

Programmer's Reference Manual

Commands(1)

Introduction

Programmer's Reference Manual

intro_prm1(1)

NAME

intro_prm1 - introduction to programming commands

DESCRIPTION

This section describes the programming commands in alphabetical order.

SYNOPSIS

Unless otherwise noted, the commands accept options and other arguments according to the following syntax:

name [-option(s)] [*cmdarg*(s)] [*nvarg*(s)]

The different syntax elements have the following meaning:

[☐] Selectable option or argument

name Name of an executable file (command)

option

(always with a preceding "-"): noargletter(s) or: argletterERRORoptarg

noargletter

One or more single letters each representing an option *without* an argument (e.g. -i or -dna).

argletter

A single letter representing an option requiring an argument (*optarg*), e.g. t in -tERRORfile.

optarg

An argument (character string) satisfying preceding argletter (e.g. file in -tERRORfile).

cmdarg

Variable argument(s) (character string) for a command, e.g. a pathname, directory, target device etc.;
 "-" indicates the standard input. Examples:

devfree key [*device* [...]]

or: colltbl [*file* | -]

nvarg

One or more arguments that cannot be modified, e.g. on☐|☐off, demand

... Repetition: One or more options or arguments can be specified several times. Example: [*attribute* ...]

| Alternative: Only one option or one argument of several commands can be specified. Example: -v | -p
 | -a

{☐} Grouping: At least one option or argument from this group must be specified. Example: {*special* |
mount_point}

VARIABLES

Throughout the manual pages there are references to *TMPDIR*, *INCDIR*, and *LIBDIR*. These represent directory names whose value is specified on each manual page as necessary. For example, *TMPDIR* might refer to */var/tmp*. These are not environment variables and cannot be set. (There is an environment variable called *TMPDIR* which can be set [see [tmpnam\(3S\)](#)].) There are also references to *LIBPATH*, the default search path of the link editor and other tools.

DIAGNOSTICS

Upon termination, each command returns two bytes of status, one supplied by the system and giving the cause for termination, and (in the case of normal termination) one supplied by the program. The former byte is 0 for normal termination; the latter is customarily 0 for successful execution and non-zero to indicate troubles such as erroneous parameters, or bad or inaccessible data. It is called variously exit code, exit status, or return code, and is described only where special conventions are involved [see `wait(2)` and `exit(2)`].

WARNINGS

Some commands produce unexpected results when processing files containing null characters. These commands often treat text input lines as strings and therefore become confused upon encountering a null character (the string terminator) within a line.

DIRECTORIES*INCDIR*

Standard directory for `#include` files (usually `/usr/include`)

LIBDIR

Usually `/usr/ccs/lib`

TMPDIR

Temporary directory, which defaults to `/var/tmp`

SEE ALSO

`getopts(1)`, `exit(2)`, `wait(2)`, `getopt(3C)`.

admin(1) - lprof(1)**NAME**

`admin` - create and administer SCCS files

SYNOPSIS

`admin [ option... ] file...`

DESCRIPTION

`admin` is used to create new SCCS files and to change parameters of existing ones. Note that SCCS filenames must begin with the characters `s.` If a named file does not exist, it is created and its parameters are initialized according to the specified keyletter arguments. Parameters not initialized by a keyletter argument are assigned a default value. If the file does exist, parameters corresponding to specified keyletter arguments are changed, and other parameters are left unchanged.

If a directory is named, `admin` behaves as though each file in the directory were specified as a named file. Unreadable files and files whose names do not begin with `s.` are ignored, without comment. If the filename is specified as `-`, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are ignored without comment.

OPTIONS

The options (keyletters) and their arguments are listed below. Each argument is explained as if only one named file were to be processed, because the effect of each argument applies independently to each named file.

`-n` indicates that a new SCCS file is to be created. If `-i` is not specified, the file contains only control information.

`-i[name]`

specifies the name of a file from which the text for a new SCCS file is to be taken.

The text constitutes the first delta of the file (see `-r` keyletter for the delta numbering scheme). If the `-i` keyletter is used, but no filename is specified, the text is expected on the standard input. If `-i` is specified, then only one SCCS file can be created. Note that the keyletter `-i` implies the keyletter `-n`.

`-r version`

specifies the *version* (release number) into which the initial delta is inserted.

This keyletter may be used only if the `-i` keyletter is also used. If the `-r` keyletter is not used, the initial delta is inserted into release 1. The level of the initial delta is always 1 (by default initial deltas are named 1.1).

`-t[name]`

specifies the name of a file from which descriptive text for the SCCS file is to be taken.

In the case of existing SCCS files, the following apply: a `-t` keyletter without a filename causes removal of the descriptive text that is currently in the SCCS file; a `-t` keyletter with a filename causes text (if any) in the named file to replace the descriptive text which is currently in the SCCS file.

`-f flag`

specifies the value for a flag to be given to the SCCS file.

Several `-f` keyletters may be used in a single `admin` command line. The allowable *flags* and their values are:

`b` Allows use of the `-b` keyletter with a `get` command to create branch deltas.

`cceil`

Specifies the highest version (i.e. ceiling) which the file can be given. *ceil* must lie between 1 and 9999 (inclusive). The default value if `c` is not specified is 9999.

`ffloor`

Specifies the number of the first version (i.e. floor). *floor* must lie between 0 and 10000 (exclusive). The default value if `f` is not specified is 1.

`dSID` The default delta number (SID) to be used by a `get` command.

`i[str]`

Causes the `No id keywords (ge6)` message issued by `get` or `delta` to be treated as a fatal error. The default is that this message is only issued as a warning. The message is issued if no SCCS identification keywords [see `get(1)`] are found in the text retrieved or stored in the SCCS file. If a value is supplied, the keywords must exactly match the given string. The string must contain exactly one keyword, and no newlines.

`j` Allows multiple concurrent `get` commands for editing on the same SID of an SCCS file. This flag allows multiple concurrent updates to the same version of the SCCS file.

`l list`

A *list* of versions (release numbers) to which deltas can no longer be made. `get -e` against one of these "locked" releases fails. The *list* has the following syntax:

`<list> ::= <range> | <list> , <range>`

`<range> ::= <releasenum> | a`

The character `a` in the *list* indicates that all releases for the named SCCS file are to be locked.

`n` Causes `delta` to create a null delta in each of those releases (if any) being skipped when a delta is made in a new release, e.g. in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped. These null deltas serve as anchor points so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be non-existent in the SCCS file, preventing branch deltas from being created from them in the future.

`qtext`

User-definable text which will be substituted for every occurrence of the `%Q%` keyword in the SCCS file retrieved by `get`.

- mmod** The module name of the SCCS file which will be substituted for every occurrence of the `%M%` keyword in SCCS file text retrieved by `get`. If the `m` keyletter is not specified, the value assigned is the name of the SCCS file with the leading `s.` removed.
- ttype** Specifies the *type* of module in the SCCS file substituted for every occurrence of `%Y%` keyword in SCCS file text retrieved by `get`.
- v[pgm]** Causes `delta` to prompt for Modification Request (MR) numbers as the reason for creating a delta.
The optional argument specifies the name of an MR number validity checking program [see `delta(1)`]. This program will receive as arguments the module name, the value of the *type* flag (see *ttype* above), and the *MR-list*. If this keyletter is set when creating an SCCS file, the `m` keyletter must also be used even if the MR list is empty.
- d flag**
flag is deleted from the SCCS file.
In any one `admin` command, a number of `-d` keyletters may be used. See the `-f` keyletter for allowable *flag* values.
- a login**
A login name, or numerical group ID can be added to the list of users who may make deltas (changes) to the SCCS file.
A group ID is equivalent to specifying all login names common to that group ID. Several `a` keyletters may be used on a single `admin` command line. As many *logins* or numerical group IDs as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If a login or group ID is preceded by a `!`, these users are explicitly denied permission to make deltas.
- e login**
A login name, or numerical group ID, to be erased from the list of users allowed to make deltas to the SCCS file.
Specifying a group ID is equivalent to specifying all *login* names included under that group ID. Several `-e` keyletters may be used on a single `admin` command line.
- m MR-list**
The list of Modification Request (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to `delta`.
The `v` flag must be set in the `-f` keyletter, and the MR numbers will be validated if the `v` flag has an argument (the name of an MR number validation program). A message will be output if the `v` flag is not set or MR validation fails.
- y[comment]**
The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of `delta`.
Omission of the `-y` keyletter results in a default comment line being inserted:
date and time created <date> <time> by <login>
The `-y` keyletter is valid only if the `-i` and/or `-n` keyletters are specified, i.e. a new SCCS file is being created.
- h** `admin` will check the structure of the SCCS file [see `sccsfile(4)`].
A newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) will be compared with the check-sum that is stored in the first line of the SCCS file, and if appropriate an error message will be output.
The keyletter `-h` cannot be combined with other keyletters. It inhibits writing to the file, nullifying the effect of all other keyletters supplied.

- z The SCCS file check-sum will be recomputed and stored in the first line of the SCCS file (see —h).
Note that use of this keyletter on a severely corrupted file may prevent future detection of the corruption.
The keyletter —z cannot be combined with other options.
- If the first filename begins with a dash (—), the end of the command-line options must be marked with —.

The name of any SCCS file, omitting the path details, must be in the form *s.filename*. New SCCS files are set up using mode 444 (provided the permissions have not been restricted by *umask*). Write permission in the pertinent directory is required to create a file. All writing done by *admin* is to a temporary file, which has the name *x.filename*. Only after successful execution of the required action is the original file, if there is one, deleted and the *x*-file is renamed. This ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be mode 444. The mode of the SCCS files ensures that modifications can only be made using the SCCS commands.

admin also makes use of a transient lock file, called *z.filename*, to prevent simultaneous updates to the SCCS file by different users. See [get\(1\)](#) for further information.

NOTES

If it is necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner to allow a text editor to be used. This must be followed by a call to *admin -z* to generate a proper check-sum. *admin -h* should be called to ensure the SCCS file is valid.

LOCALE

The language of the message texts is governed by the environment variable *LC_ALL*, *LC_MESSAGES* or *LANG*. When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

FILES

- x*-file
see [delta\(1\)](#)
- z*-file
see [delta\(1\)](#)

SEE ALSO

[bdiff\(1\)](#), [ed\(1\)](#), [delta\(1\)](#), [get\(1\)](#), [help\(1\)](#), [prs\(1\)](#), [what\(1\)](#), [sccsfile\(4\)](#).

NAME

as - assembler

SYNOPSIS

as [*options*] *file*

DESCRIPTION

The `as` command creates object files from assembly language source *files*. The output file is executable if no errors occurred during the assembly, transfer vectors were not used, and there are no unresolved external references. The following flags may be specified in any order:

`genelf`

Have the assembler produce ELF. Otherwise, assembler produces XCOFF (the default).

`abicalls`

Have the assembler produce ABI calling sequence.

`pic` Have the assembler produce position independent code.

`-g0` Have the assembler produce no symbol table information for symbolic debugging. This is the default.

`-g1` Have the assembler produce additional symbol table information for accurate but limited symbolic debugging of partially optimized code.

`-g` or `-g2`

Have the assembler produce additional symbol table information for full symbolic debugging and not do optimizations that limit full symbolic debugging.

`-g3` Have the assembler produce additional symbol table information for full symbolic debugging for fully optimized code. This option makes the debugger inaccurate.

`-w` Suppress warning messages.

`-P` Run only the C macro preprocessor and put the result in a file with the suffix of the source file changed to `.i` or if the file has no suffix then a `.i` is added to the source filename. The `.i` file has no `#` lines in it. This sets the `-cpp` option.

`-E` Run only the C macro preprocessor on the file and send the result to the standard output. This sets the `-cpp` option.

`-o output`

Name the final output file *output*. If this option is used, the file `a.out` is not generated.

`-Dname=def` or `-Dname`

Define the *name* to the C macro preprocessor, as if by `#define`. If no definition is given, the name is defined as `1`.

`-Uname`

Remove any initial definition of *name*.

`-Idir`

The `#include` files whose names do not begin with `/` are always sought first in the directory of the *file* argument, then in directories specified in `-I` options, and finally in the standard directory (`/usr/include`).

`-I` This option will cause `#include` files never to be searched for in the standard directory (`/usr/include`).

`G num`

Specify the maximum size, in bytes, of a data item that is to be accessed from the global pointer. *num* is assumed to be a decimal number. If *num* is zero, no data is accessed from the global pointer. The default value for *num* is 8 bytes.

- `-v` Print the passes as they execute with their arguments and their input and output files.
- `-V` Print the version of the driver and the versions of all passes. This is done with the `what` command.
- `-cpp` Run the C macro preprocessor on assembly source files before compiling. This is the default for `as`.
- `-nocpp`
Do not run the C macro preprocessor on assembly source files before compiling.

Either object file target byte ordering can be produced by `as`. The default target byte ordering matches the machine where the assembler is running. The options `-EB` and `-EL` specify the target byte ordering (big-endian and little-endian, respectively). The assembler also defines a C preprocessor macro for the target byte ordering. These C preprocessor macros are `MIPSEB` and `MIPSEL` for big-endian and little-endian byte ordering respectively.

- `-EB` Produce object files targeted for big-endian byte ordering. The C preprocessor macro `MIPSEB` is defined by the assembler.
- `-EL` Produce object files targeted for little-endian byte ordering. The C preprocessor macro `MIPSEL` is defined by the assembler.

The following option is specified for `as`:

- `-m` Apply the `m4` preprocessor to the source file before assembling it.

The option described below is primarily used to provide Reliant UNIX compilation environments other than the native compilation environment.

- `-systype name`
Use the named compilation environment *name*. The names of the compilation environments supported are `bsd43` and `sysv`. This has the effect of changing the standard directory for `#include` files. The new items are located in their usual paths but with `/name` prepended to their paths. Also a preprocessor macro of the form `SYSTYPE_NAME` (with *name* capitalized) is defined in place of the default `SYSTYPE_SYSV`.

The options described below primarily aid compiler development and are not generally used:

- `-Hc` Halt compiling after the pass specified by the character *c*, producing an intermediate file for the next pass. The *c* can be `[a]`. It selects the assembler pass in the same way as the `-t` option. If this option is used, the symbol table file produced and used by the passes, is the last component of the source file with the suffix changed to `.T` or a `.T` is added if the source file has no suffix. This file is not removed.
- `-K` Build and use the intermediate filenames with the last component of the source file's name replacing its suffix with the conventional suffix for the type of file. If the source file has no suffix the conventional suffix is added to the source filename. These intermediate files are never removed even when a pass encounters a fatal error.
- `-Wc[c...], arg1[,arg2...]`
Pass the argument(s) *arg_i* to the compiler pass(es) *c[c...]*. The *c*'s are one of `[pab]`. The *c*'s selects the compiler pass in the same way as the `-t` option.

The options `-t [pab]`, `-hpath`, and `-Bstring` select a name to use for a particular pass. These arguments are processed from left to right so their order is significant. When the `-B` option is encountered, the selection of names takes place using the last `-h` and `-t`. Sets of these options can be used to select any combination of names.

- `-t [pab]`
Select the names. The names selected are those designated by the characters following the `-t` option according to the following table:

Name	Character

<code>cpp</code>	<code>p</code>

<code>as0</code>	<code>a</code>

as1 b

-hpath

Use *path* rather than the directory where the name is normally found.

-Bstring

Append *string* to all names specified by the **-t** option. If no **-t** option has been processed before the **-B**, the **-t** option is assumed to be **pab**. This list designated all names.

Invoking the assembler with a name of the form *string* has the same effect as using a **Bstring** option on the command line.

If the environment variable `$COMP_HOST_ROOT` is set, the value is used as the root directory for all pass names rather than the default `/`. If the environment variable `$COMP_TARGET_ROOT` is set, the value is used as the root directory for the includes rather than the default `/`.

If the environment variable `$ROOTDIR` is set, the value is used as the root directory for all names rather than the default `/usr`. This also affects the standard directory for `#include` files, `/usr/include`.

If the environment variable `$TMPDIR` is set, the value is used as the directory to place any temporary files rather than the default `/tmp`.

Other arguments are ignored.

FILES

file.o

Object file

a.out

Assembler output

/tmp/ctm?

Temporary

/usr/lib/cpp

Cmacro preprocessor

/usr/lib/as0

Assembler

/usr/lib/as1

Assembler

/usr/include

Standard directory for include files

SEE ALSO

cc(1).

NAME

`asa` - convert control characters for positioning

SYNOPSIS

`asa [ file ... ]`

DESCRIPTION

`asa` reads input files, converts control characters for positioning into control sequences for a line printer and outputs the changed files to standard output. If *file* is not specified, `asa` reads from the standard input.

The first character of every line is evaluated as the control character for positioning and removed from the input. The following characters are recognized by `asa` and converted to printer statements:

blank

The line is output without change.

0 A newline character is output before the line.

1 A control sequence for advancing to the next page is output before the line.

+ The newline character of the previous line is replaced by a carriage return character and the current line is then output. If + is the first character of the input, it is treated as a blank.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES`, or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must set one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization – Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

SEE ALSO

[lp\(1\)](#).

NAME

`cb` - C program beautifier

SYNOPSIS

`cb` [`-s`] [`-j`] [`-l` *length*] [`-v`] [*file ...*]

DESCRIPTION

The `cb` command (C program beautifier) reads C programs either from the files specified by its arguments or from the standard input, and writes them on the standard output with spacing and indentation that display the structure of the C code. By default, `cb` preserves all user newline characters.

`cb` accepts the following options.

- `-s` Write the code in the style of Kernighan and Ritchie.
- `-j` Put split lines back together.
- `-l` *length*
`cb` will split lines that are longer than *length*.
- `-v` Print on standard error output the version of `cb` invoked.

NOTES

`cb` treats `asm` as a keyword.

The format of structure initializations is unchanged by `cb`.

Punctuation that is hidden in preprocessing directives causes indentation errors.

SEE ALSO

`cc(1)`.

Kernighan/Ritchie: "The C Programming Language".

NAME

`cdc` - change the delta comment of an SCCS delta

SYNOPSIS

`cdc -r SID [-m [MR-list]] [-y[comment]] file...`

DESCRIPTION

`cdc` (change delta comment) changes the delta comment, for the SID (SCCS identification string) specified by the `-r` keyletter, of each named SCCS file.

The delta comment consists of the Modification Request (MR) and comment information normally specified in a `delta` command (`-m` and `-y` keyletters).

If *file* is a directory, `cdc` behaves as though each file in the directory were specified as a named file, except that non-SCCS files [see `admin(1)`] and unreadable files are ignored without comment. If a name of `-` is given, the standard input is read and each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to `cdc`, which may appear in any order, consist of keyletter arguments and filenames.

All the described keyletter arguments apply independently to each named file:

`-rSID`

Used to specify the SCCS identification (SID) string of a delta for which the delta comment is to be changed.

`-mMR-list`

If the SCCS file has the `v` flag set [see `admin(1)`] then a list of MR numbers to be added and/or deleted in the delta comment of the SID specified by the `-r` keyletter may be supplied. A null *MR-list* has no effect.

MR numbers are added to the list of MRs in the same manner as that of `delta`. In order to delete an MR, the MR number must be preceded by the character `!` (see the **EXAMPLES** section). If the MR to be deleted is currently in the list of MRs, it is removed and changed into a comment line. A list of all deleted MRs is placed in the comment section of the delta comment and preceded by a comment line stating that they were deleted.

If `-m` is not used and the standard input is the keyboard, the prompt `MRs?` is issued on the standard output before the standard input is read; if the standard input is not the keyboard, no prompt is issued. The `MRs?` prompt always precedes the `comments?` prompt (see `-y`).

Entries in the *MR-list* are separated by blanks and/or tab characters. An unescaped newline character terminates the *MR-list*.

