the resulting character sequence, not including the terminating null (if any).

Errors The `wcsrtombs()` function may fail if:

`EINVAL` The *ps* argument points to an object that contains an invalid conversion state.

`EILSEQ` A wide-character code does not correspond to a valid character.

Usage If *ps* is not a null pointer, `wcsrtombs()` uses the `mbstate_t` object pointed to by *ps* and the function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale. If *ps* is a null pointer, `wcsrtombs()` uses its internal `mbstate_t` object and the function is Unsafe in multithreaded applications.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See NOTES below
Standard	See `standards(5)`.

See Also `mbsinit(3C)`, `setlocale(3C)`, `wcrtomb(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Name wcsstr – find a wide-character substring

Synopsis

```
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

ISO C++

```
#include <wchar.h>

const wchar_t *wcsstr(const wchar_t *ws1, const wchar_t *ws2);

#include <cwchar>

wchar_t *std::wcsstr(wchar_t *ws1, const wchar_t *ws2);
```

Description The `wcsstr()` function locates the first occurrence in the wide-character string pointed to by *ws1* of the sequence of wide-characters (excluding the terminating null wide-character) in the wide-character string pointed to by *ws2*.

Return Values On successful completion, `wcsstr()` returns a pointer to the located wide-character string, or a null pointer if the wide-character string is not found.

If *ws2* points to a wide-character string with zero length, the function returns *ws1*.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `wschr(3C)`, `attributes(5)`, `standards(5)`

Name wcstod, wcstof, wcstold, wstod, watof – convert wide character string to floating-point number

Synopsis

```
#include <wchar.h>

double wcstod(const wchar_t *restrict nptr,
     wchar_t **restrict endptr);

float wcstof(const wchar_t *restrict nptr,
     wchar_t **restrict endptr);

long double wcstold(const wchar_t *restrict nptr,
     wchar_t **restrict endptr);

double wstod(const wchar_t *nptr, wchar_t **endptr);

double watof(wchar_t *nptr);
```

Description The `wcstod()`, `wcstof()`, and `wcstold()` functions convert the initial portion of the wide-character string pointed to by *nptr* to `double`, `float`, and `long double` representation, respectively. They first decompose the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by `iswspace(3C)`)
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide–character code of the input wide-character string.

Then they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- One of INF or INFINITY, or any other wide string equivalent except for case
- One of NAN or NAN(*n-wchar-sequence*$_{opt}$), or any other wide string ignoring case in the NAN part, where:

```
n-wchar-sequence:
    digit
    nondigit
    n-wchar-sequence digit
    n-wchar-sequence nondigit
```

In default mode for `wcstod()`, only decimal, INF/INFINITY, and NAN/NAN(*n-char-sequence*) forms are recognized. In C99/SUSv3 mode, hexadecimal strings are also recognized.

In default mode for `wcstod()`, the *n-char-sequence* in the NAN(*n-char-equence*) form can contain any character except ')' (right parenthesis) or '\0' (null). In C99/SUSv3 mode, the *n-char-sequence* can contain only upper and lower case letters, digits, and '_' (underscore).

The `wcstof()` and `wcstold()` functions always function in C99/SUSv3-conformant mode.

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the radix character (whichever occurs first) is interpreted as a floating constant according to the rules of the C language, except that the radix character is used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A wide-character sequence INF or INFINITY is interpreted as an infinity. A wide-character sequence NAN or NAN(*n-wchar-sequence*$_{opt}$) is interpreted as a quiet NaN. A pointer to the final wide string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has either the decimal or hexadecimal form, the value resulting from the conversion is rounded correctly according to the prevailing floating point rounding direction mode. The conversion also raises floating point inexact, underflow, or overflow exceptions as appropriate.

The radix character is defined in the program's locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period ('.').

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The `wcstod()` function does not change the setting of `errno` if successful.

The `wstod()` function is identical to `wcstod()`.

The `watof`(*str*) function is equivalent to `wstod`(*nptr*`, (wchar_t **)NULL)`.

Return Values Upon successful completion, these functions return the converted value. If no conversion could be performed, `0` is returned.

If the correct value is outside the range of representable values, `±HUGE_VAL`, `±HUGE_VALF`, or `±HUGE_VALL` is returned (according to the sign of the value), a floating point overflow exception is raised, and `errno` is set to `ERANGE`.

If the correct value would cause an underflow, the correctly rounded result (which may be normal, subnormal, or zero) is returned, a floating point underflow exception is raised, and `errno` is set to `ERANGE`.

Errors The `wcstod()` and `wstod()` functions will fail if:

`ERANGE` The value to be returned would cause overflow or underflow.

The `wcstod()` and `wcstod()` functions may fail if:

`EINVAL` No conversion could be performed.

Usage Because 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set `errno` to 0 call `wcstod()`, `wcstof()`, `wcstold()`, or `wstod()`, then check `errno` and if it is non-zero, assume an error has occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	`wcstod()`, `wcstof()`, and `wcstold()` are Standard.
MT-Level	MT-Safe

See Also `iswspace(3C)`, `localeconv(3C)`, `scanf(3C)`, `setlocale(3C)`, `wcstol(3C)`, `attributes(5)`, `standards(5)`

Name wcstoimax, wcstoumax – convert wide-character string to integer type

Synopsis

```
#include <stddef.h>
#include <inttypes.h>

intmax_t wcstoimax(const wchar_t *restrict nptr,
     wchar_t **restrict endptr, int base);

uintmax_t wcstoumax(const wchar_t *restrict nptr,
     wchar_t **restrict endptr, int base);
```

Description These functions are equivalent to the wcstol(3C), wcstoll(3C), wcstoul(3C), and wcstoull(3C) functions, respectively, except that the initial portion of the wide string is converted to intmax_t and uintmax_t representation, respectively.

Return Values These functions return the converted value, if any. If no conversion could be performed, 0 is returned. If the correct value is outside the range of representable values, {INTMAX_MAX}, {INTMAX_MIN}, or {UINTMAX_MAX} is returned (according to the return type and sign of the value), and errno is set to ERANGE.

Errors These functions will fail if:

EINVAL The value of *base* is not supported.

ERANGE The value to be returned is not representable.

These functions may fail if:

EINVAL No conversion could be performed.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also wcstol(3C), wcstoul(3C), attributes(5), standards(5)

Name wcstol, wcstoll, wstol, watol, watoll, watoi – convert wide character string to long integer

Synopsis