If the flag `v` has a value [see `admin(1)`], it is taken to be the name of a program (or shell procedure) that validates the correctness of the MR numbers. If a non-zero exit status is returned from the validation program, `cdc` terminates and the delta comment remains unchanged.

`-y[comment]`

Arbitrary text which replaces the comments which previously existed for the delta specified by the `-r` keyletter.

The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

If `-y` is not specified and the standard input is the keyboard, the prompt `comments?` is issued on the standard output before the standard input is read; if the standard input is not the keyboard, no prompt is issued. An unescaped newline character terminates the comment text.

If you made the delta and have the appropriate file permissions, you can change its delta comment.

EXAMPLES

The input

```
cdc -r1.6 -m"bl88-12345 !bl87-54321 bl89-00001" -ytrouble s.file
```

adds bl88-12345 and bl89-00001 to the MR list, removes bl87-54321 from the MR list, and adds the comment trouble to delta 1.6 of s.file.

The following input produces the same result:

```
cdc -r1.6 s.file
MRs? !bl87-54321 bl88-12345 bl89-00001
comments? trouble
```

NOTES

If SCCS filenames are supplied to the `cdc` command via the standard input (– on the command line), then the `–m` and `–y` keyletters must also be used.

FILES

x-file [see [delta\(1\)](#)]

z-file

[see [delta\(1\)](#)]

SEE ALSO

[admin\(1\)](#), [delta\(1\)](#), [get\(1\)](#), [help\(1\)](#), [prs\(1\)](#), [sccsfile\(4\)](#).

NAME

cflow - generate C flowgraph

SYNOPSIS

cflow [**-r**][**-i x**][**-i _**][**-d num**][*cc-options*][**—**] *file...*

DESCRIPTION

The **cflow** command analyzes a collection of **yacc**, **lex**, assembler, C and object files and builds a graph charting the external function references. Files suffixed with **.y**, **.l**, and **.c** are processed by **yacc**, **lex**, and the C compiler as appropriate. The results of the preprocessed files, and files suffixed with **.i**, are then run through the first pass of **lint**. Files suffixed with **.s** are assembled. Assembled files, and files suffixed with **.o**, have information extracted from their symbol tables. The results are collected and turned into a graph of external references that is written on the standard output.

Each line of output consists, in sequence, of a line number, the name of the global symbol, a colon, and the type definition. The name will be indented by a number of tab positions which corresponds to its nesting depth. Normally only function names that do not begin with an underscore are listed (see **-i**). For information extracted from C source code, the definition consists of an abstract type declaration, the name of the source file delimited by angle brackets, and the line number where the definition was found. Definitions extracted from object files indicate the filename and location counter under which the symbol appeared (e.g., *text*). Leading underscores in external names are deleted. Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only **<>** is printed.

OPTIONS

In addition to the **-D**, **-I**, and **-U** options, which are interpreted just as they are by **cc/c89**, the following options are interpreted by **cflow**:

- r** Reverse the "caller: callee" relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.
- i x** Include external and static data symbols. The default is to include only functions in the flowgraph.
- i _** Include names that begin with an underscore. The default is to exclude these functions (and data if **-i x** is used).
- d num** *num* indicates the depth at which the flowgraph is cut off. By default this number is very large. Attempts to set the cutoff depth to a nonpositive integer will be ignored.
- If the first filename begins with a dash (**-**), the end of the command-line options must be marked with **—**.

EXAMPLES

Suppose the following code is in **file.c**:

```
int    i;
main()
{
    f();
    g();
    f();
}
f()
{
    i = h();
```

```
}
```

The command

```
cflow -i x file.c
```

produces the output

```
1      main: int(), <file.c 4>
2          f: int(), <file.c 11>
3              h: <>
4              i: int, <file.c 1>
5          g: <>
```

When the nesting level becomes too deep, the output of `cflow` can be piped (using the `-e` option) to the `pr` command to compress the tab expansion to less than eight spaces.

EXIT STATUS

`cflow` reports multiple definitions. Only the first is accepted.

NOTES

Files produced by `lex` and `yacc` cause the reordering of line number declarations, which can disrupt processing by `cflow`. To get proper results, feed `cflow` the `yacc` or `lex` input.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization – Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

SEE ALSO

[cc\(1\)](#), [c89\(1\)](#), [lex\(1\)](#), [lint\(1\)](#), [nm\(1\)](#), [pr\(1\)](#), [yacc\(1\)](#).

check_abi_compliance(1)**NAME**

`check_abi_compliance` - check abi compliance of an object file

SYNOPSIS

```
/usr/ccs/bin/check_abi_compliance [-abidir directory] [-liblist library_list] [-required]  
[-abi] file
```

DESCRIPTION

`check_abi_compliance` checks an object file for abi compliance and optionally displays references to routines which are not part of the abi and required routines which are missing from the object file.

OPTIONS

The following options are available:

`-abidir directory`

This specifies a directory which contains files which specify abi routines. The filenames are of the form `LIBRARY.names`. Default is `/usr/ccs/lib/abi/data`.

`-liblist library_list`

This specifies a list of shared objects on which this object file depends. This defaults to the liblist of the object file.

`-required`

Display only missing required routines. An example of this is `_end`.

`-abi` Display only undefined references which are not part of the abi.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, `check_abi_compliance` returns 1.

FILES

`/usr/ccs/lib/abi/data`
directory containing ABI entry points

SEE ALSO

`check_abi_interface(1)`, `check_for_syscalls(1)`, `dump(1)`, `ldd(1)`.

check_abi_interface(1)**NAME**

`check_abi_interface` - check abi compliance of a library interface

SYNOPSIS

```
/usr/ccs/bin/check_abi_interface [-abidir directory] [-missing] [-extra] [-libname library]  
] file
```

DESCRIPTION

`check_abi_interface` checks a library interface and displays abi entries which are missing as well as public entries which are not part of the abi.

OPTIONS

The following options are available:

`-abidir directory`

This specifies a directory which contains files which specify abi routines. The filenames are of the form `LIBRARY.names`. Default is `/usr/ccs/lib/abi/data`.

`-libname library_list`

This specifies the name of the library to use when searching in `abidir`. The default is the name of the file.

`-missing`

Display only abi routines which are missing from the library.

`-extra`

Display only public routines which are not part of the abi.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, `check_abi_interface` returns 1.

FILES

`/usr/ccs/lib/abi/data`
directory containing ABI entry points

SEE ALSO

`check_abi_compliance(1)`, `check_for_syscalls(1)`, `dump(1)`.

check_for_syscalls(1)**NAME**

`check_for_syscalls` - check for embedded syscalls in an object file

SYNOPSIS

`/usr/ccs/bin/check_for_syscalls` [`-syscalls` *syscall_file*] [`-abidir` *directory*] *file*

DESCRIPTION

`check_for_syscalls` finds embedded system calls in an object file.

OPTIONS

The following options are available:

`-syscalls` *syscall_file*

This file is used to translate from system call numbers to names. The default is

`/usr/ccs/lib/abi/data/syscall.names`.

`-abidir` *directory*

This specifies a directory which contains the file which is used to translate system call numbers to names (`syscall.names`). Default is `/usr/ccs/lib/abi/data`.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, `check_for_syscalls` returns 1.

FILES

`/usr/ccs/lib/abi/data/syscall.names`

SEE ALSO

`check_abi_compliance(1)`, `check_abi_interface(1)`, `dump(1)`.

NAME

`comb` - combine SCCS deltas

SYNOPSIS

`comb` [`-o`] [`-s`] [`-pSID`] [`-clist`] *file...*

DESCRIPTION

`comb` (combine) generates a shell procedure with which the specified SCCS files can be reconstructed. The reconstructed files are, in general, smaller than the original files. The arguments may be specified in any order, but all the options apply to all named SCCS files. If a directory is named, `comb` behaves as though each file in the directory were specified as a named file. This does not apply for non-SCCS files – i.e. those for which the filename does not begin with `s.`, ignoring the pathname – and unreadable files; these files are ignored without comment. If a name of `-` is given, the standard input is read; each line of the input is taken to be the name of an SCCS file to be processed. The generated shell procedure is written on the standard output.

OPTIONS

The options will be applied independently to each named file.

- `-o` For each `get -e`, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor. Use of the `-o` keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.
- `-s` `comb` will generate a shell procedure that, when run, produces a report that gives for each file: the filename, its size in blocks after applying `comb`, its original size in blocks, and percentage change. The latter is computed as follows:

$$100 * (original\ size - new\ size) / original\ size$$
This option should be used before any SCCS files are actually combined, to determine how much space will be saved.
- `-pSID` Specifies the SCCS identification string (SID) of the oldest delta to be preserved. All older deltas are deleted from the reconstructed file.
- `-clist` Alist of deltas to be preserved. All other deltas are discarded (see `get(1)` for the syntax of a *list*).

If no options are specified, `comb` preserves only leaf deltas and the minimal number of ancestors needed to preserve the tree.

NOTES

`comb` may rearrange the shape of the tree of deltas. In some circumstances, `comb` may not save any space; in fact, it is possible for the reconstructed file to be larger than the original.

FILES

`s.COMB`
the reconstructed SCCS file

`comb????`
temporary file

SEE ALSO

`admin(1)`, `delta(1)`, `get(1)`, `prs(1)`, `sh(1)`, `sccsfile(4)`.

NAME

`cscope` - interactively examine a C program

SYNOPSIS

`cscope` [*option...*] *file...*

DESCRIPTION

`cscope` is an interactive screen-oriented tool that allows the user to browse through C source files for specified elements of code.

By default, `cscope` examines the C, `lex`, and `yacc` source files in the current directory. `cscope` may also be invoked for source files named in the command line. In either case, `cscope` searches the standard directories for `#include` files that it does not find in the current directory. `cscope` uses a symbol cross-reference table, `cscope.out` by default, to locate functions, function calls, macros, variables, and preprocessor symbols in the files.

`cscope` builds the symbol cross-reference table the first time it is used on the source files for the program being browsed. On a subsequent invocation, `cscope` rebuilds the cross-reference table only if a source file has changed or the list of source files is different. When the cross-reference table is rebuilt, the data for the unchanged files are copied from the old cross-reference table, which makes rebuilding faster than the initial build.

OPTIONS

- `-b` Build the cross-references only.
- `-C` Ignore letter case when searching.
- `-c` Use only ASCII characters in the cross-reference table, that is, do not compress the data.
- `-d` Do not update the cross-reference table
- `-e` Suppress the command prompt between files.
- `-f reffile`
Use *reffile* as the filename for the cross-reference table instead of `cscope.out`.
- `-I incdir`
Before looking in *INCDIR*, the directory for header files (normally `/usr/include`), a search should be made in *incdir* for any `#include` files whose names do not begin with `/` and that are not specified on the command line or in *namefile* (see below). The `#include` files may be specified with either double quotes or angle brackets. The *incdir* directory is searched in addition to the current directory and the standard directories. If the option `-I` appears more than once, the directories are searched in the order they appear on the command line.
- `-i namefile`
Browse through all source files whose names are listed in *namefile* instead of the default file `cscope.files`. Filenames are separated by spaces, tabs, or new-lines. If this option is specified, `cscope` ignores any files appearing on the command line.
- `-L` Do a single search with line-oriented output when used with the `-num pattern` option.
- `-l` Line-oriented interface (see section entitled "Line-oriented interface").
- `-num pattern`
Go to input field *num* and find *pattern*. Counting starts from 0.
- `-P path`
Prepend *path* to relative filenames in a pre-built cross-reference file so you do not have to change to the directory where the cross-reference file was built. This option is only valid with the `-d` option.

- `-p n` Display the last *n* file path components instead of the default (1). Use 0 to not display the filename at all.
 - `-s dir` Look in *dir* for additional source files. This option is ignored if source files are given on the command line.
 - `-T` Use only the first eight characters to match against C symbols. A regular expression containing special characters other than a period will not match any symbol if its minimum length is greater than eight characters.
 - `-U` Do not check file time stamps. It is assumed that no files have changed.
 - `-u` Unconditionally build the cross-reference file. It is assumed that all files have changed.
 - `-V` Print on the first line of screen the version number of *cscope*.
- The `-I`, `-p`, and `-T` options can also be in the *cscope.files* file.

Requesting the initial search

After the cross-reference table is ready, *cscope* will display this menu:

```
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Press the `TAB` key repeatedly to move to the desired input field, type the text to search for, and then press the `RETURN` key.

Issuing subsequent requests

If the search is successful, any of these key combinations can be used:

- `1-9` Edit the line referenced by the number which was input
- `SPACE` Output the next lines found
- `+` Output the next lines found
- `^v` Output the next lines found
- `-` Output the next files found
- `^e` Edit all the files output, in order
- `>` Append list of lines to a file
- `|` Pass all the displayed lines to a shell command

The following key combinations can be used to control the *cscope* menu and to execute other commands:

- `TAB` Next input field
- `RETURN` Next input field
- `^n` Next input field
- `^p` Previous input field
- `^y` Search for last text input
- `^b` Next input field and search pattern
- `^f` Next input field and search pattern
- `^c` Take uppercase/lowercase into account in searching

```

^r    Recreate cross reference table
!     Start an interactive shell (input ^d to return to cscope)
^l    Redraw the screen
?     Output help information about cscope commands
^d    Exit cscope

```

If the first character of the text to be searched for matches one of the specified key combinations, type a \ before this character.

Substituting new text for old text

After the text to be changed has been typed, `cscope` will prompt for the new text, and then it will display the lines containing the old text. Select the lines to be changed with the following keys:

```

1-9   Select a line
*     Select a line
SPACE Output next line
+     Output next line
-     Output previous line
a     Select all lines
^d    Amend and exit from selected lines
ESCAPE Exit from edit mode without altering the marked lines

```

Special keys

If your terminal has arrow keys that work in `vi(1)`, you can use them to move around between the input fields. The up-arrow key is useful to move to the previous input field instead of using the `TAB` key repeatedly. If you have `CLEAR`, `NEXT`, or `PREV` keys they will act as the `^l`, `+`, and `-` commands, respectively.

Line-oriented interface

The `-l` option lets you use `cscope` where a screen-oriented interface would not be useful, e.g., from another screen-oriented program.

`cscope` prompts with `>>` and then expects input of a field number (counting from 0), immediately followed by the search pattern. For example, `lmain` will find the definition of the `main` function.

If you only want the file to be searched once, you should use the `-L` and `-num pattern` options.

For `-l`, `cscope` outputs the number of reference lines

```
cscope: 2 lines
```

For each reference found, `cscope` outputs a line consisting of the filename, function name, line number, and line text, separated by spaces, e.g.,

```
main.c main 161 main(argc, argv)
```

Note that the editor is not called to display a single reference, unlike the screen-oriented interface.

`cscope` will quit when it detects end-of-file, or when the first character of an input line is `^d` or `q`.

Environment variables

```

EDITOR
    The preferred editor
INCLUDEDIRS
    Colon-separated list of directories to search for #include files.
HOME
    Home directory, which is automatically set at login.
SHELL

```

Preferred shell, which defaults to `sh(1)`.

SOURCEDIRS

Colon-separated list of directories to search for additional source files.

TERM Terminal type, which must be a screen terminal.

TERMINFO

Full pathname of terminal information directory. If your terminal type is not in the standard `terminfo` directory, see `curses(3X)` and `terminfo(4)` for how to make your own terminal description.

TMPDIR

Temporary file directory, which defaults to `/var/tmp`.

VIEWER

Selects the file display program (such as `pg(1)`). This has priority over `EDITOR`.

VPATH

A colon-separated list of directories, each of which has the same directory structure.

If `VPATH` is set, `cscope` searches for source files in the directories specified; if it is not set, `cscope` searches only in the current directory.

NOTES

`cscope` recognizes function definitions of the form:

```
fname blank ( args ) white arg_decs white {
```

where:

fname

is the function name

blank

is zero or more spaces or tabs, not including newlines

args is any string that does not contain a `"`, a `(` or a newline

white

is zero or more spaces, tabs, or newlines

arg_decs

are zero or more argument declarations which may include comments and white space

It is not necessary for a function declaration to start at the beginning of a line. The return type may precede the function name; `cscope` will still recognize the declaration. Function definitions that deviate from this form will not be recognized by `cscope`.

The **Function** column of the search output for the menu option **Find functions called by this function:** will only display the first function called in the line. That is, for the following function

```
e()
{
    return (f() + g());
}
```

the display would be

```
Functions called by this function: e
File  Function      Line
a.c   f                 3 return(f() + g());
```

Occasionally, a function definition or call may not be recognized because of braces inside `#if` statements. Similarly, the use of a variable may be incorrectly recognized as a definition.

A `typedef` name preceding a preprocessor statement will be incorrectly recognized as a global definition, e.g.,

```
LDFILE *
#if AR16WR
```

Preprocessor statements can also prevent the recognition of a global definition, e.g.,

```

char flag
#ifdef ALLOCATE_STORAGE
    = -1
#endif
;

```

A function declaration inside a function is incorrectly recognized as a function call, e.g., the following function declaration will be incorrectly recognized as a call to `g()`:

```

f()
{
    void g();
}

```

`cscope` recognizes C++ classes by looking for the `class` keyword, but doesn't recognize that a `struct` is also a class, so it doesn't recognize inline member function definitions in a structure. It also doesn't expect the `class` keyword in a `typedef`, so it incorrectly recognizes `X` as a definition in:

```
typedef class X * Y;
```

It also doesn't recognize operator function definitions

```

Bool Feature::operator==(const Feature & other)
{
    ...
}

```

FILES

`cscope.files`

Default file containing `-I`, `-p` and `-T` options and the list of sourcefiles; another choice can be made using the `-i` option.

`cscope.out`

Symbol cross-reference table, which is put in the home directory if it cannot be created in the current directory.

`ncscope.out`

Temporary file containing new cross-reference table before it replaces the old cross-reference.

`INCDIR`

Standard directory for `#include` files (usually `/usr/include`).

SEE ALSO

[curses\(3X\)](#), [terminfo\(4\)](#).

The "cscope" chapter in the "Guide to Tools for Programming in C".

NAME

`ctags` - create a tags file

SYNOPSIS

`ctags [ option ... ] [ file ... ]`

DESCRIPTION

`ctags` helps you to locate function, macro, and data type definitions in C, Pascal, FORTRAN, YACC, and LEX sources by creating a tags file from the specified source programs. The tag descriptions are placed in a file called `tags` by default.

Each line of the tags file contains the following information about a function, macro, or `typedef` data type definition (only with the `-t` option) in separate fields delimited by space or tab characters:

- the name of the function, macro, or data type definition
- the name of the file in which the function, macro, or data type is defined
- for functions:
 - a text pattern that can be used in an editor to locate the function definition
- for macros:
 - a text pattern that can be used in an editor to locate the macro definition
- for data types:
 - the line number of the `typedef` statement

The tag `main` is treated specially in C programs. Instead of using the function name `main`, `ctags` creates the tag by prepending `M` to the program file's basename and removing its `.c` or `.h` ending if it has one. Thus, if the function `main` is in a file called `/home/janice/help/xyz.c`, for example, the tag for this file will be named `Mxyz`.

Using the information in the tags file, an editor such as `vi`, `ex`, or `ced` can quickly and easily locate function, macro and data type definitions.

Caution:

The algorithm for recognizing functions, subroutines, and procedures in FORTRAN and Pascal is extremely simple. No attempt is made to deal with block structures.

`ctags` does not recognize `#ifdef` constructs.

OPTIONS

No option specified:

The tags file is assigned the name `tags`.

Text patterns are enclosed within slashes (`/.../`), suitable for forward searching with an editor. C programs are searched for C routine and macro definitions.

- a `ctags` appends its output to an existing tags file, without overwriting it. This file may be named `tags` (default) or `tagsfile` (see the `-f` option).
- B (backward) Text patterns are enclosed within question marks (`?...?`), suitable for backward searching with an editor.
- F (forward) Text patterns are enclosed within slashes (`/.../`), suitable for forward searching with an editor. This option is set by default.
- f *tagsfile*
(file) Assigns the name *tagsfile* to the tags file.
- t (typedef) Create a tags file with `typedefs` (data type definitions) as well as function and macro

definitions.

`-t` not specified:

Only function and macro definitions are tagged.

- `-u` (update) `ctags` updates the information relating to the specified files in the tags file; that is, all references to them are deleted, and the new values are appended to the tags file. This option is implemented in a way which is rather slow; it is usually faster to simply rebuild the tags file.
- `-v` (verbose) `ctags` writes to standard output an index listing the function name, the file name, and the number of the page (based on 64 lines per page) on which the function is defined. The index is sorted in lexicographic order.
- `-w` (warning) `ctags` suppresses warning diagnostics.
- `-x` (index) Instead of writing information for the specified files to a tags file, `ctags` writes to standard output an index with the following entries in each output line:
 - the name of the function or data type,
 - the number of the line on which it is defined,
 - the name of the file containing the definition,
 - the text of the definition.

There is no special treatment for `main`. Warnings are issued if there are duplicated function names.

In conjunction with option `-t`:

Only the last line of the data type definition is output.

- If *file* begins with a dash (`-`), the end of the command-line options must be marked with `—`.

file Name of the file to be searched by `ctags`.

You may specify more than one file name on the command line. If you do, `ctags` writes its output to a tags file.

`ctags` makes assumptions about the nature of the source program on the basis of the file name ending:

`.c` and `.h`:

Csource

`.y`: YACC source

`.l`: LEX source

other:

Pascal or FORTRAN source if the file contains any function, subroutine or procedure definitions that are specific to Pascal or FORTRAN.

C source if there are no such definitions.

ERROR MESSAGES

Duplicate entry in files *file1* and *file2*: ...

There are identically named objects in *file1* and *file2*.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The file `program.c` contains the following C program:

```
#define END 20
main()
{
    int i;
    for (i=0; i<END; i++)
        output(i);
}

output(i)
int i;
{
    printf("The square of %d is equal to %d\n", i, square(i));
}

int square(i)
int i;
{
    return(i*i);
}

$ ctags -x program.c
output      10 program.c    output(i)
main        3 program.c    main()
square      16 program.c    int square(i)
```

FILES

tags Default file name of the tags file created by ctags.

SEE ALSO

[ex\(1\)](#), [vi\(1\)](#).

NAME

`ctrace` - C program debugger

SYNOPSIS

`ctrace` [*option ...*] [*file*]

DESCRIPTION

The `ctrace` command prepares a C program for monitoring at runtime. The effect is similar to executing a shell procedure with the `-x` option. `ctrace` reads the C program in *file*, or from standard input if the user does not specify *file*. To each executable statement in the program, it adds statements which will output at runtime the text of the executable statement and the values of all variables referenced or modified. `ctrace` writes this modified program to the standard output. The output of `ctrace` must be placed into a temporary file because the `cc/c89` command does not allow the use of a pipe. This file can then be compiled and executed.

As each statement in the program executes, it will be listed on the screen, followed by the name and value of any variables referenced or modified in the statement; these variable names and values will be followed by any output which the statement initiates. Loops in the trace output are detected and tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed after each 1000 loop cycles to help the user detect infinite loops. The trace output is written to the standard output, which can be redefined as a file so that the user can examine it using an editor or other commands.

OPTIONS

The options commonly used are:

`-f functions`

Trace only these *functions*.

`-v functions`

Trace all but these *functions*.

The user may want to add to the default formats for printing variables. `long` and pointer variables are always printed as signed integers. Pointers to character arrays are also printed as strings, where meaningful. `char`, `short`, and `int` variables are also printed as signed integers and, if appropriate, as characters. `double` variables are printed as floating point numbers in scientific notation.

The user can request that variables be printed in additional formats, if appropriate, with these options:

`-o` Octal.

`-x` Hexadecimal.

`-u` Unsigned.

`-e` Floating point.

These options are used only in special circumstances:

`-l n` Check every *n* consecutively executed statements for looping trace output, instead of the default of 20.

Use 0 to get all the trace output from loops.

`-s` Suppress redundant trace output from simple assignments and string copy functions.

This option can hide a bug caused by use of the `=` operator in place of the `==` operator.

`-t n` Trace *n* variables per statement instead of the default of 10 (the maximum number is 20).

`-P` Preprocess the input before tracing it.

The user can also use the `-D`, `-I`, and `-U` options of `cc/c89`.

-p *string*

Change the trace print function from the default of `printf`.

For example, `fprintf(stderr)` would send the trace to the standard error output.

-r *f* Use file *f* in place of the `runtime.c` trace function package.

This replacement lets the user change the entire print function, instead of just the name and leading arguments (see the **-p** option)

-v Prints information about the `ctrace` version on the standard error output.

-Q *arg*

If *arg* is *y*, identification information about `ctrace` will be added to the output files. This can be useful for software administration.

Giving *n* for *arg* explicitly asks for no such information, which is the default behavior.

EXAMPLES

The file `lc.c` contains the following incorrect C program:

```
1 #include <stdio.h>
2 main()      /* count lines in input */
3 {
4     int c, nl;
5
6     nl = 0;
7     while ((c = getchar()) != EOF)
8         if (c == '\n')
9             ++nl;
10    printf("%d\n", nl);
11 }
```

Input the following commands and test data:

```
cc lc.c
a.out
1
(^d)
```

The program will be compiled and executed. The output of the program will be the number 2, which is incorrect because there is only one line in the test data. The error in this program is common, but often difficult to find. If the user invokes `ctrace` with these commands:

```
ctrace lc.c >temp.c
cc temp.c
a.out
```

the output will be:

```
2 main()
6     nl = 0;
    /* nl == 0 */
7     while ((c = getchar()) != EOF)
```

The program is now waiting for input. If the user enters the same test data as before, the output will be:

```
    /* c == 49 or '1' */
8         if (c == '\n')
            /* c == 10 or '\n' */
9             ++nl;
        /* nl == 1 */
7     while ((c = getchar()) != EOF)
    /* c == 10 or '\n' */
8         if (c == '\n')
            /* c == 10 or '\n' */
9             ++nl;
        /* nl == 2 */
```

```
7      while ((c = getchar()) != EOF)
```

If an end-of-file character (ctrl-d) is entered, the final output will be:

```
/* c == -1 */
10     printf("%d\n", nl);
/* nl == 2 */
return
```

Note the information printed out at the end of the trace line for the `nl` variable following line 10. Also note the `return` comment added by `ctrace` at the end of the trace output. This shows the implicit return at the terminating brace in the function.

The trace output shows that variable `c` is assigned the value `1` in line 7, but in line 8 it has the value `\n`. Once user attention is drawn to this `if` statement, he or she will realize that the assignment operator `=` was used in place of the equality operator `==`. This error can easily be missed during code reading.

Execution-time trace control

The default operation for `ctrace` is to trace the entire program, unless the `-f` or `-v` options are used to trace specific functions. The default operation does not give the user the choice of tracing individual statements, nor does it let the user turn the tracing off and on when executing the traced program.

The user can do both of these by adding `ctroff()` and `ctron()` function calls to the program to turn the tracing off and on, respectively, at execution time. Thus, complex criteria can be arbitrarily coded for trace control with `if` statements, and this code can even be conditionally included because `ctrace` defines the `CTRACE` preprocessor variable.

The trace can be turned off and on by setting static variable `tr_ct_` to 0 and 1, respectively. This on/off option is useful if a user is using a debugger that can not call these functions directly.

EXIT STATUS

This section contains diagnostic messages from both `ctrace` and `cc/c89`, since the traced code often gets some `cc` warning messages.

DIAGNOSTICS

warning: some variables are not traced in this statement

Only 10 variables are traced in a statement to prevent the C compiler 'out of tree space; simplify expression' error. Use the `-t` option to increase this number.

warning: statement too long to trace

This statement is over 400 characters long. Make sure that tabs are used to indent the code, not spaces.

cannot handle preprocessor code, use `-P` option

This is usually caused by `#ifdef/#endif` preprocessor statements in the middle of a C statement, or by a semicolon at the end of a `#define` preprocessor statement.

'if ... else if' - sequence too long

Split the sequence by removing an `else` from the middle.

possible syntax error, try `-P` option

Use the `-P` option to preprocess the `ctrace` input, along with any appropriate `-D`, `-I`, and `-U` preprocessor options.

NOTES

Defining a function with the same name as a system function may cause a syntax error if the number of arguments is changed. Just use a different name.

`ctrace` assumes that `BADMAG` is a preprocessor macro, and that `EOF` and `NULL` are `#defined` constants. Declaring any of these to be variables, e.g., `"int EOF;"`, will cause a syntax error.

Pointers are always treated as pointers to character strings.

`ctrace` does not know about the components of aggregates like structures, unions, and arrays. When an

assignment is made to the entire aggregate, not all of the individual components will be output. In this case, `ctrace` may choose to print the address of the aggregate or use the wrong format (e.g., `3.149050e-311` for a structure with two integer members) when printing the value of an aggregate.

The loop trace output elimination is done separately for each file of a multi-file program. Separate output elimination can result in functions called from a loop still being traced, or the elimination of trace output from one function in a file until another in the same file is called.

FILES

`/usr/lib/ctrace/runtime.c`
run-time trace package

SEE ALSO

`bfs(1)`, `tail(1)`, `ctype(3C)`, `string(3C)`, `fclose(3S)`, `printf(3S)`.

NAME

`cxref` - generate C program cross-reference table

SYNOPSIS

`cxref` [*option...*] *file...*

DESCRIPTION

The `cxref` (C cross reference) command analyzes a collection of C files and builds a cross-reference table. `cxref` uses a special version of `cc` to include information declared in `#define` statements in its symbol table. It generates a list of all symbols (auto, static, and global) in each individual file, or in combination if the `-c` option is used. The table includes four fields: NAME, FILE, FUNCTION, and LINE. The LINE field contains the numbers of the lines in which the symbol NAME appears, and the nature of the reference in the lines concerned. The following three types of access are marked:

assignment, using =

declaration, using

—

definition, using

*

Other types of access are not marked.

OPTIONS

`cxref` interprets the `-D`, `-I`, `-U` options in the same manner that `cc/c89` does. In addition, `cxref` interprets the following options:

- `-c` Combine the source files into a single report. Without the `-c` option, `cxref` generates a separate report for each file on the command line.
- `-d` Disables the printing of declarations, making the report easier to read.
- `-l` Does not print local variables. Prints only global and file scope statistics.
- `-o file`
Direct output to *file*.
- `-s` Does not print input filenames.
- `-t` Formats listing for 80-column width (default).
- `-w num`
The output is formatted to be no wider than *num* (decimal) columns.
The default value is 80 if *num* is not specified or is less than 51.
- `-C` Runs only the first pass of `cxref`, creating a `.cx` file that can later be passed to `cxref`. `-C` is similar to the `-c` option of `cc` or `lint`.
- `-F` Prints the full path of the referenced filenames.
- `-Lcols`
Modifies the number of columns in the LINE field. If you do not specify a number, `cxref` defaults to five columns.
- `-V` Prints version information on the standard error output.
- `-Ww_name,w_file,w_function,w_line`
Changes the column widths of the fields to the value specified by *w_**. All 4 values must be specified, separated by commas.
The default widths are:

Field
 Characters
 NAME 15
 FILE 13
 FUNCTION
 15
 LINE 20 (4 per column)

— If the first filename begins with a dash (–), the end of the command-line options must be marked with —.

EXAMPLE

```
a.c
1      main()
2      {
3          int i;
4          extern char c;
5
6          i=65;
7          c=(char)i;
8      }
```

Resulting cross-reference table:

a.c:				
NAME	FILE	FUNCTION	LINE	
c	a.c	---	4-	7=
i	a.c	main	3*	6= 7
main	a.c	---	2*	
u3b2	predefined	---	0*	
unix	predefined	---	0*	

DIAGNOSTICS

Error messages usually mean you cannot compile the files.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization – Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

FILES

TMPPDIR/tcx.*

temporary files

TMPPDIR/cx.*

provisional files

LIBDIR/xref

accessed by cxref

LIBDIR

usually /usr/ccs/lib

TMPPDIR

usually /var/tmp, but can be redefined by setting the environment variable `TMPPDIR` [see `tempnam` in

`tmpnam(3S)`].

SEE ALSO

`cc(1)`, `lint(1)`.

NAME

`delta` - make a change to an SCCS file

SYNOPSIS

`delta [-r SID] [-s] [-n] [-g list] [-m MR-list] [-y[comment]] [-p] [—] file...`

DESCRIPTION

`delta` is used to copy into the named SCCS *file* changes that were made to the file created by `get -e` (called the *g*-file or generated file), i.e. to create a new version of it.

If a directory is specified, `delta` processes all the files in the directory, except for any files which are non-SCCS files (i.e. those for which the filename does not begin with *s.*, ignoring the pathname) and unreadable files; these files are ignored without comment. If a name of `-` is given, the standard input is read (see the **NOTES** section); each line of the standard input is taken to be the name of an SCCS file to be processed.

`delta` may issue prompts on the standard output, depending on whether the user specifies certain keyletters and flags that may be present in the SCCS file (see `-m` and `-y` keyletters).

OPTIONS

Keyletter arguments apply independently to each named file.

`-r SID`

SID uniquely identifies which `delta` is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding `get` calls for editing (`get -e`) on the same SCCS file were done by the same person (login name). The *SID* value specified with the `-r` keyletter can be either the *SID* specified on the `get` command line or the *SID* to be made as reported by the `get` command [see [get\(1\)](#)]. An error message results if the specified *SID* is ambiguous or is not specified.

`-s` Suppresses the issue, on the standard output, of the created `delta`'s *SID*, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

`-n` Specifies retention of the edited *g*-file. This is normally deleted on completion of `delta` processing.

`-g list`

A *list* of `deltas` that are to be ignored when the file is accessed at the change level (*SID*) created by this `delta` [for the definition of *list* see [get\(1\)](#)].

`-m MR-list`

If the SCCS file has the *v* flag set [see [admin\(1\)](#)] then a Modification Request (MR) number must be supplied as the reason for creating the new `delta`. If `-m` is not used and the standard input is a screen, the prompt `MRs?` is issued on the standard output before the standard input is read; if the standard input is not a screen, no prompt is issued. The `MRs?` prompt always precedes the `comments?` prompt (see `-y`). MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the *MR-list*.

Note that if the *v* flag has a value, it is taken to be the name of a program or shell procedure that will validate the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, `delta` terminates. It is assumed that the MR numbers were not all valid.

`-y[comment]`

comment is arbitrary text used to describe the reason for producing the `delta`. A null string is considered a valid *comment*. If `-y` is not specified and the standard input is a screen, the prompt `comments?` is issued on the standard output before the standard input is read; if the standard input is not a screen, no prompt is issued. An unescaped new-line character terminates the comment text.

`-p` The SCCS file differences before and after the `delta` is applied are printed on the standard output, in a

`diff(1)` format.

- If the first filename begins with a dash (–), the end of the command-line options must be marked with —.

NOTES

Aget of many SCCS files, followed by a `delta` of those files, should be avoided when the `get` generates a large amount of data. Instead, multiple `get/delta` sequences should be used.

If the standard input is specified by – in the `delta` command line, the `–y` and, if applicable, `–m` keyletters must also be present. Omission of these keyletters causes an error.

Comments are limited to text strings of at most 1024 characters. Line lengths greater than 1000 characters cause undefined results.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to `environ(5)` for information on setting the user environment.

FILES

- `g-file`
Existed before the execution of `delta`; removed after completion of `delta`.
- `p-file`
Existed before the execution of `delta`; may exist after completion of `delta`.
- `q-file`
Created during the execution of `delta`; removed after completion of `delta`.
- `x-file`
Created during the execution of `delta`; renamed to SCCS file after completion of `delta`.
- `z-file`
Created during the execution of `delta`; removed during the execution of `delta`.
- `d-file`
Created during the execution of `delta`; removed after completion of `delta`.
- `bdiff`
Program to compute differences between the "gotten" file and the `g-file`.

SEE ALSO

`admin(1)`, `bdiff(1)`, `cdc(1)`, `diff(1)`, `get(1)`, `help(1)`, `prs(1)`, `rmDEL(1)`, `sccsfile(4)`.

Chapter on "SCCS" in the "Guide to Tools for Programming in C"

NAME

`dis` - object code disassembler

SYNOPSIS

`dis [-o] [-V] [-L] [-d sec] [-D sec] [-F function] [-t sec] [-l string] [—] file...`

DESCRIPTION

The `dis` command produces an assembly language listing of *file*, which may be an object file or an archive of object files. The listing includes assembly statements and an octal or hexadecimal representation of the binary that produced those statements.

OPTIONS

- `-d sec`
Disassemble the named section as data, printing the offset of the data from the beginning of the section.
- `-D sec`
Disassemble the named section as data, printing the actual address of the data.
- `-F function`
Disassemble only the named function in each object file specified on the command line. The `-F` option may be specified multiple times on the command line.
- `-L`
Lookup source labels for subsequent printing. This option works only if the file was compiled with additional debugging information (for example, the `-g` option of `cc`).
- `-l string`
Disassemble the archive file specified by *string*. For example, one would issue the command `dis -l x -l z` to disassemble `libx.a` and `libz.a`, which are assumed to be in `LIBDIR`.
- `-o`
Print numbers in octal. The default is hexadecimal.
- `-t sec`
Disassemble the named section as text.
- `-V`
Print, on standard error, the version number of the disassembler being executed.
- `—`
If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `—`.

If the `-d`, `-D` or `-t` options are specified, only those named sections from each user-supplied filename will be disassembled. Otherwise, all sections containing text will be disassembled.

On output, a number enclosed in brackets at the beginning of a line, such as `[5]`, indicates that the break-pointable line number starts with the following instruction. These line numbers will be printed only if the file was compiled with additional debugging information (for example, the `-g` option of `cc`). An expression such as `<40>` in the operand field or in the symbolic disassembly, following a relative displacement for control transfer instructions, is the computed address within the section to which control will be transferred. A function name will appear in the first column, followed by `()` if the object file contains a symbol table.

NOTES

Since the `-da` option did not adhere to the command syntax rules, it has been replaced by `-D`. At this time, symbolic disassembly does not take advantage of additional information available if the file is compiled with the `-g` function.

FILES

`LIBDIR`

Usually `/usr/ccs/lib`

SEE ALSO

`as(1)`, `cc(1)`.

NAME

`dump` - dump parts of an object file

SYNOPSIS

`dump [ option... ] file...`

DESCRIPTION

The `dump` command dumps selected parts of each of the specified object files. This command will accept both object files and archives of object files.

OPTIONS

- `-a` Dump the archive header of all the elements of an archive.
- `-C` Dump the names in a C++ symbol table.
- `-c` Dump the string table(s).
- `-D` Dump debugging information.
- `-f` Dump each file header.
- `-g` Dump the global symbols in the symbol table of an archive.
- `-h` Dump the section headers.
- `-L` Dump dynamic linking information and static shared library information, if available.
- `-l` Dump line number information.
- `-o` Dump each program execution header.
- `-r` Dump relocation information.
- `-s` Dump section contents in hexadecimal.
- `-T index[,index2]`
Dump only the indexed symbol table entry defined by *index* or a range of entries lying between *index* and *index2*.
- `-t` Dump symbol table entries.
- `-u` When reading a COFF object file, `dump` translates the file to ELF internally. This translation does not affect the file contents. `-u` controls how much translation occurs from COFF values to ELF. When this option is not used, the COFF values are preserved as much as possible, showing the actual bytes in the file. If `-u` is used, `dump` updates the values and completes the internal translation, giving a consistent ELF view of the contents. Although the bytes displayed under this option might not match the file itself, they show how the file would look if it were converted to ELF.
- `-V` Print version details for `dump`.

The following modifiers are used in conjunction with the options listed above to modify their capabilities.

- `-d number[,number2]`
Dump the section number indicated by *number* or the range of sections starting at *number* and ending at *number2*.

This modifier can be used with `-h`, `-s`, and `-r`. When `-d` is used with `-h` or `-s`, the argument is treated as the number of a section or range of sections. When `-d` is used with `-r`, the argument is treated as the number of the section or range of sections to which the relocation applies.

For example, to print out all relocation entries associated with the `.text` section, specify the number of the section as the argument to `-d`. If `.text` is section number 2 in the file, `dump -r -d 2` will print all associated entries. To print out a specific relocation section use `dump -s -n name` for raw data

output, or `dump -sv -n name` for interpreted output.

-n *Name*

Dump information about the object called *Name*. This modifier can be used with `-h`, `-s`, `-r`, and `t`.

When `-n` is used with `-h` or `-s`, the argument will be treated as the name of a section. When `-n` is used with `-t` or `-r`, the argument will be treated as the name of a symbol. For example, `dump -t -n .text` will dump the symbol table entry associated with the symbol whose name is `.text`, where `dump -h -n .text` will dump the section header information for the `.text` section.

-p Suppress printing of the headings.

-v Dump information in symbolic representation rather than numeric.

This modifier can be used with

- `-a` (date, user id, group id),
- `-f` (class, data, type, machine, version, flags),
- `-h` (type, flags),
- `-o` (type, flags),
- `-r` (name, flags),
- `-s` (interpret section contents wherever possible),
- `-t` (type, bind), and
- `-L` (value).

When `-v` is used with `-s`, all sections that can be interpreted, such as the string table or symbol table, will be interpreted.

For example,