```
#include <wchar.h>

long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr,
     int base);

long long wcstoll(const wchar_t *restrict nptr, wchar_t **restrict endptr,
     int base);

#include <widec.h>

long wstol(const wchar_t *nptr, wchar_t **endptr, int base);

long watol(wchar_t *nptr);

long long watoll(wchar_t *nptr);

int watoi(wchar_t *nptr);
```

Description The `wcstol()` and `wcstoll()` functions convert the initial portion of the wide character string pointed to by *nptr* to `long` and `long long` representation, respectively. They first decompose the input string into three parts:

1. an initial, possibly empty, sequence of white-space wide-character codes (as specified by `iswspace(3C)`)
2. a subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. a final wide character string of one or more unrecognised wide character codes, including the terminating null wide-character code of the input wide character string

They then attempt to convert the subject sequence to an integer, and return the result.

If the value of *base* is `0`, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a '+' or '–' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix '0x' or '0X' followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

If the value of *base* is between `2` and `36`, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a '+' or '–' sign, but not including an integer suffix. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is `16`, the wide-character code representations of '0x' or '0X' may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide character string, starting with the first non-white-space wide-character code, that is of the expected

form. The subject sequence contains no wide-character codes if the input wide character string is empty or consists entirely of white-space wide-character code, or if the first non-white-space wide-character code is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is `0`, the sequence of wide-character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between `2` and `36`, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign (-), the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

These functions do not change the setting of `errno` if successful.

Since 0, {`LONG_MIN`} or {`LLONG_MIN`}, and {`LONG_MAX`} or {`LLONG_MAX`} are returned on error and are also valid returns on success, an application wanting to check for error situations should set `errno` to 0, call one of these functions, then check `errno`.

The `wstol()` function is equivalent to `wcstol()`.

The `watol()` function is equivalent to `wstol(`*str*`,(wchar_t **)NULL, 10)`.

The `watoll()` function is the long-long (double long) version of `watol()`.

The `watoi()` function is equivalent to `(int)watol( )`.

Return Values Upon successful completion, these functions return the converted value, if any. If no conversion could be performed, 0 is returned and `errno` may be set to indicate the error. If the correct value is outside the range of representable values, {`LONG_MIN`}, {`LONG_MAX`}, {`LLONG_MIN`}, or {`LLONG_MAX`} is returned (according to the sign of the value), and `errno` is set to `ERANGE`.

Errors These functions will fail if:

`EINVAL` The value of *base* is not supported.

`ERANGE` The value to be returned is not representable.

These functions may fail if:

`EINVAL` No conversion could be performed.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	wcstol() and wcstoll() are Standard.
MT-Level	MT-Safe

See Also iswalpha(3C), iswspace(3C), scanf(3C), wcstod(3C), attributes(5), standards(5)

Notes Truncation from long long to long can take place upon assignment or by an explicit cast.

Name wcstombs – convert a wide-character string to a character string

Synopsis

```
#include <stdlib.h>

size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs, size_t n);
```

Description The `wcstombs()` function converts the sequence of wide-character codes from the array pointed to by *pwcs* into a sequence of characters and stores these characters into the array pointed to by *s*, stopping if a character would exceed the limit of *n* total bytes or if a null byte is stored. Each wide-character code is converted as if by a call to `wctomb(3C)`.

The behavior of this function is affected by the `LC_CTYPE` category of the current locale.

No more than *n* bytes will be modified in the array pointed to by *s*. If copying takes place between objects that overlap, the behavior is undefined. If *s* is a null pointer, `wcstombs()` returns the length required to convert the entire array regardless of the value of *n*, but no values are stored.

Return Values If a wide-character code is encountered that does not correspond to a valid character (of one or more bytes each), `wcstombs()` returns `(size_t)-1`. Otherwise, `wcstombs()` returns the number of bytes stored in the character array, not including any terminating null byte. The array will not be null-terminated if the value returned is *n*.

Errors The `wcstombs()` function may fail if:

`EILSEQ` A wide-character code does not correspond to a valid character.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `mblen(3C)`, `mbstowcs(3C)`, `mbtowc(3C)`, `setlocale(3C)`, `wctomb(3C)`, `attributes(5)`, `standards(5)`

Name wcstoul, wcstoull – convert wide-character string to unsigned long

Synopsis

```
#include <wchar.h>

unsigned long  wcstoul(const wchar_t *restrict nptr,
     wchar_t **restrict endptr, int base);

unsigned long long wcstoull(const wchar_t *restrict nptr,
     wchar_t **restrict endptr, int base);
```

Description The `wcstoul()` and `wcstoull()` functions convert the initial portion of the wide-character string pointed to by *nptr* to `unsigned long` and `unsigned long long` representation, respectively. First they decompose the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by the function `iswspace(3C)`)
2. Asubject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. a final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide character string

They then attempt to convert the subject sequence to an unsigned integer and return the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, an octal constant, or a hexadecimal constant, any of which may be preceded by a ‘+' or a ‘–' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix ‘0', optionally followed by a sequence of the digits ‘0' to ‘7' only. A hexadecimal constant consists of the prefix ‘0x' or ‘0X', followed by a sequence of the decimal digits and letters ‘a' (or ‘A') to ‘f' (or ‘F'), with values 10 to 15, respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a ‘+' or a ‘–' sign, but not including an integer suffix. The letters from ‘a' (or ‘A') to ‘z' (or ‘Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the wide-character codes ‘0x' or ‘0X' may optionally precede the sequence of letters and digits, following the sign, if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first wide-character code that is not a white space and is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white-space wide-character codes, or if the first wide-character code that is not a white space is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of wide-character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The `wcstoul()` function does not change the setting of `errno` if successful.

Since 0, {`ULONG_MAX`}, and {`ULLONG_MAX`} are returned on error and 0 is also a valid return on success, an application wanting to check for error situations should set `errno` to 0, then call `wcstoul()` or `wcstoull()`, then check `errno`.

The `wcstoul()` and `wcstoull()` functions do not change the setting of `errno` if successful.

Return Value Upon successful completion, `wcstoul()` and `wcstoull()` return the converted value, if any. If no conversion could be performed, `0` is returned and `errno` may be set to indicate the error. If the correct value is outside the range of representable values, {`ULONG_MAX`} or {`ULLONG_MAX`}, respectively, is returned and `errno` is set to `ERANGE`.

Errors The `wcstoul()` and `wcstoull()` functions will fail if:

`EINVAL` The value of *base* is not supported.

`ERANGE` The value to be returned is not representable.

The `wcstoul()` and `wcstoull()` functions may fail if:

`EINVAL` No conversion could be performed.

Usage Unlike `wcstod(3C)` and `wcstol(3C)`, `wcstoul()` and `wcstoull()` must always return a non-negative number; using the return value of `wcstoul()` for out-of-range numbers with `wcstoul()` or `wcstoull()` could cause more severe problems than just loss of precision if those numbers can ever be negative.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also isspace(3C), iswalpha(3C), scanf(3C), wcstod(3C), wcstol(3C), attributes(5), standards(5)

Name wcstring, wcscasecmp, wcsncasecmp, wcscat, wscat, wcsncat, wsncat, wcscmp, wscmp, wcsncmp, wsncmp, wcscpy, wscpy, wcsncpy, wsncpy, wcpcpy, wcpncpy, wcsdup, wcslen, wslen, wcsnlen, wcschr, wschr, wcsrchr, wsrchr, windex, wrindex, wcspbrk, wspbrk, wcswcs, wcsspn, wsspn, wcscspn, wscspn, wcstok, wstok – wide-character string operations

Synopsis

```
#include <wchar.h>

int wcscasecmp(const wchar_t *ws1, const wchar_t *ws2);

int wcsncasecmp(const wchar_t ws1*, const wchar_t ws2*, size_t n);

wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);

wchar_t *wcsncat(wchar_t *restrict ws1, const wchar_t *restrict ws2,
     size_t n);

int wcscmp(const wchar_t *ws1, const wchar_t *ws2);

int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);

wchar_t *wcscpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);

wchar_t *wcsncpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
     size_t n);

wchar_t *wcpncpy(wchar_t restrict *ws1, const wchar_t *restrict ws2,
     size_t n);

wchar_t *wcsdup(const wchar_t *s);

size_t wcslen(const wchar_t *ws);

size_t wcsnlen(const wchar_t *ws, size_t maxlen);

wchar_t *wcschr(const wchar_t *ws, wchar_t wc);

wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);

wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);

wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);

size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);

size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
```

XPG4, SUS, SUSv2, SUSv3

```
wchar_t *wcstok(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

Default and other standards

```
wchar_t *wcstok(wchar_t *ws1, const wchar_t *ws2, wchar_t **ptr);
#include <widec.h>

wchar_t *wscat(wchar_t *ws1, const wchar_t *ws2);

wchar_t *wsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);

int wscmp(const wchar_t *ws1, const wchar_t *ws2);
```

```
int wsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

wchar_t *wscpy(wchar_t *ws1, const wchar_t *ws2);

wchar_t *wsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);

size_t wslen(const wchar_t *ws);

wchar_t *wschr(const wchar_t *ws, wchat_t wc);

wchar_t *wsrchr(const wchar_t *ws, wchat_t wc);

wchar_t *wspbrk(const wchar_t *ws1, const wchar_t *ws2);

size_t wsspn(const wchar_t *ws1, const wchar_t *ws2);

size_t wscspn(const wchar_t *ws1, const wchar_t *ws2);

wchar_t *wstok(wchar_t *ws1, const wchar_t *ws2);

wchar_t *windex(const wchar_t *ws, wchar_t wc);

wchar_t *wrindex(const wchar_t *ws, wchar_t wc);
```

ISO C++

```
#include <wchar.h>

const wchar_t *wcschr(const wchar_t *ws, wchar_t wc);

const wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);

const wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);

#include <cwchar>

wchar_t *std::wcschr(wchar_t *ws, wchar_t wc);

wchar_t *std::wcspbrk(wchar_t *ws1, const wchar_t *ws2);

wchar_t *std::wcsrchr(wchar_t *ws, wchar_t wc);
```

Description These functions operate on wide-character strings terminated by `wchar_t` `NULL` characters. During appending or copying, these routines do not check for an overflow condition of the receiving string. In the following, *ws*, *ws1*, and *ws2* point to wide-character strings terminated by a `wchar_t` `NULL`.

`wcscasecmp()`, `wcsncasecmp()` The `wcscasecmp()` function is the wide-character equivalent of the `strcasecmp(3C)` function. It compares the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*, ignoring case differences. It returns 0 if the wide-character strings at *ws1* is equal to *ws2* except for case differences. It returns a positive integer if *ws1* is greater than *ws2* and a negative integer if *ws1* is smaller than *ws2*, ignoring case.

The `wcsncasecmp()` function is the wide-character equivalent of the `strncasecmp(3C)` function. It compares at most *n* wide-characters from the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*, while ignoring differences in case. It returns 0 if the wide-character strings at *ws1* and *ws2*, truncated to at most length *n*, are equal

except for case distinctions. It returns a positive integer if truncated *ws1* is greater than *ws2* and a negative integer if truncated *ws1* is smaller than *ws2*, ignoring case.