```
dump -sv -n .symtab files
```

will produce the same formatted output as

```
dump -tv files,
```

but

```
dump -s -n .symtab files
```

will print raw data in hexadecimal. Without additional modifiers, `dump -sv files` will dump all sections in the specified files, interpreting all those that it can. The rest, such as `.text` or `.data` are dumped as raw data.

The `dump` command attempts to format the information it dumps in a meaningful way, printing certain information in character, hexadecimal, octal or decimal representation as appropriate.

SEE ALSO

`a.out(4)`, `ar(4)`.

NAME

`exstr` - extract strings from source files

SYNOPSIS**Format 1**

`exstr [-e] file ...`

Format 2

`exstr -r [-d] file ...`

DESCRIPTION

`exstr` is used to extract strings from C language source files (**Format 1**) and replace them by calls to a function (**Format 2**) that retrieves matching strings from a message catalog at program runtime [see `gettext(3C)`]. This utility will extract all strings of one or more printable characters surrounded by double quotes ("..."), not just strings used as arguments to `printf` commands.

Format 1: Extract strings from C source files

`exstr [-e] file ...`

No option specified:

`exstr` finds all strings in the named source *file* and writes them to standard output. Each string is preceded by the source file name and a colon.

`-e` `exstr` lists the strings that it finds plus information about their position in *file*. This list consists of lines made up of the following fields:

file : *line* : *column* : *msgfile* : *msgnum* : *string*

file The name of the C language source file

line Line number in the *file* containing *string*

column

Column number at which *string* begins

msgfile

Empty field

msgnum

Empty field

string

The extracted text string

You normally redirect the output to a file which you can subsequently edit in order to add a message file name and a message number to the *msgfile* and *msgnum* fields.

msgfile

Name of the file containing the text strings that are to replace the original strings in the C source files. A file with this name must be created and installed at the appropriate place in the file tree by using the `mkmsgs` command [see `mkmsgs(1)`].

msgnum

The sequence number of the *string* in *msgfile*.

To replace strings in the source file you use the second format of `exstr`.

file Name of the C source file in which `exstr` is to search for strings. You may name more than one file.

Format 2: Replace strings in C source files

`exstr -r [-d] file ...`

- `-r` `exstr` replaces each string in *file* with a call to the `gettext(3C)` function, which retrieves a matching string at program runtime.
- `-d` This option must only be used together with the `-r` option. If the `gettext(3C)` function call fails at program runtime, the extracted string is printed instead. This option is set by default; it is retained to provide compatibility with other systems.
- file* Name of the C source file in which `exstr` is to search for strings. You may name more than one file.

Application usage

`exstr` is useful for application programs that need to run in an international environment and print messages in more than one language. `exstr` replaces text strings in C source files with function calls that point at strings in a message database. The database used depends on the runtime value of the `LC_MESSAGES` environment variable [see `environ(5)`].

Procedural steps with `exstr`

- First, use `exstr -e file.c > out_file` to extract the required strings from a named C source file (*file.c*) and to save them together with positional information in a file named *out_file*.
- Next, examine the list of strings in *out_file* and determine which strings can be translated to another language and subsequently retrieved by the message retrieval function `gettext(3C)`.
- Then, modify this file by deleting lines that cannot be translated and, for lines that can be translated, by adding the message file names and the appropriate message numbers to the *msgfile* and *msgnum* fields. The message files named must have been created beforehand with `mkmsgs` and must reside in the directory `/usr/lib/locale/Locale/LC_MESSAGES`. The basename of the directory `/usr/lib/locale/Locale`, i.e. *Locale*, corresponds to the language in which the text strings are written in the message files [see `setlocale(3C)`]. The message numbers used must correspond to the sequence numbers of strings in the message files.
- Now use your modified file (*out_file*) as input to `exstr -r` to produce a new version (*new_file.c*) of the original C language source file *file.c*, in which the strings have been replaced by calls to the message retrieval function `gettext(3C)`. This can be done by issuing the following call:

```
exstr -r file.c < out_file > new_file.c
```

Two arguments are passed to the `gettext(3C)` function. The first argument consists of the fields *msgfile* and *msgnum* in *out_file*; the second argument consists of the strings to be printed if the call to `gettext(3C)` fails at runtime. This argument is the original string (see above).

Restrictions

`exstr` cannot be used to replace all types of strings in C source files. For example:

- `Astatic` initialized character string cannot be replaced by a function call.
- Strings in the form of escape sequences cannot be translated into another language and should therefore not be replaced using `exstr`. In order to prevent invalid code from being generated, the output produced by invoking `exstr -e` must be examined, and lines containing strings that are not directly replaceable by function calls to `gettext(3C)` must be deleted or modified accordingly.

ERROR MESSAGES

The most important error messages are listed below. They are intended to be self-explanatory.

```
ERROR: cannot replace string '%s' in line (%d) of file (%s)
```

```
ERROR: stdin: invalid message number '%s'
```

```
ERROR: stdin: invalid message file name '%s'
```

```
ERROR: stdin: badly formed replacement string
```

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The following example shows how `exstr` can be used:

Let us assume that the C source file `file.c` contains two strings:

```
main()
{
    printf("First string\n");
    printf("Second string\n");
}
```

When `exstr` is invoked (without options) with the argument `file.c`, it extracts the strings enclosed in double quotes from the named file and prints them on the standard output.

```
$ exstr file.c
file.c:First string\n
file.c:Second string\n
```

The strings themselves and supplementary information about their positions in the file can be obtained with the `-e` option. To examine the extracted strings and possibly delete or modify them, you must redirect the standard output to a file named `out_file`.

```
$ exstr -e file.c > out_file
$ cat out_file
file.c:3:8::First string\n
file.c:4:8::Second string\n
```

Each output line consists of six colon-separated fields. Field 1 contains the name of the C source file; field 2 the numbers of the lines containing the strings (3 and 4 in this case); field 3 the column number of the first character in the string (8 in this example); field 6 the string itself without double quotes. Fields 4 and 5 you can edit to define the message file from which `gettext(3C)` is to retrieve the new string at runtime and the message number of the string in the message file, respectively. If `UX` is the name of the message file, and the numbers 1 and 2 represent the sequence numbers of the strings in the file, here is what `out_file` should look like after you have added this information:

```
file.c:3:8:UX:1:First string\n
file.c:4:8:UX:2:Second string\n
```

You can now call `exstr -r` to replace the strings in the C source file by calls to the message retrieval function `gettext(3C)`. The modified file `out_file` is passed as an input argument to `exstr`. The output is redirected to `new_file.c`, which will subsequently contain the following new source code:

```
$ exstr -r file.c < out_file > new_file.c
$ cat new_file.c
extern char *gettext();
main()
{
    printf(gettext("UX:1", ""));
    printf(gettext("UX:2", ""));
}
```

If the `-d` option is also specified in this call, the string extracted from the original C source file will be passed to `gettext(3C)` as the second argument instead of the null string `""`. This string will then be output at program runtime if the call to `gettext(3C)` fails:

```
$ exstr -rd file.c < out_file > new_file.c
$ cat new_file.c
extern char *gettext();
main()
```

```
{
    printf(gettxt("UX:1", "First string\n"));
    printf(gettxt("UX:2", "Second string\n"));
}
```

FILES

`/usr/lib/locale/Locale/LC_MESSAGES`

Directory containing the message files, where the basename of the directory

`/usr/lib/locale/Locale`, i.e. *Locale*, corresponds to the language in which the text strings are written in the message file.

SEE ALSO

`gettext(1)`, `mkmsgs(1)`, `printf(1)`, `srchtxt(1)`, `gettext(3C)`, `setlocale(3C)`, `printf(3S)`, `environ(5)`.

Programmer's Guide: Internationalization – Localization.

NAME

`extract`, `strextact` - interactive string extract and replace

SYNOPSIS

`extract` [*option ...*] *file ...*

DESCRIPTION

`extract` is used to interactively extract strings from source files and replace them with calls to a function that retrieves matching strings from a message catalog at program runtime. This process is controlled by two files:

- `apattern` file
- an ignore file

The pattern file contains

1. text patterns with which strings from the specified source file are to be compared, and
2. rules by which strings that match the patterns are extracted and rewritten.

The ignore file contains strings that are not to be rewritten in the source.

OPTIONS

—*i* *ignorefile*

Uses the file named *ignorefile* as the ignore file.

—*i* *ignorefile* not specified:

The default ignore file is called `ignore`. This file is looked for in the current directory, then in the home directory, and finally in `/usr/lib/nls/extract`.

—*m* *prefix*

For *prefix* you can specify characters or strings to be placed before all message numbers in the `nl_file` and in the `.msf` file.

—*m* *prefix* not specified:

No prefix is added to the message numbers.

—*n* Creates a separate message catalog source file for each file given in the file list. Message numbers in each message file start with 1.

—*n* not specified:

Only one message catalog source file is created, with the messages being numbered sequentially.

—*p* *patternfile*

Uses the file named *patternfile* as the pattern file.

—*p* *patternfile* not specified:

The default pattern file is called `pattern`. This file is looked for in the current directory, then in the home directory, and finally in `/usr/lib/nls/extract`.

—*s* *string*

For *string* you can specify characters to be output at the start of the `.msf` file.

—*s* *string* not specified:

The default output string is `$quote "`, or if defined, the `CATHEAD` directive from the pattern file.

—*u* Uses a message file produced by a previous run of `strextact`. For each given file, the current directory is scanned for a corresponding file with the suffix `.msg`. This file contains detailed information on all strings that match patterns in the pattern file, including, among other things, their respective positions and line numbers in the file. `extract` calls `strextact` and interprets its output.

file Name of the source file to be processed by `extract`. More than one file may be specified.

The pattern file

The pattern file contains text patterns to be matched with strings in the specified source file and rules by which matching strings are to be extracted and rewritten. You can also define patterns for strings which you do not want to have replaced, so as to prevent strings being illegally modified and thus producing illegal code.

The syntax of text patterns is identical to the regular expression syntax used in `ed`. The example below illustrates a pattern file for use with C source programs.

The pattern file is split into several sections, each section uniquely identified by `$KEYWORD` at the start of a line.

`$KEYWORD` is one of the following:

`$SRCHEAD1`

Header file to be inserted at the beginning of the first new source file.

`$SRCHEAD2`

Header file to be inserted at the beginning of the second new source file.

`$CATHEAD`

Header file to be inserted at the beginning of the message catalog to be produced. Here you might want to include the SCCS ID, the definition of the current message set, and so on. You can override this by using the `-s` option.

`$REWRITE`

Rule for rewriting matching strings.

`$MATCH`

List of patterns for matching strings. The regular expression syntax is based on `ed`. Note: `[^xyz]*` means that any character other than x, y or z will match the pattern.

`$REJECT`

Various special C constructs are to be disregarded.

`$ERROR warning`

A warning is printed for initialized strings.

The ignore file

The ignore file contains literal strings (rather than patterns in the form of regular expressions) which are to be left unchanged in the new source file (the `n1_` file). This file, if present, is read at startup and causes specific strings in the input file to be ignored. During a run, further strings to be ignored can be added to this file by using the `ADD` command (see below). The ignore file can be created and edited using an ordinary editor, with each string to be ignored in the input file being entered on a separate line.

Mode of operation

When `extract` is called, a new version of the source file is produced with the characters `n1_` prepended to the original file name. A message catalog source file is also produced; its name is derived from the first file given by dropping the `.c` ending and adding the suffix `.msf`. This file can be input directly to `gencat`, or its contents can be translated into another native language before being passed to `gencat`.

When run, `extract` displays three windows on the terminal. The first contains the source code with the string to be extracted and converted. This string is displayed in inverse video.

The second window contains a list of the strings extracted so far. Each string is displayed with a message number.

The third window contains a list of the available commands. The recognized commands are:

`EXTRACT`

Extract the matching string from the source file, putting it into the catalog source file. Use the `REWRITE` rule in the pattern file to determine what text should replace the extracted string in the `n1_` file.

`DUPLICATE`

If a given string has already been extracted, `extract` indicates this fact by displaying it in inverse video.

Entering the `DUPLICATE` command in such a case instructs `extract` to rewrite the source program using the same message number as for the previously extracted string. This is a way of saving storage space in message catalogs.

IGNORE

Ignore this and any subsequent occurrence of the string.

PASS Skip (ignore) this occurrence of the string. Every subsequent occurrence of the string will still be offered to the user for a new selection.

ADD Same as `IGNORE`, except that the string is also added to the ignore file.

COMMENT

You can insert a comment, which will then be entered in the message source file (suffix `.msf`). This is useful for `gencat`.

QUIT Exit the program after confirming whether the user really wants to quit. The output file will contain the results of string extraction up to the point of quitting. Unfortunately, it is not yet possible to restart `extract` on a source file that has already been partially processed (with `extract`).

HELP Display a summary of available commands. The help texts are located in `/usr/lib/nls/extract/help`.

The current command is displayed in inverse video and is executed if you simply press the return key `<RETURN>`. To select a new command you enter the first character of its name: e.g. `i` or `I` for `IGNORE`.

Restrictions

- The current syntax for the pattern file does not allow strings in multi-line comments to be ignored.
- Only one rewrite string can be specified for all classes of pattern matches.
- The program does not recursively descend through all header files. These must be processed individually.
- Multiline strings are not recognized.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

Answers to yes/no queries must be given in the language appropriate to the current locale.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

Example 1

Execution of `extract` on the source program `file1.c` using `newignore` as the ignore file and `c_patterns` as the pattern file:

```
$ extract -i newignore -p c_patterns file1.c
```

The following files are created:

a new version of the source file: `nl_file1.c`

the message text file: `file1.msf`

```
$ ls *file1*
file1.c
file1.msf
nl_file1.c
```

Example 2

Use of `gencat` to generate the message catalog `file1.cat` for `file1.c` from the message text source file

```
file1.msf:
```

```
$ gencat file1.cat file1.msf
```

Example 3

Compilation of the new source file `nl_file1.c` generated by `extract`:

```
$ cc nl_file1.c
```

Example 4

Example of a pattern file for a C source program:

```
# Comment lines begin with the hash character #, which can be
# escaped with a backslash \#
#
# First we have a few sections to be included in each new source
# file to be created
#
# Header file to be inserted at the beginning of the first
# new source file
$SRCHHEAD1
\#include <nl_types.h>
nl_catd catd;          /* definition of the catalog descriptor */
# Header file to be inserted at the beginning of the
# second new source file
$SRCHHEAD2
\#include <nl_types.h>
extern nl_catd catd;    /* the catalog descriptor          */
#
# The next header is added to the beginning of the message catalog
# to be produced. You could, for example, put the SCCS ID,
# the definition of the current message set, and so on here.
# However, you can override this by calling extract with
# the -s option.
#
$CATHEAD
\ $quote "
\ $set 1
#
# Rule by which selected matching strings are to be
# rewritten
$REWRITE
catgets(catd, 1, %n, %t)
# The % descriptors are rewritten as follows:
# % - a %
# n - message number (from the cat file)
# t - actual text
# l - length of the text string %t
# r - raw text with quotes removed
# N - insert a newline character
# T - insert a tab character
#
# List of patterns to be matched by strings.
# The regular expression syntax is based on ed.
# Note: [^xyz]* matches any character except xyz.
#
$MATCH
#
" [^"]*"
#
# Some special C constructs are to be ignored.
```