`wcscat(),wscat()` The `wcscat()` and `wscat()` functions append a copy of the wide-character string pointed to by *ws2* (including the terminating null wide-character code) to the end of the wide-character string pointed to by *ws1*. The initial wide-character code of *ws2* overwrites the null wide-character code at the end of *ws1*. If copying takes place between objects that overlap, the behavior is undefined. Both functions return *s1*; no return value is reserved to indicate an error.

`wcsncat(),wsncat()` The `wcsncat()` and `wsncat()` functions append not more than *n* wide-character codes (a null wide-character code and wide-character codes that follow it are not appended) from the array pointed to by *ws2* to the end of the wide-character string pointed to by *ws1*. The initial wide-character code of *ws2* overwrites the null wide-character code at the end of *ws1*. A terminating null wide-character code is always appended to the result. Both functions return *ws1*; no return value is reserved to indicate an error.

`wcscmp(),wscmp()` The `wcscmp()` and `wscmp()` functions compare the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared. Upon completion, both functions return an integer greater than, equal to, or less than zero, if the wide-character string pointed to by *ws1* is greater than, equal to, or less than the wide-character string pointed to by *ws2*.

`wcsncmp(),wsncmp()` The `wcsncmp()` and `wsncmp()` functions compare not more than *n* wide-character codes (wide-character codes that follow a null wide character code are not compared) from the array pointed to by *ws1* to the array pointed to by *ws2*. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared. Upon successful completion, both functions return an integer greater than, equal to, or less than zero, if the possibly null-terminated array pointed to by *ws1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *ws2*.

`wcscpy(),wscpy(), wcpcpy()` The `wcscpy()`, `wscpy()`, and `wcpcpy()` functions copy the wide-character string pointed to by *ws2* (including the terminating null wide-character code) into the array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is undefined.

The `wcscpy()` and `wscpy()` functions return *ws1*. The `wcpcpy()` function returns a pointer to the terminating null wide-character code copied into *ws1*.

`wcsncpy(),wsncpy(), wcpncpy()` The `wcsncpy()`, `wsncpy()`, and `wcpncpy()` functions copy not more than *n* wide-character codes (wide-character codes that follow a null wide character code are not copied) from the array pointed to by *ws2* to the array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by *ws2* is a wide-character string that is shorter than *n* wide-character codes, null wide-character codes are appended to the copy in the array pointed to by *ws1*, until

a total *n* wide-character codes are written. The `wcsncpy()` and `wsncpy()` functions return *ws1*. The `wcpncpy()` function returns a pointer to the last wide character written.

`wcsdup()` The `wcsdup()` function is the wide-character equivalent of the `strdup(3C)` function. It returns a pointer to a new wide-character string whose initial contents is a duplicate of the wide-character string pointed to by *s*. Memory for the new wide-character string is allocated with `malloc(3C)` and can be freed with a call to `free(3C)`. A null pointer is returned and *errno* set to `ENOMEM` if there is insufficient memory available for the duplicate string.

`wcslen(),wslen(), wcsnlen()` The `wcslen()` and `wslen()` functions compute the number of wide-character codes in the wide-character string to which *ws* points, not including the terminating null wide-character code. Both functions return *ws*; no return value is reserved to indicate an error.

The `wcsnlen()` is the wide-character equivalent of the `strnlen(3C)` function. It returns the number of wide-characters in the string pointed to by *ws*, not including the terminating null wide-character code but at most *maxlen*, while never looking beyond the first *maxlen* characters. It returns *maxlen* if there is no terminating null wide-character code among the first *maxlen* wide characters pointed to by *ws*.

`wcschr(),wschr()` The `wcschr()` and `wschr()` functions locate the first occurrence of *wc* in the wide-character string pointed to by *ws*. The value of *wc* must be a character representable as a type `wchar_t` and must be a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string. Upon completion, both functions return a pointer to the wide-character code, or a null pointer if the wide-character code is not found.

`wcsrchr(),wsrchr()` The `wcsrchr()` and `wsrchr()` functions locate the last occurrence of *wc* in the wide-character string pointed to by *ws*. The value of *wc* must be a character representable as a type `wchar_t` and must be a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string. Upon successful completion, both functions return a pointer to the wide-character code, or a null pointer if *wc* does not occur in the wide-character string.

`windex(),wrindex()` The `windex()` and `wrindex()` functions behave the same as `wschr()` and `wsrchr()`, respectively.

`wcspbrk(),wspbrk()` The `wcspbrk()` and `wspbrk()` functions locate the first occurrence in the wide character string pointed to by *ws1* of any wide-character code from the wide-character string pointed to by *ws2*. Upon successful completion, the function returns a pointer to the wide-character code, or a null pointer if no wide-character code from *ws2* occurs in *ws1*.

`wcswcs()` The `wcswcs()` function locates the first occurrence in the wide-character string pointed to by *ws1* of the sequence of wide-character codes (excluding the terminating null wide-character code) in the wide-character string pointed to by *ws2*. Upon successful completion, the function returns a pointer to the located wide-character string, or a null pointer if the wide-character string is not found. If *ws2* points to a wide-character string with zero length, the function returns *ws1*.

`wcsspn(),wsspn()` The `wcsspn()` and `wsspn()` functions compute the length of the maximum initial segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character codes from the wide-character string pointed to by *ws2*. Both functions return the length *ws1*; no return value is reserved to indicate an error.

`wcscspn(),wscspn()` The `wcscspn()` and `wscspn()` functions compute the length of the maximum initial segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character codes *not* from the wide-character string pointed to by *ws2*. Both functions return the length of the initial substring of *ws1*; no return value is reserved to indicate an error.

`wcstok(),wstok()` A sequence of calls to the `wcstok()` and `wstok()` functions break the wide-character string pointed to by *ws1* into a sequence of tokens, each of which is delimited by a wide-character code from the wide-character string pointed to by *ws2*.

Default and other standards The third argument points to a caller-provided `wchar_t` pointer into which the `wcstok()` function stores information necessary for it to continue scanning the same wide-character string. This argument is not available with the XPG4 and SUS versions of `wcstok()`, nor is it available with the `wstok()` function. See `standards(5)`.

The first call in the sequence has *ws1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *ws2* may be different from call to call.

The first call in the sequence searches the wide-character string pointed to by *ws1* for the first wide-character code that is *not* contained in the current separator string pointed to by *ws2*. If no such wide-character code is found, then there are no tokens in the wide-character string pointed to by *ws1*, and `wcstok()` and `wstok()` return a null pointer. If such a wide-character code is found, it is the start of the first token.

The `wcstok()` and `wstok()` functions then search from that point for a wide-character code that *is* contained in the current separator string. If no such wide-character code is found, the current token extends to the end of the wide-character string pointed to by *ws1*, and subsequent searches for a token will return a null pointer. If such a wide-character code is found, it is overwritten by a null wide character, which terminates the current token. The `wcstok()` and `wstok()` functions save a pointer to the following wide-character code, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Upon successful completion, both functions return a pointer to the first wide-character code of a token. Otherwise, if there is no token, a null pointer is returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See below.

For `wcscat()`, `wcsncat()`, `wcscmp()`, `wcsncmp()`, `wcscpy()`, `wcsncpy()`, `wcslen()`, `wcschr()`, `wcsrchr()`, `wcspbrk()`, `wcswcs()`, `wcsspn()`, `wcscspn()`, and `wcstok()`, see `standards(5)`.

See Also `malloc(3C)`, `string(3C)`, `wcswidth(3C)`, `wcwidth(3C)`, `attributes(5)`, `standards(5)`

Name wcswidth – number of column positions of a wide-character string

Synopsis

```
#include <wchar.h>

int wcswidth(const wchar_t *pwcs, size_t n);
```

Description The `wcswidth()` function determines the number of column positions required for *n* wide-character codes (or fewer than *n* wide-character codes if a null wide-character code is encountered before *n* wide-character codes are exhausted) in the string pointed to by *pwcs*.

Return Values The `wcswidth()` function either returns `0` (if *pwcs* points to a null wide-character code), or returns the number of column positions to be occupied by the wide-character string pointed to by *pwcs*, or returns `−1` (if any of the first *n* wide-character codes in the wide-character string pointed to by *pwcs* is not a printing wide-character code).

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `setlocale(3C)`, `wcwidth(3C)`, `attributes(5)`, `standards(5)`

Name wcsxfrm, wsxfrm – wide character string transformation

Synopsis

```
#include <wchar.h>

size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);

size_t wsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

Description The `wcsxfrm()` and `wcsxfrm()` functions transform the wide character string pointed to by *ws2* and place the resulting wide character string into the array pointed to by *ws1*. The transformation is such that if either the `wcscmp(3C)` or `wscmp(3C)` functions are applied to two transformed wide strings, they return a value greater than, equal to, or less than 0, corresponding to the result of the `wcscoll(3C)` or `wscoll(3C)` function applied to the same two original wide character strings. No more than *n* wide-character codes are placed into the resulting array pointed to by *ws1*, including the terminating null wide-character code. If *n* is 0, *ws1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

The `wcsxfrm()` and `wsxfrm()` functions do not change the setting of `errno` if successful.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to 0, then call `wcsxfrm()` or `wsxfrm()`, then check `errno`.

Return Values The `wcsxfrm()` and `wsxfrm()` functions return the length of the transformed wide character string (not including the terminating null wide-character code). If the value returned is *n* or more, the contents of the array pointed to by *ws1* are indeterminate.

On error, `wcsxfrm()` and `wsxfrm()` may set `errno` but no return value is reserved to indicate an error.

Errors The `wcsxfrm()` and `wsxfrm()` functions may fail if:

`EINVAL` The wide character string pointed to by *ws2* contains wide-character codes outside the domain of the collating sequence.

Usage The transformation function is such that two transformed wide character strings can be ordered by the `wcscmp()` or `wscmp()` functions as appropriate to collating sequence information in the program's locale (category `LC_COLLATE`).

The fact that when *n* is 0, *ws1* is permitted to be a null pointer, is useful to determine the size of the *ws1* array prior to making the transformation.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	`wcsxfrm()` is Standard

MT-Level	MT-Safe with exceptions

The `wcsxfrm()` and `wsxfrm()` functions can be used safely in multithreaded applications as long as `setlocale(3C)` is not being called to change the locale.

See Also `setlocale(3C)`, `wcscmp(3C)`, `wcscoll(3C)`, `wscmp(3C)`, `wscoll(3C)`, `attributes(5)`, `standards(5)`

Name wctob – wide-character to single-byte conversion

Synopsis

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

int wctob(wint_t c);
```

Description The `wctob()` function determines whether *c* corresponds to a member of the extended character set whose character representation is a single byte when in the initial shift state.

The behavior of this function is affected by the `LC_CTYPE` category of the current locale. See `environ(5)`

Return Values The `wctob()` function returns `EOF` if *c* does not correspond to a character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `btowc(3C)`, `setlocale(3C)`, `attributes(5)`, `environ(5)`, `standards(5)`

Notes The `wctob()` function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.