```

$REJECT
# The null string
""
# Output control characters
"\.\"
# Ignore preprocessor directives
\#.*
# Ignore SCCS ID lines
[sS][cC][cC][sS][iI][dD]\\[ ]*=[ ]*\".*\"
# Ignore some library functions that use file names
fopen[ ]*([^\,]*,[^)]*)
creat[ ]*([^\,]\",[^\)]*)
#
# Issue a warning for initialized strings.
#
$ERROR Initialized strings cannot be replaced.
# Print the message following ERROR when the strings below are
# found:
^char[^\=]*=[ ]*\"^[^\"]*\"
char[ ]*[A-Za-z][A-Za-z0-9]*\\[[^\\\]*\\][ ]*=[ ]*\"^[^\"]*\"

```

SEE ALSO

[ed\(1\)](#), [gencat\(1\)](#).

Programmer's Guide: Internationalization – Localization.

NAME

`get` - `get` a version of an SCCS file

SYNOPSIS

`get` [*option ...*] *file ...*

DESCRIPTION

`get` generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments. The options may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, `get` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (i.e. those for which the filename does not begin with `s.`, ignoring the path name) and unreadable files; these files are ignored without comment. If a name of `-` is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed.

The generated text is normally written into a file called the `g`-file whose name is derived from the SCCS filename by simply removing the leading `s.` (see the **FILES** section).

OPTIONS

Each of the keyletters is explained below as though only one SCCS file is to be processed, but the effects of any keyletter apply to each named file.

`-r` *SID*

The SCCS identification string (SID) of the version (delta) of an SCCS file to be retrieved.

The table below shows what version of an SCCS file is retrieved, as a function of the SID specified.

`-c` *cutoff*

Cutoff point (date/time) after which changes are to be ignored, in the form:

`YY[MM[TT[HH[MM[SS]]]]]`

No changes (deltas) to the SCCS file that were created after the specified *cutoff* date-time are included in the generated ASCII text file. Parts of the *cutoff* which are not specified default to their maximum possible values; that is, `-c 7502` is equivalent to `-c 750228235959`. Any number of non-numeric characters may separate the two-digit parts of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the following form: `-c "77/2/2 9:22:25"`.

Note: If the two-digit year notation *YY* is in the range 70 to 99, the 20th century is assumed; if it is in the range 00 to 37, the 21st century is assumed. All other entries are rejected.

`-i` *list*

A list of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

`<list> ::= <range> | <list> , <range>`

`<range> ::= SID | SID - SID`

SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of the table.

`-x` *list*

A list of deltas to be excluded in the creation of the generated file. The format of the *list* is as for `-i`.

`-e`

Indicates that the `get` is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of `delta(1)`. The `-e` keyletter used in a `get` for a particular version (SID) of the SCCS file prevents further calls of `get` for editing on the same SID until `delta` is executed or the `j` (joint edit) flag is set in the SCCS file [see `admin(1)`]. Concurrent use of `get -e` for different SIDs is

always allowed.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file [see `admin(1)`] are enforced when the `-e` keyletter is used.

- `-b` Used together with the `-e` keyletter to indicate that the new delta should have a SID in a new branch as shown in the table. This keyletter is ignored if the `b` flag is not set in the file [see `admin(1)`] or if the retrieved `delta` is not a leaf `delta` (a leaf `delta` is one that has no successors on the SCCS file tree). A branch `delta` may always be created from a non-leaf `delta`. Partial SIDs are interpreted as shown in the "SID Retrieved" column in the table below.
- `-k` Suppresses replacement of keywords in the retrieved text by their value. The `-k` keyletter is implied by the `-e` keyletter.
- `-l` Causes a delta summary to be written into an `l`-file. The format of the `l`-file will be found in the section on **Keywords**.
- `-L` The delta summary is written to the standard output. Additional information that is normally written to the standard output is written to the standard error output in this case.
This corresponds to the old option `-lp`, which will continue to be supported.
- `-p` The text retrieved from the SCCS file is written on the standard output. No `g`-file is created. All output that normally goes to the standard output goes to the standard error output instead, unless the `-s` keyletter is used. In the latter case these outputs will disappear.
- `-s` Suppresses all output normally written on the standard output. However, fatal error messages, which always go to the standard error output, remain unaffected.
- `-m` Each text line retrieved from the SCCS file is preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.
- `-n` Each generated text line is preceded by the `%M%` keyword value. The format is: `%M%` value, followed by a horizontal tab, followed by the text line. When both the `-m` and `-n` keyletters are used, the format is: `%M%` value, followed by a horizontal tab, followed by the `-m` keyletter generated format.
- `-g` Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an `l`-file, or to verify the existence of a particular SID.
- `-t` Used to access the most recently created delta in a given release (e.g., `-r□1`), or release and level (e.g., `-r□1.2`).
- `-w string`
Substitute *string* for all occurrences of `%W%` when getting the file. Substitution occurs prior to keyword expansion.
- `-a seq-no.`
The delta sequence number of the SCCS file delta (version) to be retrieved. This keyletter is used by the `comb` command. If both the `-r` and `-a` keyletters are specified, only the `-a` keyletter is used. Care should be taken when using the `-a` keyletter in conjunction with the `-e` keyletter, as the SID of the delta to be created may not be what one expects. The `-r` keyletter can be used with the `-a` and `-e` keyletters to control the naming of the SID of the delta to be created.
- `---` If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `---`.

For each file processed, `get` responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the `-e` keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each filename is printed, preceded by a new-line, before it is processed. If the `-i` keyletter is used, included deltas are listed following the notation `Included`; if the `-x` keyletter is used, excluded deltas are listed following the notation `Excluded`.

SID specified ¹⁾	-b keyletter used ⁴⁾	other conditions	SID retrieved	SID of delta to be created
none ⁵⁾	no	R defaults to mR	mF.mL	mR.(mL+1)
none ⁵⁾	yes	R defaults to mR	mF.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1 ³⁾
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	—	R < mR and R does not exist	hR.mL ²⁾	hR.mL.(mB+1).1
R	—	Trunk successor in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	—	Trunk successor in release R	R.L	R.L(mB+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	—	Branch successor	R.L.B.S	R.L.(mB+1).1

- 1) R, L, B, or S are the release (version), level, branch and sequence components of the SID, respectively; m means maximum. Thus for example, R.mL means the maximum level number within release R; R.L.(mB+1).1 means the first sequence number on the new branch (i.e., maximum branch number plus one) of level L within release R. Note that if the SID specified is of the form R.L, R.L.B, or R.L.B.S, each of the specified components must exist.
- 2) hR is the highest existing release that is lower than the specified, nonexistent, release R.
- 3) This is used to force creation of the first delta in a new release.
- 4) The -b keyletter is effective only if the b flag [see [admin\(1\)](#)] is present in the file. An entry of — means "irrelevant".
- 5) This case applies if the d (default SID) flag is not present in the file. If the d flag is present in the file, then the SID obtained from the d flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

Keywords

Identifying information is inserted into the text retrieved from the SCCS file by replacing identification keywords with their value wherever they occur. If the keyletters -b or -k are used, the keywords in the text are not expanded. The following keywords may be used in the text stored in an SCCS file:

- %M% Module name: either the value of the m flag in the file [see [admin\(1\)](#)], or if absent, the name of the SCCS file with the leading s removed.
- %I% SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
- %R% Release
- %L% Level
- %B% Branch
- %S% Sequence
- %D% Current date (YY/MM/DD)

%H% Current date (MM/DD/YY)
 %T% Current time (HH:MM:SS)
 %E% Date newest applied delta was applied (YY/MM/DD)
 %G% Date newest applied delta was applied (YY/MM/DD)
 %U% Time newest applied delta was created (HH:MM:SS)
 %Y% Module type: value of the `t` flag in the SCCS file [see `admin(1)`]
 %F% SCCS filename
 %P% Full pathname of the SCCS file
 %Q% The value of the `q` flag in the file [see `admin(1)`]
 %C% Current line number
 %Z% The four-character string `@(##)` recognizable by the `what` command
 %W% A shorthand notation for constructing `what` strings for Reliant UNIX System program files:
 %W%□=□%Z%%M%<Horizontal-Tab>%I%
 %A% Another shorthand notation for constructing `what` strings for non Reliant UNIX System program files:
 %A% = %Z%%Y% %M% %I%%Z%

Several auxiliary files may be created by `get`. These files are called the `g`-file, `l`-file, `p`-file, and `z`-file. The letter before the hyphen is called the tag. An auxiliary filename is formed from the SCCS filename: the filename must be of the form `s.modulename`, the auxiliary files are named by replacing the leading `s` with the tag.

The `g`-file is an exception to this scheme: the name of the `g`-file is formed by removing the `s.` prefix. For example, for `s.xyz.c` the auxiliary filenames would be `xyz.c`, `l.xyz.c`, `p.xyz.c`, and `z.xyz.c`, respectively.

The `g`-file, which contains the generated text, is created in the current directory, unless the `-p` keyletter is used. A `g`-file is created in all cases, whether or not any lines of text were generated by the `get`. It is owned by the real user. If the `-k` keyletter is used or implied, its mode is 644; otherwise its mode is 444 (provided the permissions have not been restricted by `umask`). Only the real user need have write permission in the current directory.

The `l`-file contains a list of the deltas applied in generating the retrieved text. The `l`-file is created in the current directory if the `-l` keyletter is used; its mode is 444 (provided the permissions have not been restricted by `umask`) and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the `l`-file have the following format:

```
<a><b><c> <SID><TAB><date time> <login>
```

Whereby:

- a* A blank character if the delta was applied; * otherwise.
- b* A blank character if the delta was applied or was not applied and ignored; * if the delta was not applied and was not ignored.
- c* A code indicating a "special" reason why the delta was or was not applied: "I" (included), "X" (excluded), or "C" (cut off by a `-c` keyletter).

SID SCCS identification (SID).

date time

Date and time (in the form `YY/MM/DD HH:MM:SS`) of creation.

login Login name of person who created delta.

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The `p`-file is used to pass information resulting from a `get` with an `e` keyletter along to `delta`. Its contents are also used to prevent a subsequent execution of `get` with an `-e` keyletter for the same SID until `delta` is

executed or the joint edit flag, `j`, [see `admin(1)`] is set in the SCCS file. The `p`-file is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. Any line in the `p`-file has the following format:

```
SID new_SID login date/time_of_get [argument_for_-i] [argument_for_-x]\n
```

There can be an arbitrary number of lines in the `p`-file at any time; no two lines can have the same new delta SID.

The `z`-file serves as a lock-out mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (i.e., `get`) that created it. The `z`-file is created in the directory containing the SCCS file for the duration of `get`. The same protection restrictions as those for the `p`-file apply for the `z`-file. The `z`-file is created with mode 444.

NOTES

If the effective user has write permission in the directory containing the SCCS files, but the real user does not, then only one file may be named when the `-e` keyletter is used.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`. When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization – Localization". Refer also to `environ(5)` for information on setting the user environment.

FILES

s-file: prefix: `s`. (e.g. `s.xyz.c`)

Contains the current information stored in SCCS on the contents and history of a module

Is created by `admin -n`, and never deleted. The directory containing the `s`-file is identified as an administration directory below.

g-file: prefix: none (e.g. `xyz.c`)

Contains an expression of the module stored in the `s`-file, i.e. the current status of a source file.

Is created in the local directory by `get` and deleted by `unget` or `delta` (exception: if the `-n` switch is used for `unget` or `delta`, the `g`-file is retained).

p-file: prefix: `p`. (e.g. `p.xyz.c`)

Contains information on current processing of the module (who is processing which SID). Prevents parallel access to the same SID of the same module.

Is created in the administration directory by `get -e` and deleted by `unget` or `delta`.

l-file: prefix: `l`. (e.g. `l.xyz.c`)

Contains information in table format as to which deltas were used for creating a `g`-file.

Is created in the local directory by `get -l` (in addition to the `g`-file).

z-file: prefix: `z`. (e.g. `z.xyz.c`)

Temporary lock file, prevents simultaneous write access to the files in the administration directory.

Is created by all write SCCS commands (`get -e`, `unget`, `delta`, `admin`) and deleted after access is terminated, and is stored in the administration directory.

x-file: prefix: `x`. (e.g. `x.xyz.c`)

Temporary file, created as a copy of the `s`-file in the administration directory. Changes to the `s`-file are made in this temporary copy. This copy is then renamed as the `s`-file after the action has been performed successfully. This prevents the `s`-file from being destroyed in the event of an error during write access.

Is also used by `admin` or `delta`.

SEE ALSO

`admin(1)`, `bdiff(1)`, `delta(1)`, `help(1)`, `prs(1)`, `what(1)`.

NAME

`help` - ask for help with message numbers or SCCS commands

SYNOPSIS

`help` [*argument* ...]

DESCRIPTION

`help` finds information to explain a message from a command or explain the use of a SCCS command. Zero or more arguments may be supplied. If no arguments are given, `help` will prompt for one.

The arguments may be either information within the parentheses following a message or SCCS command names.

The program will provide further details about *argument*.

If `help` is not successfully executed, try entering "help stuck".

FILES

LIBDIR/help

directory containing files of message text

LIBDIR/help/helploc

file containing locations of help files not in *LIBDIR*/help

LIBDIR

usually /usr/ccs/lib

NAME

`install` - install files

SYNOPSIS

`/usr/sbin/install` [*option...*] *file* [*directory...*]

DESCRIPTION

The `install` command is most commonly used in "makefiles" [see [make\(1\)](#)] to install a *file* (updated target file) in a specific place within a file system. Each *file* is installed by copying it into the appropriate directory, thereby retaining the mode and owner of the original command. The program prints messages telling the user exactly what files it is replacing or creating and where they are going.

If no *options* or *directories* are specified, `install` will search a set of default directories (`/bin`, `/usr/bin`, `/etc`, `/lib`, and `/usr/lib`, in that order) for a file with the name *file*. If a file is found, `install` issues a message saying that it is overwriting that file with *file*. If no file is found, the program states this and exits without further action.

If a *directory* or *directories* are specified after *file*, those directories will be searched before the directories specified in the default list.

OPTIONS

`-c` *dira*

Installs a new command (*file*) in the directory specified by *dira*, only if it is not found. If it is found, `install` issues a message saying that the file already exists, and exits without overwriting it. May be used alone or with the `-s` option.

`-f` *dirb*

Forces *file* to be installed in given directory, whether or not one already exists. If the file being installed does not already exist, the mode and owner of the new file will be set to `755` and `bin`, respectively. If the file already exists, the mode and owner will be that of the already existing file. May be used alone or with the `-o` or `-s` options.

`-i`

Ignores default directory list, searching only through the specified *directories*. May be used alone or with any other options except `-c` and `-f`.

`-n` *dirc*

If *file* is not found in any of the searched directories, it is put in the directory specified in *dirc*. The mode and owner of the new file will be set to `755` and `bin`, respectively. May be used alone or with any other options except `-c` and `-f`.

`-m` *mode*

The mode of the new file is set to *mode*.

`-u` *user*

The owner of the new file is set to *user*.

`-g` *group*

The group id of the new file is set to *group*. Only available to the superuser.

`-o`

If *file* is found, this option saves the "found" file by copying it to `OLDfile` in the directory in which it was found. This option is useful when installing a frequently used file such as `/bin/sh` or `/lib/saf/ttymon`, where the existing file cannot be removed. May be used alone or with any other options except `-c`.

`-s`

Suppresses printing of messages other than error messages. May be used alone or with any other options.

SEE ALSO

`make(1)`.

install(1-ucb)
BSD Compatibility Package**NAME**

`install` - install files

SYNOPSIS

`/usr/ucb/install [-cs] [-g group] [-mmode] [-o owner] file1 file2`

`/usr/ucb/install [-cs] [-g group] [-mmode] [-o owner] file ... directory`

`/usr/ucb/install -d [-g group] [-m mode] [-o owner] directory`

DESCRIPTION

`install` is used within makefiles to copy new versions of files into a destination directory and to create the destination directory itself.

The first two forms are similar to the `cp(1)` command with the addition that executable files can be stripped during the copy and the owner, group, and mode of the installed file(s) can be given.

The third form can be used to create a destination directory with the required owner, group and permissions.

In the first two forms, `install` will create the required target directory if it does not exist.

Note: `install` uses no special privileges to copy files from one place to another. The implications of this are:

- You must have permission to read the files to be installed.
- You must have permission to copy into the destination file or directory.
- You must have permission to change the modes on the final copy of the file if you want to use the `-m` option to change modes.
- You must be superuser if you want to specify the ownership (or group) of the installed file with `-o`. If you are not the super-user, or if `-o` (or `-g`) is not in effect, the installed file will be owned by you, regardless of who owns the original.

OPTIONS

`-g group`

If run as root, set the group of the installed file to the group-ID of *group* (bin by default if super-user).

`-m mode`

Set the mode for the installed file or directory (0755 by default).

`-o owner`

If run as root, set the ownership of the installed file to the user-ID of *owner* (bin by default if super-user).

`-c`

This option is retained for backwards compatibility but is otherwise ignored.

`-s`

Strip executable files as they are copied.

`-d`

Create a directory. Missing parent directories are created as required as in `mkdir -p`. If the directory already exists, the owner, group and mode will be set to the values given on the command line (must be super-user to set owner and group).

SEE ALSO

`chown(1)`, `chgrp(1)`, `chmod(1)`, `cp(1)`, `install(1)`, `mkdir(1)`, `strip(1)`.

NAME

ld - link editor for object files

SYNOPSIS

ld [*option ...*] *file ...*

DESCRIPTION

The **ld** command combines relocatable object files, performs relocation, and resolves external symbols. **ld** operates in two modes, static or dynamic, as governed by the **-d** option. In static mode, **-dn**, relocatable object files given as arguments are combined to produce an executable object file; if the **-r** option is specified, relocatable object files are combined to produce one relocatable object file. In dynamic mode, **-dy**, the default, relocatable object files given as arguments are combined to produce an executable object file that will be linked at execution with any shared object files given as arguments; if the **-G** option is specified, relocatable object files are combined to produce a shared object. In all cases, the output of **ld** is left in **a.out** by default.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. The library may be either a relocatable archive or a shared object. For an archive library, only those routines defining an unresolved external reference are loaded. The archive library symbol table [see [ar\(4\)](#)] is searched sequentially with as many passes as are necessary to resolve external references that can be satisfied by library members. Thus, the ordering of members in the library is functionally unimportant, unless there exist multiple library members defining the same external symbol. A shared object consists of a single entity, all of whose references must be resolved within the executable being built or within other shared objects with which it is linked.

If other shared objects were linked when creating a shared object, these will be indicated as **DT_NEEDED** entries in the shared object. Such second-level libraries are searched automatically by the linker, as if they had been specified in the command line as a library. The search sequence in this case is: First, all libraries specified in the command line, then those libraries specified in the **DT_NEEDED** entries in a sequence called "breadth first ordering".

OPTIONS

-a In static mode only:

Produces an executable object file; gives error messages for undefined references. This is the default behavior for static mode. **-a** may not be used with the **-r** option.

-b In dynamic mode only:

When creating an executable, does not do special processing for relocations that reference symbols in shared objects. Without the **-b** option, the link editor will create special position-independent relocations for references to functions defined in shared objects and will arrange for data objects defined in shared objects to be copied into the memory image of the executable by the dynamic linker at run time. With the **-b** option, the output code may be more efficient, but it will be less shareable.

-d[y|n]

When **-dy** (the default) is specified, **ld** uses dynamic linking; when **-dn** is specified, **ld** uses static linking.

-e *epsym*

Sets the entry point address for the output file to be that of the *epsym* symbol.

-h *name*

In dynamic mode only:

When building a shared object, records *name* in the object's dynamic section. *name* will be recorded in executables that are linked with this object rather than the object's Reliant UNIX system filename.

Accordingly, *name* will be used by the dynamic linker as the name of the shared object to search for at run time.

- lx Searches a `libx.so` or `libx.a` library. These are the conventional names for shared object and archive libraries, respectively. In dynamic mode, unless the `-Bstatic` option is in effect, `ld` searches each directory specified in the library search path for a file `libx.so` or `libx.a`. The directory search stops at the first directory containing either. `ld` chooses the file ending in `.so` if `lx` expands to two files whose names are of the form `libx.so` and `libx.a`. If no `libx.so` is found, then `ld` accepts `libx.a`. In static mode, or when the `-Bstatic` option is in effect, `ld` selects only the file ending in `.a`. A library is searched when its name is encountered, so the placement of `-l` is significant.
- m Produces a memory map or listing of the input/output sections on standard output.
- o *outfile*
Produces an output object file named *outfile*. The name of the default object file is `a.out`.
- r Combines relocatable object files to produce one relocatable object file. `ld` will not complain about unresolved references. This option cannot be used in dynamic mode or with `-a`.
- s Strips symbolic information from the output file. The debug and line sections and their associated relocation entries will be removed. Except for relocatable files or shared objects, the symbol table and string table sections will also be removed from the output object file.
- t Turns off the warning about multiply defined symbols that are not the same size.
- u *symname*
Enters *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from an archive library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. The placement of this option on the command line is significant; it must be placed before the library that will define the symbol.
- z *defs*
Forces a fatal error if any undefined symbols remain at the end of the link. This is the default when building an executable. It is also useful when building a shared object to assure that the object is self-contained, that is, that all its symbolic references are resolved internally.
- z *nodefs*
Allows undefined symbols. This is the default when building a shared object. It may be used when building an executable in dynamic mode and linking with a shared object that has unresolved references in routines not used by that executable. This option should be used with caution.
- z *text*
In dynamic mode only:
Forces a fatal error if any relocations against non-writable, allocatable sections remain.
- B[*dynamic|static*]
Controls the inclusion of libraries. `-Bdynamic` is valid in dynamic mode only. These options may be specified any number of times on the command line as toggles: if the `-Bstatic` option is given, no shared objects will be accepted until `-Bdynamic` is seen. See also the `-l` option.
Warning: If the last option you specify when using the `-B` options is `-Bstatic`, the `libc` standard library will be used as the static library.
- Bcount_multigots=*number*
Used in conjunction with `-Bmultigot` to set the number of GOTs and GPs that the linker will create (default = 3). If your program needs too many GOT entries you may get a "not enough space for multiple GOTs, use `-Bcount_multigots=<number>`" error at link time.
- Bdo_jmpopt
Directs the linker to fill empty delay slots of jump and jump-and-link instructions during linkage. This is most effective with the `-Kold` or `-Kselpic` flags. Note that this option may effect debugging because the compiler may generate code that skips the first instruction of a function.
- Bkeepmdebug

Keep the `.mdebug` section in the output file. This section contains symbolic debugging information for `dbx`. By default the `.mdebug` section is removed from the output file.

-Blargedynsym

When building an dynamic executable, put all global symbols in the dynamic symbol table, `.dynsym`. By default, `ld` only puts symbols needed for dynamic linkage in the `.dynsym` section, i.e. undefined symbols and symbols referenced by shared libraries. This option may be necessary when loading shared libraries using `libdl`. Adding this option ensures that all defined global symbols in `a.out` are available to shared libraries loaded via `dlopen(3C)`.

-Bload_all

Causes the linker to unconditionally load all the object modules from an archive. Only affects those archives that appear after it on the link-line until canceled by `-Bload_symbolic`.

-Bload_symbolic

Restores the default behavior for loading modules from archives so that the modules are loaded to resolved undefined symbols, canceling the effect of `-Bload_all`. Only affects those archives that appear after it on the link-line.

-Bmultigot

Creates multiple Global Offset Tables (GOT). All entries in the GOT need to be accessible from a 16-bit offset from a common GP (Global Pointer). If the GOT grows too big you may get a "gp relocation overflows 16 bits" error at link time. This option will allow the linker to create multiple GOTs and GPs thus avoiding the error. By default, the linker creates 3 Global Offset Tables.

All the GP-specific relocations of one relocatable object file must fit together in one GOT. The compiler guarantees this for simple relocatable object files. Using `ld -r`, you can exceed this limit when you combine some relocatable object files to a new one. By linking such a relocatable object file with `-Bmultigot` you may get an error such as "A module requires <number> GOT entries. The maximum number is 16384." at link time.

-Bnocalltuning

When building an executable, does not adjust the values of global function symbols to skip the gp-register calculation at the beginning of PIC functions. Adding this option will reduce performance.

-Bnolibraryreplacement

When building an executable, directs the dynamic linker to always ignore the `LD_LIBRARY_PATH` environment variable.

-Bnoneeddlibs

The libraries linked to a shared object (`DT_NEEDED`) are not considered automatically by the linker. This was the default in previous linker versions.

-Bprintlibs

Shows the pathnames of all the libraries that `ld` actually uses to build the output file.

-Bptc_bigpage

Orders the output file so it is suitable for preloading using the `preload` utility. Applications that have a large working set size (e.g. databases) will benefit the most from this feature. To benefit from this option, the output file must be preloaded before it is executed. See the `preload(8)` man page for details.

-Brpath=pathname

pathname contains a directory list which is used to specify library search directories to the dynamic linker.

This option has the same meaning as the environment variable `LD_RUN_PATH` (see below). If `LD_RUN_PATH` is also set, the environment variable is ignored.

-Bsymbolic

In dynamic mode only:

When building a shared object, binds references to global symbols to their definitions within the object, if definitions are available. Normally, references to global symbols within shared objects are not bound

until run time, even if definitions are available, so that definitions of the same symbol in an executable or other shared objects can override the object's own definition. `ld` will issue warnings for undefined symbols unless `-z defs` overrides.

`-Bnum: num`

Controls the allocation of `SHN_COMMON` symbols to the `.sbss` section. `ld` allocates `SHN_COMMON` symbols whose size is less than or equal to `num` to the `.sbss` section. All other `SHN_COMMON` symbols are allocated in the `.bss` section. `num` defaults to 8. No space may appear on entire side of the `:` symbol.

`-BACOMMONsize: size`

In dynamic mode, determines what `SHN_COMMON` symbols are allocated in `.bss` or `.sbss`. `ld` allocates in `.sbss` or `.bss` all `SHN_COMMON` symbols whose size is less than or equal to `size`. Any remaining `SHN_COMMON` symbols is allocated at run-time by the dynamic linker. By default, `size` is effectively infinite. No space may appear on the entire side of the `:` symbol.

`-B64s`

Indicates that a 64-bit object is to be produced, default is 32-bit. No mixture of 32-bit and 64-bit object files is allowed. If this option is not set, but the first `o` file on `ld`'s command line is a 64-bit object file, `ld` will switch to 64-bit mode.

`-G` In dynamic mode only:

Produces a shared object. Undefined symbols are allowed.

`-I name`

When building an executable file, uses `name` as the pathname of the interpreter to be written into the program header. The default in static mode is no interpreter; in dynamic mode, the default is the name of the dynamic linker, `/usr/lib/libc.so.1`. Either case may be overwritten by `-I. exec` will load this interpreter when it loads the `a.out` and will pass control to the interpreter rather than to the `a.out` directly.

`-L path`

Adds `path` to the library search directories. `ld` searches for libraries first in any directories specified with `-L` options, then in the standard directories. This option is only effective if it precedes the `-l` option on the command line.

`-M mapfile`

In static mode only:

Reads `mapfile` as a text file of directives to `ld`. Because these directives change the shape of the output file created by `ld`, use of this option is strongly discouraged.

`-Q[y|n]`

Under `-Qy`, an ident string is added to the `.comment` section of the output file to identify the version of the link editor used to create the file. This will result in multiple `ld` `idents` when there have been multiple linking steps, such as when using `ld -r`. This is identical with the default action of the `cc` command. `-Qn` suppresses version.

`-V` Outputs a message giving information about the version of `ld` being used.

`-YP dirlist`

Changes the default directories used for finding libraries. `dirlist` is a comma-separated list of pathnames.

The `LD_LIBRARY_PATH` environment variable may be used to specify library search directories. In the most general case, it will contain two directory lists separated by a semicolon:

`dirlist1;dirlist2`

If `ld` is called with any number of occurrences of `-L`, as in

`ld ... -Lpath1 ... -Lpathn ...`

then the search path ordering is:

`dirlist1 path1 ... pathn dirlist2 LIBPATH`

Note that `LD_LIBRARY_PATH` is also used by the dynamic linker. If `LD_LIBRARY_PATH` is set in the environment, the dynamic linker will search the directories listed there for shared objects to be linked with the program at execution time.

If a 64-bit object is created, `LD_LIBRARY64_PATH` is evaluated instead of `LD_LIBRARY_PATH`, if it is set in the environment. If `LD_LIBRARY64_PATH` is not set, `LD_LIBRARY_PATH` is used.

The `LD_RUN_PATH` environment variable, containing a directory list, may also be used to specify library search directories to the dynamic linker. If present and not null, it is passed to the dynamic linker by `ld` via data stored in the output object file.

The `LD_RUN_PATH` environment variable or the `-Brpath=...` option is evaluated both when executable files are created and when shared objects are created. Previous linker versions only took account of this when creating executable files. If a shared object is linked to an executable file and if it contains an entry for `LD_RUN_PATH`, the entry for the executable file is extended by this entry. The sequence in which the shared object is searched by the linker is considered in this case and duplicate entries are eliminated.

This extended entry for `LD_RUN_PATH` (or the `-Brpath=...`) entry in the executable file is already used by the linker as a search path for the second-level libraries (see above) if these libraries do not yet exist under the paths specified by the user with the `-L` option or via the `LD_LIBRARY_PATH` environment variable. Only then does the linker use its standard search path. This is the same procedure as followed by the runtime loader when an executable file is started.

NOTES

Global variables, which are declared with the same name in different objects (without the suffix "extern"), are summarized as one variable. In this case, `ld` issues a warning that must be acknowledged:

```
ld: file.o: warning: size of symbol `Prognome` from file /usr/lib/libnsl.so
overridden with size of tentative definition
```

In the example above, this message indicates that a global variable "Prognome" which was declared in the file "file.c" is already contained in the "libnsl.so" library. At runtime, both have the same memory address.

If this is not intended, the above warning must be heeded and the individual variable renamed.

FILES

`libx.so`
libraries

`libx.a`
libraries

`a.out`
output file

`LIBPATH`
usually `/usr/ccs/lib:/usr/lib`

SEE ALSO

`cc(1)`, `exec(2)`, `exit(2)`, `dlopen(3C)`, `end(3C)`, `a.out(4)`, `ar(4)`, `preload(8)`.

ld(1-ucb)
BSD Compatibility Package**NAME**

ld - link editor, dynamic link editor

SYNOPSIS

/usr/ucb/ld [*options*]

DESCRIPTION

/usr/ucb/ld is the link editor for the BSD Compatibility Package. /usr/ucb/ld is identical to /usr/bin/ld [see [ld\(1\)](#)] except that BSD libraries and routines are included *before* System V libraries and routines.

OPTIONS

/usr/ucb/ld accepts the same options as /usr/bin/ld, with the following exceptions:

-L *dir*

Add *dir* to the list of directories searched for libraries by /usr/ccs/bin/ld. Directories specified with this option are searched before /usr/ucb/lib, /usr/ccs/lib and /usr/lib.

-Y LU,*dir*

Change the default directory used for finding libraries. Warning: this option may have unexpected results, and should not be used.

FILES

/usr/ucb/lib
/usr/lib
/usr/ucb/lib/libucb.a
/usr/ccs/lib
/usr/ucb
/usr/ccs/bin

SEE ALSO

[as\(1\)](#), [ld\(1\)](#), [lorder\(1\)](#), [strip\(1\)](#), [tsort\(1\)](#).

NAME

`ldd` - list dynamic dependencies

SYNOPSIS

`ldd [-d | -r ] file`

DESCRIPTION

The `ldd` command lists the pathnames of all shared objects that would be loaded as a result of executing *file*. If *file* is a valid executable but does not require any shared objects, `ldd` will succeed, producing no output.

`ldd` may also be used to check the compatibility of *file* with the shared objects it uses. It does this by printing warnings for any unresolved symbol references that would occur if *file* were executed.

OPTIONS

- `-d` Causes `ldd` to check all references to data objects.
- `-r` Causes `ldd` to check references to both data objects and functions.

EXIT STATUS

`ldd` prints its record of shared object pathnames to `stdout`. The optional list of symbol resolution problems are printed to `stderr`. If *file* is not an executable file or cannot be opened for reading, a nonzero exit status is returned.

NOTES

`ldd` doesn't list shared objects explicitly attached via `dlopen(3C)`.

`ldd` uses the same algorithm as the dynamic linker to locate shared objects.

SEE ALSO

`cc(1)`, `ld(1)`.

Chapter on "The C compilation system" in the "Guide to Tools for Programming in C".

NAME

`lex` - generate scanner

SYNOPSIS

`lex [ -ctv ] [ -n|-v ] [ -Q[o ] ] [ -- ] [ file ]`

DESCRIPTION

`lex` generates a C program from a *file* containing the "lex source text", which the user has developed for a particular problem. A `lex` source text consists of a maximum of three sections: definitions, rules and user functions. The rules specify which patterns to search for in an input text, and what actions are to be performed if a pattern is found. It is mandatory that these are specified. The definitions and user functions are optional. This produces the following structure for the `lex` source file:

Definitions

[%%

Rules]

[%%

User functions]

Multiple files are treated as a single file. If no files are specified or `-` is specified as the filename, standard input is used.

`lex` generates a file named `lex.yy.c`. When `lex.yy.c` is compiled and linked with the `lex` library, the generated program copies the standard input to the standard output, except when a pattern specified in the file is found. In this latter case, the corresponding program text is executed. The pattern for which a match has been found is held in the `yytext` variable. Matching is done in order of the search patterns in the input file.

OPTIONS

- `-c` Indicates C actions (as opposed to other programming languages such as Fortran) and is the default.
- `-t` Causes the program to be written to the standard output, not to the file `lex.yy.c`.
- `-v` Provides a two-line summary of statistics.
- `-n` Will not print out the `-v` summary.
- `-V` Print out version information on standard error.
- `-Q[o]`

Specifies whether version information is to be written to the output file `lex.yy.c`

`o` stands for a yes/no specification in the language environment set. In an English-language environment `-Qy` should be specified to write version information to `lex.yy.c`, and `-Qn` to suppress this. In a German-language environment, for example, `-Qj` or `-Qn` should be specified.

By default, no version information is output.

- `--` If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `--`.

Definitions

Substitutes can be defined in the definitions section. They are specified in the following format and must be placed at the beginning of the line:

name substitute

The character string *substitute* replaces `{ name }` if the name appears in the rules section.

The type of `yytext` can be declared in the definitions section:

%array

`yytext` is of type character array

`%pointer`

`yytext` is of type string pointer

The start states of `lex` are also defined here:

`%sname` or `%Sname`

simple start state

`%xname` or `%Xname`

exclusive start state

If the created scanner is in an exclusive state, only patterns specified for this state are taken into consideration. Simple states, on the other hand, take all patterns, without state specifications, into consideration.

Certain default table sizes are too small for some users. The table sizes for the resulting finite state machine can be set in the definitions section:

`%p n` number of positions is *n* (default 2500)

`%n n`

number of states is *n* (500)

`%e n` number of parse tree nodes is *n* (1000)

`%a n` number of transitions is *n* (2000)

`%k n` number of packed character classes is *n* (2500)

`%o n` size of output array is *n* (3000)

The use of one or more of the above automatically implies the `-v` option, unless the `-n` option is used.

Rules

The rules section of the `file` begins with the delimiting symbol `%%`. In this rules section, the user can declare local variables for `yylex()`. Every line in the rules section which begins with a space or tab character or enclosed in `%{` and `%}`, and which is positioned before the first rule, will be copied at the start of the `yylex()` function, immediately following the first opening parenthesis.

Each rule consists of a regular expression, which defines a character pattern which is to be found, and actions which are to be executed when the pattern is found. Input text which does not represent a pattern to be found will be transferred by `lex` into the output file, unchanged.

`lex` supports extended regular expressions [see [expressions\(5\)](#)] with the following exceptions and extensions:

`"xy"` *xy*, also if *x* and/or *y* are lex operators (excluding `\`)

`^x` *x* at the beginning of the line (only at the start of a pattern)

`<y>x` or `<y1,y2,...>x`

x if `lex` is in the *y* state or in one of the states *y1*, *y2* etc.

`x$` *x* at the end of the line (only at the end of a pattern)

`x/y` *x* if *y* follows

`{xx}` Substitution for *xx* from the definitions section

`\octal`

Character with the octal code *octal*

`\xhexadecimal`

Character with the hexadecimal code *hexadecimal*

`\character`

character, excluding `\character`, is one of the following escape sequences: `\\`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`

The precedence with which regular expressions are evaluated deviates from the standard sequence with regard to some points. The following table is arranged in descending precedence:

classes / character sets
 [==] [:::] [...]

quoted characters
 characters

bracketed expressions
 []

expressions in ""
 "..."

grouping
 ()

definitions
 {*name*}

repeat
 * + ?

concatenation
 xy

intervals
 {*m,n*}

or |

In the actions section for a rule, it is possible to carry out special tasks. `lex` provides the following macros for this purpose:

`input()` reads another character from the input stream
`unput()`

 pushes back a character, for a later read pass

`output()`
 writes a character to the output stream

`echo yytext` is written to the output stream

The user can redefine these macros to effect his/her own control of the input/output. However, care must be taken to ensure consistency.

Apart from saving patterns which have been found in `yytext`, there are further ways in which the text which is found can be processed using `lex` functions:

`yymore()`
 Characters which have been newly recognized will be appended to those already in `yytext` (normally, `yytext` will be overwritten by the next characters which are found).

`yyless(n)`
 Only the first *n* characters in `yytext` will be taken into consideration.

`REJECT`
 Character strings which overlap, or are partially contained in another character string will be processed. `REJECT` jumps directly to the next rule, without altering the contents of `yytext`.

LOCALE

The language of the message texts and yes/no specifications is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English and yes/no specifications must also be made in English (`y` or `n`). You must change one of these variables in order to change the language of the message texts.

In regular expressions the environment variable `LC_COLLATE` determines the meaning of character ranges, equivalence classes and character units, and the environment variable `LC_CTYPE` determines the meaning of classes. If these variables are not set, the value of `LANG` is used. If `LANG` is not defined or is empty, the

system behaves as if it were not internationalized.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

SEE ALSO

[yacc\(1\)](#), [expressions\(5\)](#).

Chapter on "lex" in the "Guide to Tools for Programming in C".

NAME

`lint` - check C programs

SYNOPSIS

`lint` [*option...*] *file...*

DESCRIPTION

`lint` detects features of C program files which are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compiler. `lint` issues error and warning messages. Among the things it checks for are unreachable code, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. `lint` checks for functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Arguments whose names end with `.c` are taken to be C source files. Arguments whose names end with `.ln` are taken to be the result of an earlier invocation of `lint` with either the `-c` or the `-o` option used. The `.ln` files are analogous to `.o` (object) files that are produced by the `cc(1)` command when given a `.c` file as input. Files with other suffixes are warned about and ignored.

`lint` takes all the `.c`, `.ln`, and `llib-lx.ln` (specified by `-l[x]`) files and processes them in their command line order. By default, `lint` appends the standard C lint library `llib-lc.ln` to the end of the list of files.

When the `-c` option is used, the `.ln` and the `llib-lx.ln` files are ignored. When the `-c` option is not used, the second pass of `lint` checks the `.ln` and the `llib-lx.ln` list of files for mutual compatibility.

OPTIONS

The following options are used to suppress certain kinds of complaints:

- `-a` Suppress complaints about assignments of `long` values to variables that are not `long`.
- `-b` Suppress complaints about `break` statements that cannot be reached.
- `-h` Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.
- `-m` Suppress complaints about external symbols that could be declared `static`.
- `-u` Suppress complaints about functions and external variables used and not defined, or defined and not used. This option is suitable for running `lint` on a subset of files of a larger program.
- `-v` Suppress complaints about unused arguments in functions.
- `-x` Do not report variables referred to by external declarations but never used.

The following options alter `lint`'s behavior:

- `-l x` Include the lint library `llib-lx.ln`

For example, you can include a lint version of the math library `llib-lm.ln` by inserting `-l[m]` on the command line. This argument does not suppress the default use of `llib-lc.ln`. These lint libraries must be in the directory provided for them. This option can be used to reference local lint libraries and is useful in the development of multi-file projects. The sequence of the specified `-l` options is significant, as the libraries are processed in this sequence.

- `-n` Do not check compatibility against the standard C lint library.
- `-p` Attempt to check portability to other dialects of C. Along with stricter checking, this option causes all nonexternal names to be truncated to eight characters and all external names to be truncated to six characters and one case.
- `-s` Produce one-line diagnostics only. `lint` occasionally buffers messages to produce a compound report.

- k Alter the behavior of `/*
LINTED [message]
*/` directives. Normally, `lint` will suppress warning messages for the code following these directives. Instead of suppressing the messages, `lint` prints an additional message containing the comment inside the directive.
- y Specifies that the file being linted will be treated as if the `/*
LINTLIBRARY
/` directive had been used. A `lint` library is normally created by using the `/
LINTLIBRARY
*/` directive.
- F Print pathnames of files `lint` normally prints the filename without the path.
- c Cause `lint` to produce a `.ln` file for every `.c` file on the command line. These `.ln` files are the product of `lint`'s first pass only, and are not checked for inter-function compatibility.
- o x Cause `lint` to create a `lint` library with the name `llib-lx.ln`. The `-c` option nullifies any use of the `-o` option. The `lint` library produced is the input that is given to `lint`'s second pass. The `-o` option simply causes this file to be saved in the named `lint` library. To produce a `llib-lx.ln` without extraneous messages, use of the `-x` option is suggested. The `-v` option is useful if the source files for the `lint` library are just external interfaces.
- v Write to standard error the product name and release.
- W *file* Write a `.ln` file to *file*, for use by `cflow(1)`.
- R *file* Write a `.ln` file to *file.lnt*, for use by `cxref(1)`.
- If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `—`.

`lint` recognizes many `cc` options, such as `-D`, `-U`, `-I`, `-L`, `-g`, `-O`, `-Xt`, `-Xa`, and `-Xc`, although `-g` and `-O` are ignored. If other options are used, a warning is issued and they are ignored.

Certain conventional comments in the C source can influence the behavior of `lint`:

```
/* ARGSUSED n */
    makes lint check only the first n arguments for usage; a missing n is taken to be 0. This option acts
    like the -v option for the next function.