Name wctomb – convert a wide-character code to a character

Synopsis

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wchar);
```

Description The `wctomb()` function determines the number of bytes needed to represent the character corresponding to the wide-character code whose value is *wchar*. It stores the character representation (possibly multiple bytes) in the array object pointed to by *s* (if *s* is not a null pointer). At most `MB_CUR_MAX` bytes are stored.

A call with *s* as a null pointer causes this function to return `0`. The behavior of this function is affected by the `LC_CTYPE` category of the current locale.

Return Values If *s* is a null pointer, `wctomb()` returns `0` value. If *s* is not a null pointer, `wctomb()` returns `–1` if the value of *wchar* does not correspond to a valid character, or returns the number of bytes that constitute the character corresponding to the value of *wchar*.

In no case will the value returned be greater than the value of the `MB_CUR_MAX` macro.

Errors No errors are defined.

Usage The `wctomb()` function can be used safely in a multithreaded application, as long as `setlocale(3C)` is not being called to change the locale.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe with exceptions

See Also `mblen(3C)`, `mbstowcs(3C)`, `mbtowc(3C)`, `setlocale(3C)`, `wcstombs(3C)`, `attributes(5)`, `standards(5)`

Name wctrans – define character mapping

Synopsis

```
#include <wctype.h>

wctrans_t wctrans(const char *charclass);
```

Description The `wctrans()` function is defined for valid character mapping names identified in the current locale. The *charclass* is a string identifying a generic character mapping name for which codeset-specific information is required. The following character mapping names are defined in all locales – "tolower" and "toupper".

The function returns a value of type `wctrans_t`, which can be used as the second argument to subsequent calls of `towctrans(3C)`. The `wctrans()` function determines values of `wctrans_t` according to the rules of the coded character set defined by character mapping information in the program's locale (category `LC_CTYPE`). The values returned by `wctrans()` are valid until a call to `setlocale(3C)` that modifies the category `LC_CTYPE`.

Return Values The `wctrans()` function returns `0` if the given character mapping name is not valid for the current locale (category `LC_CTYPE`), otherwise it returns a non-zero object of type `wctrans_t` that can be used in calls to `towctrans(3C)`.

Errors The `wctrans()` function may fail if:

`EINVAL` The character mapping name pointed to by *charclass* is not valid in the current locale.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `setlocale(3C)`, `towctrans(3C)`, `attributes(5)`, `standards(5)`

Name wctype – define character class

Synopsis

```
#include <wchar.h>

wctype_t wctype(const char *charclass);
```

Description The `wctype()` function is defined for valid character class names as defined in the current locale. The *charclass* is a string identifying a generic character class for which codeset-specific type information is required. The following character class names are defined in all locales:

alnum	alpha	blank
cntrl	digit	graph
lower	print	punct
space	upper	xdigit

Additional character class names defined in the locale definition file (category `LC_CTYPE`) can also be specified.

The function returns a value of type `wctype_t`, which can be used as the second argument to subsequent calls of `iswctype(3C)`. `wctype()` determines values of `wctype_t` according to the rules of the coded character set defined by character type information in the program's locale (category `LC_CTYPE`). The values returned by `wctype()` are valid until a call to `setlocale(3C)` that modifies the category `LC_CTYPE`.

Return Values The `wctype()` function returns `0` if the given character class name is not valid for the current locale (category `LC_CTYPE`); otherwise it returns an object of type `wctype_t` that can be used in calls to `iswctype()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Standard
MT-Level	MT-Safe with exceptions

See Also `iswctype(3C)`, `setlocale(3C)`, `attributes(5)`, `standards(5)`

Name wcwidth – number of column positions of a wide-character code

Synopsis

```
#include <wchar.h>

int wcwidth(wchar_t wc);
```

Description The `wcwidth()` function determines the number of column positions required for the wide character *wc*. The value of *wc* must be a character representable as a `wchar_t`, and must be a wide-character code corresponding to a valid character in the current locale.

Return Values The `wcwidth()` function either returns `0` (if *wc* is a null wide-character code), or returns the number of column positions to be occupied by the wide-character code *wc*, or returns `−1` (if *wc* does not correspond to a printing wide-character code).

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Committed
MT-Level	MT-Safe with exceptions
Standard	See `standards(5)`.

See Also `setlocale(3C)`, `wcswidth(3C)`, `attributes(5)`, `standards(5)`

Name wmemchr – find a wide-character in memory

Synopsis

```
#include <wchar.h>

wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);
```

ISO C++

```
#include <wchar.h>

const wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);

#include <cwchar>

wchar_t *std::wmemchr(wchar_t *ws, wchar_t wc, size_t n);
```

Description The `wmemchr()` function locates the first occurrence of *wc* in the initial *n* wide-characters of the object pointed to be *ws*. This function is not affected by locale and all `wchar_t` values are treated identically. The null wide-character and `wchar_t` values not corresponding to valid characters are not treated specially.

If *n* is 0, *ws* must be a valid pointer and the function behaves as if no valid occurrence of *wc* is found.

Return Values The `wmemchr()` function returns a pointer to the located wide-character, or a null pointer if the wide-character does not occur in the object.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `wmemcmp(3C)`, `wmemcpy(3C)`, `wmemmove(3C)`, `wmemset(3C)`, `attributes(5)`, `standards(5)`

Name wmemcmp – compare wide-characters in memory

Synopsis

```
#include <wchar.h>

int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

Description The `wmemcmp()` function compares the first *n* wide-characters of the object pointed to by *ws1* to the first *n* wide-characters of the object pointed to by *ws2*. This function is not affected by locale and all `wchar_t` values are treated identically. The null wide-character and `wchar_t` values not corresponding to valid characters are not treated specially.

If *n* is zero, *ws1* and *ws2* must be a valid pointers and the function behaves as if the two objects compare equal.

Return Values The `wmemcmp()` function returns an integer greater than, equal to, or less than 0, accordingly as the object pointed to by *ws1* is greater than, equal to, or less than the object pointed to by *ws2*.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `wmemchr(3C)`, `wmemcpy(3C)`, `wmemmove(3C)`, `wmemset(3C)`, `attributes(5)`, `standards(5)`

Name wmemcpy – copy wide-characters in memory

Synopsis

```
#include <wchar.h>

wchar_t *wmemcpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

Description The `wmemcpy()` function copies *n* wide-characters from the object pointed to by *ws2* to the object pointed to be *ws1*. This function is not affected by locale and all `wchar_t` values are treated identically. The null wide-character and `wchar_t` values not corresponding to valid characters are not treated specially.

If *n* is zero, *ws1* and *ws2* must be a valid pointers, and the function copies zero wide-characters.

Return Values The `wmemcpy()` function returns the value of *ws1*.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `wmemchr(3C)`, `wmemcmp(3C)`, `wmemmove(3C)`, `wmemset(3C)`, `attributes(5)`, `standards(5)`

Name wmemmove – copy wide-characters in memory with overlapping areas

Synopsis

```
#include <wchar.h>

wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

Description The `wmemmove()` function copies *n* wide-characters from the object pointed to by *ws2* to the object pointed to by *ws1*. Copying takes place as if the *n* wide-characters from the object pointed to by *ws2* are first copied into a temporary array of *n* wide-characters that does not overlap the objects pointed to by *ws1* or *ws2*, and then the *n* wide-characters from the temporary array are copied into the object pointed to by *ws1*.

This function is not affected by locale and all `wchar_t` values are treated identically. The null wide-character and `wchar_t` values not corresponding to valid characters are not treated specially.

If *n* is 0, *ws1* and *ws2* must be a valid pointers, and the function copies zero wide-characters.

Return Values The `wmemmove()` function returns the value of *ws1*.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `wmemchr(3C)`, `wmemcmp(3C)`, `wmemcpy(3C)`, `wmemset(3C)`, `attributes(5)`, `standards(5)`

Name wmemset – set wide-characters in memory

Synopsis

```
#include <wchar.h>

wchar_t *wmemset(wchar_t *ws, wchar_t wc, size_t n);
```

Description The `wmemset()` function copies the value of *wc* into each of the first *n* wide-characters of the object pointed to by *ws*. This function is not affected by locale and all `wchar_t` values are treated identically. The null wide-character and `wchar_t` values not corresponding to valid characters are not treated specially.

If *n* is 0, *ws* must be a valid pointer and the function copies zero wide-characters.

Return Values The `wmemset()` functions returns the value of *ws*.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `wmemchr(3C)`, `wmemcmp(3C)`, `wmemcpy(3C)`, `wmemmove(3C)`, `attributes(5)`, `standards(5)`

Name wordexp, wordfree – perform word expansions

Synopsis

```
#include <wordexp.h>

int wordexp(const char *restrict words, wordexp_t *restrict pwordexp,
     int flags);

void wordfree(wordexp_t *pwordexp);
```

Description The `wordexp()` function performs word expansions, subject to quoting, and places the list of expanded words into the structure pointed to by *pwordexp*.

The `wordfree()` function frees any memory allocated by `wordexp()` associated with *pwordexp*.

words Argument The *words* argument is a pointer to a string containing one or more words to be expanded. The expansions will be the same as would be performed by the shell if *words* were the part of a command line representing the arguments to a utility. Therefore, *words* must not contain an unquoted NEWLINE or any of the unquoted shell special characters:

```
| & ; < >
```

except in the context of command substitution. It also must not contain unquoted parentheses or braces, except in the context of command or variable substitution. If the argument *words* contains an unquoted comment character (number sign) that is the beginning of a token, `wordexp()` may treat the comment character as a regular character, or may interpret it as a comment indicator and ignore the remainder of *words*.

pwordexp Argument The structure type `wordexp_t` is defined in the header <`wordexp.h`> and includes at least the following members:

`size_t we_wordc`	Count of words matched by *words*.
`char **we_wordv`	Pointer to list of expanded words.
`size_t we_offs`	Slots to reserve at the beginning of *pwordexp*–>`we_wordv`.

The `wordexp()` function stores the number of generated words into *pwordexp*–>`we_wordc` and a pointer to a list of pointers to words in *pwordexp*–>`we_wordv`. Each individual field created during field splitting is a separate word in the *pwordexp*–>`we_wordv` list. The words are in order. The first pointer after the last word pointer will be a null pointer.

It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. The `wordexp()` function allocates other space as needed, including memory pointed to by *pwordexp*–>`we_wordv`. The `wordfree()` function frees any memory associated with *pwordexp* from a previous call to `wordexp()`.

flags Argument

The *flags* argument is used to control the behavior of `wordexp()`. The value of *flags* is the bitwise inclusive OR of zero or more of the following constants, which are defined in `<wordexp.h>`:

`WRDE_APPEND`	Append words generated to the ones from a previous call to `wordexp()`.
`WRDE_DOOFFS`	Make use of *pwordexp*–>`we_offs`. If this flag is set, *pwordexp*–>`we_offs` is used to specify how many `NULL` pointers to add to the beginning of *pwordexp*–>`we_wordv`. In other words, *pwordexp*–>`we_wordv` will point to *pwordexp*–>`we_offs` `NULL` pointers, followed by *pwordexp*–>`we_wordc` word pointers, followed by a `NULL` pointer.
`WRDE_NOCMD`	Fail if command substitution is requested.
`WRDE_REUSE`	The *pwordexp* argument was passed to a previous successful call to `wordexp()`, and has not been passed to `wordfree()`. The result will be the same as if the application had called `wordfree()` and then called `wordexp()` without `WRDE_REUSE`.
`WRDE_SHOWERR`	Do not redirect `stderr` to `/dev/null`.
`WRDE_UNDEF`	Report error on an attempt to expand an undefined shell variable.

The `WRDE_APPEND` flag can be used to append a new set of words to those generated by a previous call to `wordexp()`. The following rules apply when two or more calls to `wordexp()` are made with the same value of *pwordexp* and without intervening calls to `wordfree()`:

1. The first such call must not set `WRDE_APPEND`. All subsequent calls must set it.
2. All of the calls must set `WRDE_DOOFFS`, or all must not set it.
3. After the second and each subsequent call, *pwordexp*–>`we_wordv` will point to a list containing the following:
   a. zero or more `NULL` pointers, as specified by `WRDE_DOOFFS` and *pwordexp*–>`we_offs`.
   b. pointers to the words that were in the *pwordexp*–>`we_wordv` list before the call, in the same order as before.
   c. pointers to the new words generated by the latest call, in the specified order.
4. The count returned in *pwordexp*–>`we_wordc` will be the total number of words from all of the calls.
5. The application can change any of the fields after a call to `wordexp()`, but if it does it must reset them to the original value before a subsequent call, using the same *pwordexp* value, to `wordfree()` or `wordexp()` with the `WRDE_APPEND` or `WRDE_REUSE` flag.

If *words* contains an unquoted:

NEWLINE `| & ; < > ( ) { }`

in an inappropriate context, `wordexp()` will fail, and the number of expanded words will be zero.

Unless `WRDE_SHOWERR` is set in *flags*, `wordexp()` will redirect `stderr` to `/dev/null` for any utilities executed as a result of command substitution while expanding *words*.

If `WRDE_SHOWERR` is set, `wordexp()` may write messages to *stderr* if syntax errors are detected while expanding *words*. If `WRDE_DOOFFS` is set, then *pwordexp*–> `we_offs` must have the same value for each `wordexp()` call and `wordfree()` call using a given *pwordexp*.

The following constants are defined as error return values:

`WRDE_BADCHAR` One of the unquoted characters:

NEWLINE `| & ; < > ( ) { }`

appears in *words* in an inappropriate context.

`WRDE_BADVAL` Reference to undefined shell variable when `WRDE_UNDEF` is set in *flags*.

`WRDE_CMDSUB` Command substitution requested when `WRDE_NOCMD` was set in flags.

`WRDE_NOSPACE` Attempt to allocate memory failed.

`WRDE_SYNTAX` Shell syntax error, such as unbalanced parentheses or unterminated string.

Return Values On successful completion, `wordexp()` returns `0`.

Otherwise, a non-zero value as described in `<wordexp.h>` is returned to indicate an error. If `wordexp()` returns the value `WRDE_NOSPACE`, then *pwordexp*–>`we_wordc` and *pwordexp*–>`we_wordv` will be updated to reflect any words that were successfully expanded. In other cases, they will not be modified.

The `wordfree()` function returns no value.

Errors No errors are defined.

Usage This function is intended to be used by an application that wants to do all of the shell's expansions on a word or words obtained from a user. For example, if the application prompts for a filename (or list of filenames) and then uses `wordexp()` to process the input, the user could respond with anything that would be valid as input to the shell.

The `WRDE_NOCMD` flag is provided for applications that, for security or other reasons, want to prevent a user from executing shell command. Disallowing unquoted shell special characters also prevents unwanted side effects such as executing a command or writing a file.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See `standards(5)`.

See Also `fnmatch(3C)`, `glob(3C)`, `attributes(5)`, `standards(5)`

Name wsprintf – formatted output conversion

Synopsis

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

int wsprintf(wchar_t *s, const char *format, /* arg */ ... ););
```

Description The `wsprintf()` function outputs a Process Code string ending with a Process Code (`wchar_t`) null character. It is the user's responsibility to allocate enough space for this `wchar_t` string.

This returns the number of Process Code characters (excluding the null terminator) that have been written. The conversion specifications and behavior of `wsprintf()` are the same as the regular `sprintf(3C)` function except that the result is a Process Code string for `wsprintf( )`, and on Extended Unix Code (EUC) character string for `sprintf()`.

Return Values Upon successful completion, `wsprintf()` returns the number of characters printed. Otherwise, a negative value is returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `wsscanf(3C)`, `printf(3C)`, `scanf(3C)`, `sprintf(3C)`, `attributes(5)`

Name wsscanf – formatted input conversion

Synopsis

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

int wsscanf(wchar_t *s, const char *format, /* pointer */ ... );
```

Description The `wsscanf()` function reads Process Code characters from the Process Code string *s*, interprets them according to the *format*, and stores the results in its arguments. It expects, as arguments, a control string *format*, and a set of *pointer* arguments indicating where the converted input should be stored. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

The conversion specifications and behavior of `wsscanf()` are the same as the regular `sscanf(3C)` function except that the source is a Process Code string for `wsscanf()` and on Extended Unix Code (EUC) character string for `sscanf(3C)`.

Return Values Upon successful completion, `wsscanf()` returns the number of characters matched. Otherwise, it returns a negative value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `wsprintf(3C)`, `printf(3C)`, `scanf(3C)`, `attributes(5)`

Name wstring, wscasecmp, wsncasecmp, wsdup, wscol – Process Code string operations

Synopsis

```
#include <widec.h>

int wscasecmp(const wchar_t *s1, const wchar_t *s2);

int wsncasecmp(const wchar_t *s1, const wchar_t *s2, int n);

wchar_t *wsdup(const wchar_t *s);

int wscol(const wchar_t *s);
```

Description These functions operate on Process Code strings terminated by `wchar_t` null characters. During appending or copying, these routines do not check for an overflow condition of the receiving string. In the following, *s*, *s1*, and *s2* point to Process Code strings terminated by a `wchar_t` null.

wscasecmp(), wsncasecmp() The `wscasecmp()` function compares its arguments, ignoring case, and returns an integer greater than, equal to, or less than 0, depending upon whether *s1* is lexicographically greater than, equal to, or less than *s2*. It makes the same comparison but compares at most *n* Process Code characters. The four Extended Unix Code (EUC) codesets are ordered from lowest to highest as 0, 2, 3, 1 when characters from different codesets are compared.

wsdup() The `wsdup()` function returns a pointer to a new Process Code string, which is a duplicate of the string pointed to by *s*. The space for the new string is obtained using `malloc(3C)`. If the new string cannot be created, a null pointer is returned.

wscol() The `wscol()` function returns the screen display width (in columns) of the Process Code string *s*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `malloc(3C)`, `string(3C)`, `wcstring(3C)`, `attributes(5)`