/* CONSTCOND */ or /* CONSTANTCOND */ or /* CONSTANTCONDITION */
    suppresses complaints about constant operands for the next expression.

/* EMPTY */
    suppresses complaints about a null statement consequent on an if statement. This directive should
    be placed after the test expression, and before the semicolon. This directive is supplied to support
    empty if statements when a valid else statement follows. It suppresses messages on an empty else
    consequent.

/* FALLTHRU */ or /* FALLTHROUGH */
    suppresses complaints about fall through to a case or default labeled statement. This directive
    should be placed immediately preceding the label.

/* LINTLIBRARY */
    at the beginning of a file shuts off complaints about unused functions and function arguments in this
    file. This is equivalent to using the -v and -x options.

/* *LINTED [message] */
    suppresses any intra-file warning except those dealing with unused variables or functions. This
    directive should be placed on the line immediately preceding where the lint warning occurred. The -k
    option alters the way in which lint handles this directive. Instead of suppressing messages, lint will
    print an additional message, if any, contained in the comment. This directive is useful in conjunction
    with the -s option for post-lint filtering.

/* NOTREACHED */
    at appropriate points stops comments about unreachable code. This comment is typically placed just
    after calls to functions like exit(2).

/* PRINTFLIKE n */
```


makes `lint` check the first $(n-1)$ arguments as usual. The n th argument is interpreted as a `printf` format string to check the remaining arguments.

`/* PROTLIB  $n$  */`

causes `lint` to treat function declaration prototypes as function definitions if n is non-zero. This directive can only be used in conjunction with the `/*
LINTLIBRARY
*/` directive. If n is zero, function prototypes will be treated normally.

`/* SCANFLIKE  $n$  */`

makes `lint` check the first $(n-1)$ arguments as usual. The n th argument is interpreted as a `scanf` format string to check the remaining arguments.

`/* VARARGS  $n$  */`

suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first n arguments are checked; a missing n is taken to be 0. The use of the ellipsis terminator (...) in the definition is suggested in new or updated code.

`lint` produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed, if `-s` is not specified. Finally, if the `-c` option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source filename will be printed followed by a question mark.

The behavior of the `-c` and the `-o` options allows for incremental use of `lint` on a set of C source files. Generally, one invokes `lint` once for each source file with the `-c` option. Each of these invocations produces a `.ln` file that corresponds to the `.c` file, and prints all messages that are about just that source file. After all the source files have been separately run through `lint`, it is invoked once more (without the `-c` option), listing all the `.ln` files with the needed `-l` options. This will print all the inter-file inconsistencies. This scheme works well with `make`; it allows `make` to be used to `lint` only the source files that have been modified since the last time the set of source files were `linted`.

NOTES

`lint` is tailored for the PCC C compiler. However, `lint` can also be used to check programs that are to be compiled using the CES C compiler; but when it is so used, not all of the warnings which are output will be meaningful.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`. When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

FILES

LIBDIR

the directory where the lint libraries specified by the `-lx` option must exist

LIBDIR/lint[12]

first and second passes

LIBDIR/lib-lc.ln

declarations for C Library functions (binary format; source is in *LIBDIR/lib-lc*)

LIBPATH/lib-lm.ln

declarations for Math Library functions (binary format; source is in *LIBDIR/lib-lm*)

*TMPDIR/*lint**

temporary files

MPDIR

usually `/var/tmp` but can be redefined by setting the environment variable `TMPDIR` [see `tempnam` in [tempnam\(3S\)](#)].

LIBDIR

usually `/ccs/lib`

LIBPATH

usually `/usr/ccs/lib:/usr/lib`

SEE ALSO

[cc\(1\)](#), [cflow\(1\)](#), [cxref\(1\)](#), [lex\(1\)](#), [make\(1\)](#), [yacc\(1\)](#).

Chapter on "lint" in the "Guide to Tools for Programming in C".

lorder(1)**NAME**

`lorder` - find ordering relation for an object library

SYNOPSIS

`lorder file...`

DESCRIPTION

`lorder` can be used to determine the ordering relation for one or more object or library archive files [see [ar\(1\)](#)]. The standard output is a list of pairs of object file or archive member names; the first file of the pair refers to external identifiers defined in the second. The output may be processed by [tsort\(1\)](#) to find an ordering of a library suitable for onepass access by [ld\(1\)](#). Note that the link editor `ld` is capable of multiple passes over an archive in the portable archive format [see [ar\(4\)](#)] and does not require that `lorder` be used when building an archive. The usage of the `lorder` command may, however, allow for a more efficient access of the archive during the link edit process.

The following example builds a new library from existing `.o` files.

```
ar -cr library 'lorder *.o | tsort'
```

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Reliant UNIX Programmer's Guide: Internationalization – Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

NOTES

`lorder` will accept as input any object or archive file, regardless of its suffix, provided there is more than one input file. If there is but a single input file, its suffix must be `.o`.

FILES

`TMPDIR/*symref`
temporary files

`MPDIR/*symdef`
temporary files

`TMPDIR`
is usually `/var/tmp` but can be redefined by setting the environment variable `TMPDIR` [see [tmpnam\(3S\)](#)].

SEE ALSO

[ar\(1\)](#), [ld\(1\)](#), [tsort\(1\)](#), [tmpnam\(3S\)](#), [ar\(4\)](#).

NAME

lprof - display line-by-line execution count profile data

SYNOPSIS

```
lprof [-p][-s][-x][-I incdir][-r srcfile][-c cntfile][-o prog][-V]
lprof -m file1.cnt file2.cnt filen.cnt [-T]-d destfile.cnt
```

DESCRIPTION

lprof reports the execution characteristics of a program on a (source) line by line basis. This is useful as a means to determine which and how often portions of the code were executed.

lprof interprets a profile file (*prog.cnt* by default) produced by the profiled program *prog* (*a.out* by default). *prog* creates a profile file if it has been loaded with the `-ql` option of `cc`. The profile information is computed for functions in a source file if the `-ql` option was used when the source file was compiled.

A shared object may also be profiled by specifying `-ql` when the shared object is created. When a dynamically linked executable is run, one profile file is produced for each profiled shared object linked to the executable. This feature is useful in building a single report covering multiple and disparate executions of a common library. For example, if programs *prog1* and *prog2* both use library *libx.a*, running these profiled programs will produce two profile files, *prog1.cnt* and *prog2.cnt*, which cannot be combined. However, if *libx* is built as a profiled shared object, *libx.so*, and *prog1* and *prog2* are built as profiled dynamically linked executables, then running these programs with the *merge* option will produce three profile files; one of them, *libx.so.cnt*, will contain the *libx* profile information from both runs.

By default, **lprof** prints a listing of source files (the names of which are stored in the symbol table of the executable file), with each line preceded by its line number (in the source file) and the number of times the line was executed.

OPTIONS

The following options may appear singly or be combined in any order:

- `-p` Print listing, each line preceded by the line number and the number of times it was executed (default). This option can be used together with the `-s` option to print both the source listing and summary information.
- `-s` Print summary information of percentage of lines of code executed per function.
- `-x` Instead of printing the execution count numbers of each line, print each line preceded by its line number and a `[U]` if the line was not executed. If the line was executed, print only the line number.
- `-I incdir` Look for source or header files in the directory *incdir* in addition to the current directory and the standard place for `#include` files (usually `/usr/include`). The user can specify more than one directory by using multiple `-I` options.
- `-r srcfile` Instead of printing all source files, print only those files named in `-r` options (to be used with the `-p` option only). The user can specify multiple files with a single `-r` option.
- `-c cntfile` Use the file *cntfile* instead of *prog.cnt* as the input profile file.
- `-o prog` Use the name of the program *prog* instead of the name used when creating the profile file. Because the program name stored in the profile file contains the relative path, this option is necessary if the executable file or profile file has been moved.
- `-V` Print, on standard error, the version number of **lprof**.

Merging Data Files

`lprof` can also be used to merge profile files. The `-m` option must be accompanied by the `-d` option:

- `-m file1.cnt file2.cnt filen.cnt -d destfile.cnt` Merge the data files *file1.cnt* through *filen.cnt* by summing the execution counts per line, so that data from several runs can be accumulated. The result is written to *destfile.cnt*. The data files must contain profiling data for the same *prog* (see the `-T` option below).
- `-T` Time stamp override. Normally, the time stamps of the executable files being profiled are checked, and data files will not be merged if the time stamps do not match. If `-T` is specified, this check is skipped.

Controlling the run-time profiling environment

The environment variable `PROFOPTS` provides run-time control over profiling. When a profiled program (or shared object) is about to terminate, it examines the value of `PROFOPTS` to determine how the profiling data are to be handled. A terminating shared object will honor every `PROFOPTS` option except `file=filename`.

The environment variable `PROFOPTS` is a comma-separated list of options interpreted by the program being profiled. If `PROFOPTS` is not defined in the environment, then the default action is taken: The profiling data are saved in a file (with the default name, *prog.cnt*) in the current directory. If `PROFOPTS` is set to the null string, no profiling data are saved. The following are the available options:

`msg=[y|n]`

If `msg=y` is specified, a message stating that profile data are being saved is printed to `stderr`. If `msg=n` is specified, only the profiling error messages are printed. The default is `msg=y`.

`merge=[y|n]`

If `merge=y` is specified, the data files will be merged after successive runs. If `merge=n` is specified, the data files are not merged after successive runs, and the data file is overwritten after each execution. The merge will fail if the program has been recompiled, and the data file will be left in `TMPDIR`. The default is `merge=n`.

`pid=[y|n]`

If `pid=y` is specified, the name of the data file will include the process ID of the profiled program. Inclusion of the process ID allows for the creation of different data files for programs calling `fork`. If `pid=n` is specified, the default name is used. The default is `pid=n`. For `lprof` to generate its profiling report, the `-c` option must be specified with `lprof` otherwise the default will fail.

`dir=dirname`

The data file is placed in the directory *dirname* if this option is specified. Otherwise, the data file is created in the directory that is current at the end of execution.

`file=filename`

filename is used as the name of the data file in *dir* created by the profiled program if this option is specified. Otherwise, the default name is used. For `lprof` to generate its profiling report, the `-c` option must be specified with `lprof` if the file option has been used at execution time; otherwise the default will fail.

NOTES

For the `-m` option, if *destfile.cnt* exists, its previous contents are destroyed.

Optimized code cannot be profiled; if both optimization and line profiling are requested, profiling has precedence.

Programs that modify `argv[0]` are not operable under `lprof`.

Different parts of one line of a source file may be executed different numbers of times (for example, the for loop below); the count corresponds to the first part of the line.

In the following example line 5 with the for loop has three parts:

```
main()
1 [2] {
```

```

        int j;
1 [5]    for (j = 0; j > 5; j++)
5 [6]    sub(j);
1 [8] }

        sub(a)
        int a;
5 [12]   {
5 [13]   printf(a is %d\n, a);
5 [14]   }

```

The number for the line refers to the initialization (`j = 0`).

FILES

prog.cnt

Profile data

TMPDIR

Usually `/var/tmp` but can be redefined by setting the environment variable `TMPDIR`.

SEE ALSO

`cc(1)`, `fork(2)`, `tmpnam(3S)`.

m4(1) - yacc(1)

Programmer's Reference Manual

m4(1)

NAME

`m4` - macro processor

SYNOPSIS

`m4` [*option...*] [*file...*]

DESCRIPTION

The `m4` command is a macro processor intended as a front end for C, assembler, and other languages. Each of the argument files is processed in order; if there are no files, or if a filename is `-`, the standard input is read. The processed text is written on the standard output.

OPTIONS

- `-e` Operate interactively. Interrupts are ignored and the output is unbuffered.
- `-s` Enable line sync output for the C preprocessor (`#line...`)
- `-B int`
Change the size of the push-back and argument collection buffers (default is 4,096).
- `-H int`
Change the size of the symbol table hash array (default is 199). The size should be prime.
- `-S int`
Change the size of the call stack (default is 100 slots).
Macros take three slots in the call buffer, and non-macro arguments take one.
- `-T int`
Change the size of the token buffer (default is 512 bytes).

To be effective, the above flags must appear before any filenames and before any `-D` or `-U` flags. The sequence of the `-D` and `-U` options is important.

- `-D name[=val]`
Defines *name* to *val* or to null in *val*'s absence.

`-U name`

Undefines *name*.

— If the first filename begins with a dash (—), the end of the command-line options must be marked with —.

Macro calls have the form:

name(arg1,arg2, ..., argn)

There must be no spaces between the parenthesis and the macro name. If the name of a defined macro is not followed by a (, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphanumeric characters and underscore, and the first character must not be a digit.

Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses that happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

m4 makes available the following built-in macros. These macros may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

`changecom`

change left and right comment markers from the default # and newline. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes newline. With two arguments, both markers are affected. Comment markers may be up to five characters long.

`changequote`

changes the left and right hand quote symbols (two arguments) The symbols may be up to five characters long. `changequote` without arguments restores the original values (i.e., ' ').

`decr` returns the value of its argument decremented by 1.

`define`

expects two arguments, of which the first is the name of the macro and the second is its value. Each $\$n$ in the replacement text, where n is a digit, is replaced by the n th argument. Missing arguments are replaced by the null string; $\$ \#$ is replaced by the number of arguments; $\$ *$ is replaced by a list of all the arguments separated by commas; $\$ @$ is like $\$ *$, but each argument is quoted (with the current quotes).

`defn` returns the quoted definition of its argument. It is useful for renaming macros, especially built-ins.

`divert`

m4 maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The `divert` macro changes the current output stream to its (digitstring) argument. Output diverted to a stream other than 0 through 9 is discarded.

`divnum`

returns the value of the current output stream.

`dnl` reads and discards characters up to and including the next newline.

`dumpdef`

prints current names and definitions, for the named items, or

`errprint`

prints its argument on the error output file.

`eval`

evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, -, *, /, %, ** (exponentiation), bitwise &, |, ^, and ~, relationals, parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

ifdef
if the first argument is defined and is not equal to 0 (Null), the value is the second argument, otherwise the third. If there is no third argument, the value is null.

ifelse
replaces text in the output text, subject to conditions **ifelse** expects three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with the remaining arguments. Otherwise, the value is either the fourth string, or, if it is not present, null.

include
returns the contents of the file named in the argument.

incr returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digitstring as a decimal number.

index
returns the position in its first argument where the second argument begins (zero origin), or -1 if the second argument does not occur.

len returns the number of characters in its argument.

undefine
the expected argument is a macro whose definition is to be deleted.

maketemp
fills in a string of `xxxxxx` in its argument with the current process ID.

m4exit m4
causes immediate exit from `m4`. Argument 1, if given, is the exit code; the default is 0.

m4wrap
the first argument will be pushed back to the final EOF.

popdef
deletes the current definition of its argument, exposing the previous one, if any.

pushdef
like **define**, but saves any previous definition.

shift
returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

sinclude
is identical to **include**, except that `m4` says nothing if the file is inaccessible.

substr
returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

syscmd
executes the Reliant UNIX System command given in the first argument. No value is returned.

sysval
is the return code from the last call to **syscmd**.

traceon
switches on the trace for the specified macros.
If no arguments are specified, turns on tracing for all macros, including built-ins.

traceoff
turns off trace globally and for any macros specified. Macros specifically traced by **traceon** can be untraced only by specific calls to **traceoff**.

translit
transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

`undivert`

supplies all the outputs in numerical order. If arguments are specified, `undivert` will supply the requested outputs to the specified output buffers, in the specified order.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

SEE ALSO

`cc(1)`.

Chapter on "m4" in the "Guide to Tools for Programming in C".

NAME

`make` - maintain groups of files

SYNOPSIS

`make [ option ... ] [ macro-definition ... ] [ target ... ]`

DESCRIPTION

When modular programming is used, programs generally consist of a number of file. `make` offers a method of updating such programs.

`make` uses a *makefile*, in which the user can define targets, and can specify how one target depends others. When a source file is altered, `make` recreates the program by recompiling only those parts which are directly or indirectly dependent on the amended file.

`make` recreates the target when the latter is older than at least one of the files on which it depends. `make` takes into account the interdependencies of the files, and supplies the date and time of the last change to a file.

The *makefile* is normally called `makefile`, `Makefile`, `s.makefile` or `s.Makefile`. If this naming convention is adhered to, `make` can be called without specifying any arguments. `make` will search for the *makefile* in the current directory or in the `SCCS` directory, and will recreate the target file if at least one change has taken place.

OPTIONS

- `-e` Overwrites the definitions in *makefiles* by environment variables.
- `-f makefile`
makefile is assumed to be the name of a description file.
- `-i` Ignores error codes returned by commands which are part of a command sequence. `make` will continue processing even after an error. `.IGNORE` in the *makefile* has the same effect.
- `-k` Cancels the command sequence if an error occurs in one of the *makefiles* entries. Processing is continued for all other targets which do not depend on the defective target. `-k` overwrites preceding `-S` options.
- `-n` Serves as a debugging aid. All the commands will be output, including those beginning with an `@`. Only commands preceded by a plus sign `+` are executed.
- `-p` Outputs a complete list of the macro definitions and target file descriptions.
- `-q` Returns a zero status if the target file is up-to-date. A nonzero status indicates that the target file must be recreated. The target file is not changed, however, commands preceded by a plus sign `+` are executed.
- `-r` Ignores the built-in rules.
- `-s` Suppresses the setting of a new timestamp output of the commands which are executed. `.SILENT` as the target in the *makefile* has the same effect.
- `-S` Terminates if an error occurs in one of the *makefiles* entries. This is the default setting. `-S` overwrites preceding `-k` options.
- `-t` Gives a new timestamp to the targets of the *makefiles* and these are then taken to be up-to-date. Only commands preceded by a plus sign `+` are executed.
- `---` If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `---`.

Creating a makefile

The makefile invoked with the `-f` option is a carefully structured file of explicit instructions for updating and regenerating programs, and contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a `:`, then a (possibly null) list of prerequisite files or dependencies. Text following a `;` and all following lines that begin with a tab are shell commands to be executed to update the target. The first non-empty line that does not begin with a tab or `#` begins a new dependency or macro definition. Shell commands may be continued across lines with a backslash-newline (`\` newline) sequence. Everything printed by `make` (except the initial tab) is passed directly to the shell as is. Thus,

```
echo a\  
b
```

will produce

```
a b
```

exactly the same as the shell would.

Commands which are more than one line long must be a maximum of `LINE_MAX` lines long in total if the `.POSIX` command is used.

Comments are enclosed between `#` and the newline character.

The following *makefile* says that `pgm` depends on two files `a.o` and `b.o`, and that they in turn depend on their corresponding source files (`a.c` and `b.c`) and a common file `incl.h`:

```
pgm: a.o b.o  
    cc a.o b.o -o pgm  
a.o: incl.h a.c  
    cc -c a.c  
b.o: incl.h b.c  
    cc -c b.c
```

Executing a makefile

Command lines are executed in sequence from a separate shells. The shell used by `make` to execute the commands can be specified using the `SHELL` environment variable or the `SHELL` macro. The default is `/usr/bin/sh`.

The following commands can be specified in a *makefile* to control the execution of the commands:

`.POSIX`

`make` fully supports the XPG4 specification. `make` deviates from the description in some points to ensure that existing *makefiles* can be executed.

`.DEFAULT`

If a file must be created, but there are no specific commands or relevant inference rules available, the commands specified in `.DEFAULT` are used.

`.IGNORE`

Error messages from commands are ignored for all files that depend on this target. If no file is specified, it has the same effect as `-i`.

`.PRECIOUS`

Files that depend on this target are not deleted if the quit signal or the interrupt signal is received. If no file is specified, all files are retained.

`.SILENT`

Output of the executed commands is suppressed for all files independent of this target. If no file is specified, it has the same effect as `-s`.

The execution of individual commands can be controlled by preceding them with `@`, `-` or `+`:

`@` Command output is suppressed

`-` Errors are ignored

`+` The command is always executed, even if one of the `-n`, `-q` or `-t` options is set.

A line is output when it is executed if the `-s` option is present, the `.SILENT` entry is valid for the file, or the

initial string contains a @. The `-n` option outputs the command without executing it, unless the command line is preceded by + (in which case the line is always executed). The `-t` option brings the altered date of a file up-to-date without executing the command (unless a command is preceded by +).

`make` is normally terminated by commands (or when the `-S` option is set) which return a non-zero status. The end status is ignored if the `-i` option is present, the `.IGNORE` entry is valid for the file, or the initial string of the command contains -. If the `-k` option is present, work is terminated on the current entry but continued in other branches which do not depend on this entry.

The target file is deleted by canceling and quitting, unless the `.PRECIOUS` entry is valid for it.

Environment

The environment is read by `make`. All variables are assumed to be macro definitions and are processed as such – except the `SHELL` variable. The value of the `SHELL` environment variable will not be used as a macro and will not be modified by defining the `SHELL` macro in a *makefile* or on the command line. The environment variables are processed before any *makefile* and immediately after the internal rules; thus, macro assignments in a *makefile* override environment variables. The `-e` option causes the environment to override the macro assignments in a *makefile*. Suffixes and their associated rules in the *makefile* will override any identical suffixes in the built-in rules.

The `MAKEFLAGS` environment variable can contain macros and all input options which are valid for the command line with the exception of `-f` and `-p`. `make` evaluates the variable in front of the command line. The `MAKEFLAGS` macro (and the variable if it is not defined) is automatically assigned the current options and macros when `make` is called, and is passed on when the commands are called. In this way, the `MAKEFLAGS` macro always contains the current definitions. This feature proves very useful for "super-makes". In fact, as noted above, when the `-n` option is used, the command `$(MAKE)` is executed anyway; hence, one can perform a `make -n` recursively on a whole software system to see what would have been executed. This result is possible because the `-n` is put in `MAKEFLAGS` and passed to further invocations of `$(MAKE)`. This usage is one way of debugging all of the *makefiles* for a software project without actually doing anything.

The `PROJECTDIR` environment variable specifies in which directory the SCCS files should be sought if they are not in the current directory. SCCS files are searched for in a `SCCS` subdirectory. If a relative pathname is specified, the `HOME` directory is searched first for the `src` or `source` subdirectories. If these directories do not exist, the path is set relative to the current directory.

Include files

If the string `include` appears as the first seven letters of a line in a *makefile*, and is followed by a blank or a tab, the rest of the line is assumed to be a filename and will be read by the current invocation, after substituting for any macros.

Macro definitions

Entries of the form `string1 = string2` are macro definitions. `string2` is defined as all characters up to a comment character or an unescaped newline. Subsequent appearances of `$(string1[:subst1=[subst2]])` are replaced by `string2`. The parentheses are optional if a single-character macro name is used and there is no substitute sequence. The optional `:subst1=subst2` is a substitute sequence. If it is specified, all nonoverlapping occurrences of `subst1` in the named macro are replaced by `subst2`. Strings for this type of substitution are delimited by blanks, tabs, newline characters, and beginnings of lines. An example of the use of the substitute sequence is shown in the "Libraries" section below.

The `SHELL` macro is treated specially. It is provided and set to the pathname of the shell command language interpreter `sh(1)`. The `SHELL` environment variable will not affect the value of the `SHELL` macro. If `SHELL` is defined in the *makefile* or is specified on the command line, it will replace the original value of the `SHELL` macro, but will not affect the `SHELL` environment variable.

Internal macros

There are five internally maintained macros that are useful for writing rules for building targets.

`$(*)` stands for the filename part of the current dependent with the suffix deleted. It is evaluated only for

inference rules.

- `$@` stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.
- `$<` is only evaluated for inference rules or the `.DEFAULT` rule. It is the module that is outdated with respect to the target, i.e. the "manufactured" dependent filename. Thus, in the `.c.o` rule, the `$<` macro would evaluate to the `.c` file.

An example for making optimized `.o` files from `.c` files is:

```
.c.o:
cc -c -O $*.c
```

or:

```
.c.o:
cc -c -O $<
```

- `$%?` is evaluated when explicit rules from the *makefile* are evaluated. It is the list of prerequisites that are outdated with respect to the target.
- `$%` is only evaluated when the target is an archive library member of the form `lib(file.o)`. In this case, `$@` evaluates to `lib` and `$%` evaluates to the library member, `file.o`.

The macros can have alternative forms. When an upper case `D` or `F` is appended, the meaning is changed to "directory part" for `D` and "file part" for `F`. Thus, `$(@D)` refers to the directory part of the string `$@`. If there is no directory part, `./` is generated. If the `$?` macro contains more than one filename, the `$(?D)` and `$(?F)` macros (or `${?D}` and `${?F}`) expand into a list of directory parts and file parts.

Inference rules

Certain names, such as those ending with `.o`, have inferable prerequisites such as `.c`, `.s`, etc. If no update commands for such a file appear in *makefile*, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, *make* has inference rules that allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. The current default inference rules are:

```
.c      .c~      .f      .f~      .s      .s~      .sh      .sh~      .C      .C~
.c.a    .c.o     .c~.a   .c~.c   .c~.o   .f.a     .f.o     .f~.a   .f~.f   .f~.o
.h~.h   .l.c     .l.o     .l~.c   .l~.l   .l~.o   .s.a     .s.o     .s~.a   .s~.o
.s~.s   .sh~.sh   .y.c     .y.o     .y~.c   .y~.o   .y~.y   .C.a     .C.o     .C~.a
.C~.C   .C~.o    .L.C     .L.o     .L~.C   .L~.L   .L~.o   .Y.C     .Y.o     .Y~.C
.Y~.o   .Y~.y
```

The user may add rules to this list by simply putting them in the *makefile*.

The inference of prerequisites can be controlled. The rule to create a file with suffix `.o` from a file with suffix `.c` is specified as an entry with `.c.o:` as the target and no dependents. Shell commands associated with the target define the rule for making a `.o` file from a `.c` file. Any target that has no slashes in it and starts with a dot is identified as a rule and not a true target.

The internal rules for *make* are contained in the source file `rules.c` for the *make* program. These rules can be locally modified. To print out the rules compiled into the *make* on any machine in a form suitable for recompilation, the following command is used:

```
make -pf - 2>/dev/null </dev/null
```

A tilde in the above rules refers to an SCCS file [see [sccsfile\(4\)](#)]. Thus, the rule `.c~.o` would transform an SCCS C source file into an object file (`.o`). Because the `s.` of the SCCS files is a prefix, it is incompatible with the *make* suffix point of view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (for example, `.c:`) is the definition of how to build `x` from `x.c`. In effect, the other suffix is null. This feature is useful for building targets from only one source file (for example, shell procedures and simple C programs).

Additional suffixes are given as the dependency list for `.SUFFIXES`. Order is significant; the first possible

name for which both a file and a rule exist is inferred as a prerequisite. The default list is:

```
.SUFFIXES .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~ .f .f~
.C .C~ .Y .Y~ .L .L~
```

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; `.SUFFIXES:` with no dependencies clears the list of suffixes.

The example given in "Creating a makefile" can be formulated shorter using this:

```
pgm: a.o b.o
    cc a.o b.o -o pgm
a.o b.o: incl.h
```

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, `CFLAGS`, `LFLAGS`, and `YFLAGS` are used for compiler options to `cc(1)`, `lex(1)`, and `yacc(1)`, respectively.

With the `.SCCS_GET` command, the default commands for accessing SCCS files which are not in the current directory, can be changed. The inference rule is:

```
.SCCS_GET: sccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@
```

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus, `lib(file.o)` and `$(LIB) (file.o)` both refer to an archive library that contains `file.o`. This example assumes the `LIB` macro has been previously defined. The expression `$(LIB) (file1.o file2.o)` is not legal. Rules pertaining to archive libraries have the form `.XX.a` where the `XX` is the suffix from which the archive member is to be made. An unfortunate by-product of the current implementation requires the `XX` to be different from the suffix of the archive member. Thus, one cannot have `lib(file.o)` depend upon `file.o` explicitly.

The most common use of the archive interface follows. Here, we assume the source files are all C type source:

```
lib: lib(file1.o) lib(file2.o) lib(file3.o)
    @echo lib is now up-to-date
.c.a:
    $(CC) -c $(CFLAGS) $<
    $(AR) $(ARFLAGS) $@ $*.o
    rm -f $*.o
```

In fact, the `.c.a` rule listed above is built into `make` and is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib: lib(file1.o) lib(file2.o) lib(file3.o)
    $(CC) -c $(CFLAGS) $(?:.o=.c)
    $(AR) $(ARFLAGS) lib $?
    rm $?
    @echo lib is now up-to-date
.c.a:;
```

Here the substitution mode of the macro expansions is used. The `$?` list is defined to be the set of object filenames (inside `lib`) whose C source files are outdated. The substitution mode translates the `.o` to `.c`. Unfortunately, one cannot as yet transform to `.c`; however, this transformation may become possible in the future. Also note the disabling of the `.c.a:` rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

DIAGNOSTICS

With the defective output of `make` an error message appears which contains a contraction in the form (`bucode_no`). The `help code_no` command can be used to assess this code. You then receive a detailed explanation of the error. Example:

```

make: *** Error code 1 (bu21) (ignored) □
$ help bu21
bu21:
"*** Error code '...' " OR "*** Termination code ' ' "
This error is reported but does not occur in make(1). It
is associated with the last command executed. See the UNIX manual for
additional information on the error codes for that command. If
"ignored" has been printed on the line immediately following the above
error message, this error is ignored by make(1). Errors
will be ignored if the -i option was set on execution, or if the dummy
target .IGNORE appears in the makefile, or if the particular command was
prefaced with a minus sign (-).

```

Note: The help command is not internationalized.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization – Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

NOTES

Some commands return non-zero status inappropriately; use `-i` or the `-` command line prefix to overcome the difficulty.

Filenames with the characters `=` `:` `@` will not work. Commands that are directly executed by the shell, notably [cd\(1\)](#), are ineffectual across newlines in make. The syntax `lib(file1.o□file2.o□file3.o)` is illegal. You cannot build `lib(file.o)` from `file.o`.

FILES

`[Mm]akefile` and `s.[Mm]akefile`
`/usr/bin/sh`

SEE ALSO

[cc\(1\)](#), [cd\(1\)](#), [lex\(1\)](#), [sh\(1\)](#), [yacc\(1\)](#), [printf\(3S\)](#), [sccsfile\(4\)](#).

Chapter on "make" in the "Guide to Tools for Programming in C".

NAME

mcs - manipulate the comment section of an object file.

SYNOPSIS

mcs [**-a** *string*] [**-c**] [**-d**] [**-n** *name*] [**-p**] [**-v**] *file*...

DESCRIPTION

The **mcs** command (manipulate comment section) is used to manipulate a section, by default the `.comment` section, in an ELF object file. It is used to add to, delete, print, and compress the contents of a section in an ELF object file, and only print the contents of a section in a COFF object file. **mcs** must be given one or more of the options described below. It applies each of the options in order to each file.

OPTIONS

-a *string*

Append *string* to the comment section of the ELF object files.

If *string* contains embedded blanks, it must be enclosed in quotation marks.

-c Compress the contents of the comment section of the ELF object files.

All duplicate entries are removed. The ordering of the remaining entries is not disturbed.

-d Delete the contents of the comment section from the ELF object files.

The section header for the comment section is also removed.

-n *Name*

Specify the *name* of the comment section to access.

By default, **mcs** deals with the section named `.comment`.

-p Print the contents of the comment section on the standard output.

Each section printed is tagged by the name of the file from which it was extracted, using the format *filename* [*member_name*] : for archive files; and *filename* : for other files.

-v Print the version number of **mcs** on the standard error output.

If the input file is an archive [see [ar\(4\)](#)], the archive is treated as a set of individual files. For example, if the **-a** option is specified, the string is appended to the comment section of each ELF object file in the archive; if the archive member is not an ELF object file, then it is left unchanged.

If **mcs** is executed on an archive file the archive symbol table will be removed, unless only the **-p** option has been specified. The archive symbol table must be restored by executing the **ar** command with the **s** option before the archive can be linked by the **ld** command. **mcs** will produce appropriate warning messages when this situation arises.

EXAMPLES

```
mcs -p file           # Print file's comment section
mcs -a string file    # Append string to file's comment section
```

NOTES

mcs cannot add to, delete or compress the contents of a section that is contained within a segment.

FILES

`TMPDIR/mcs*`

temporary files

`TMPDIR`

usually `/var/tmp` but can be redefined by setting the environment variable `TMPDIR` [see `tempnam` in


```
tmpnam(3S)].
```

SEE ALSO

```
ar(1), cc(1), ld(1), tmpnam(3S), a.out(4), ar(4).
```

NAME

nm - print name list of an object file

SYNOPSIS

nm [*option ...*] *file ...*

DESCRIPTION

The nm command (name list) displays the symbol table of each ELF or COFF object file, specified by *file*. The file may be a relocatable or absolute ELF or COFF object file; or it may be an archive of relocatable or absolute ELF or COFF object files. For each symbol, the following information will be printed:

Lib/Obj

Library or object name (only for -A)

Name The name of the symbol.

Type Symbol type:

A	global absolute symbol
a	local absolute symbol
B	global bss symbol
b	local bss symbol
D	global data symbol
d	local data symbol
T	global text symbol
t	local text symbol
U	undefined symbol

Value

The value of the symbol is one of the following: a section offset for defined symbols in a relocatable file; alignment constraints for symbols whose section index is SHN_COMMON; a virtual address in executable and dynamic library files.

Size The size in bytes of the associated object.

Index

The index of the symbol, which appears in brackets.

Bind The symbol's binding attributes. LOCAL symbols have a scope limited to the object file containing their definition; GLOBAL symbols are visible to all object files being combined; and WEAK symbols are essentially global symbols with a lower precedence than GLOBAL.

Other

A field reserved for future use, currently containing 0.

Shndx

Except for three special values, this is the section header table index in relation to which the symbol is defined. The following special values exist: ABS indicates the symbol's value will not change through relocation; COMMON indicates an unallocated block and the value provides alignment constraints; and UNDEF indicates an undefined symbol.

The output of nm may be controlled using the following options:

-A	Print the full pathname or library name of an object on each line
-e	See NOTES section.
-f	See NOTES section.

- g Print external symbols information only.
- h Do not display the output heading data.
- l Distinguishes between `WEAK` and `GLOBAL` symbols by appending a `*` to the key letter for `WEAK` symbols.
- n Sort external symbols by name before they are printed.
- o Print the value and size of a symbol in octal instead of decimal (same as `-t o`).
- p Produce terse output that can be parsed easily. Each symbol name is preceded by its value (blanks if undefined) and one of the letters `U` (undefined), `N` (symbol has no type), `D` (data object symbol), `T` (text symbol), `S` (section symbol), or `F` (file symbol). If the symbol's binding attribute is `LOCAL`, the key letter is lower case; if the symbol's binding attribute is `WEAK`, the key letter is upper case; if the `-l` modifier is specified, the upper case key letter is followed by a `*`; if the symbol's binding attribute is `GLOBAL`, the key letter is upper case.
- P Print output in a portable format. If the `-A` option is also specified, a line in the following format is displayed for each symbol:

```
<Library name/object name> <Symbol name> <Type> <Value> <Size>
```

 If `-A` is not specified, the library or object name is written to a separate line. If the `-t` is not specified, `-t x` is accepted implicitly.
- r Prepend the name of the object file or archive to each output line.
- t *format*
 Specifies the output format of a symbol value and size:
 - d The value and size are written in decimal format (default)
 - o The value and size are written in octal format (same as `-o`)
 - x The value and size are written in hexadecimal format (same as `-x`)
- T See **NOTES** section.
- u Print undefined symbols only.
- v Sort external symbols by value before they are printed (instead of alphabetically).
- V Prints the version of the `nm` command executing on the standard error output. (Should not be used in portable scripts.)
- x Print the value and size of a symbol in hexadecimal instead of decimal (same as `-t x`).
- If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `—`.

Options may be used in any order, either singly or in combination. When conflicting options are specified (such as `nm -v -n`) the first is taken and the second ignored with a warning message to the user.

NOTES

The following options are obsolete because of changes to the object file format.

- e Print only external and static symbols.
 Automatic symbols no longer appear in the symbol table. They do appear in the debugging information produced by `cc -g`, which may be examined using `dump(1)`.
- f Produces full output. Redundant symbols (such as `.text`, `.data`, etc.) which existed previously do not exist and producing full output will be identical to the default output.
- T By default, `nm` prints the entire name of the symbols listed. Since symbol names have been moved to the last column, the problem of overflow is removed and it is no longer necessary to truncate the symbol name.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization – Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

SEE ALSO

`cc(1)`, `dump(1)`, `ld(1)`, `a.out(4)`, `ar(4)`.

NAME

`prof` - display profile data

SYNOPSIS

`prof [-t | -c | -a | -n] [-o | -x] [-g | -l] [-z] [-h] [-s] [-mdata] [-V] [prog]`

DESCRIPTION

The `prof` command interprets a profile file produced by the `monitor` function. The symbol table in the object file *prog* (`a.out` by default) is read and correlated with a profile file, `mon.out` by default. For each external text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

OPTIONS

The mutually exclusive options `-t`, `-c`, `-a`, and `-n` determine the type of sorting of the output lines:

- `-t` Sort by decreasing percentage of total time (default).
- `-c` Sort by decreasing number of calls.
- `-a` Sort by increasing symbol address.
- `-n` Sort lexically by symbol name.

The mutually exclusive options `-o` and `-x` specify the printing of the address of each symbol monitored:

- `-o` Print each symbol address (in octal) along with the symbol name.
- `-x` Print each symbol address (in hexadecimal) along with the symbol name.

The mutually exclusive options `-g` and `-l` control the type of symbols to be reported. The `-l` option must be used with care; it applies the time spent in a `static` function to the preceding (in memory) global function, instead of giving the `static` function a separate entry in the report. If all `static` functions are properly located (see example below), this feature can be very useful. If not, the resulting report may be misleading.

Assume that `A` and `B` are global functions and only `A` calls `static` function `S`. If `S` is located immediately after `A` in the source code (that is, if `S` is properly located), then, with the `-l` option, the amount of time spent in `A` can easily be determined, including the time spent in `S`. If, however, both `A` and `B` call `S`, then, if the `-l` option is used, the report will be misleading; the time spent during `B`'s call to `S` will be attributed to `A`, making it appear as if more time had been spent in `A` than really had. In this case, function `S` cannot be properly located.

- `-g` Include `static` (non-global) functions.
- `-l` Do not include `static` (non-global) functions (default).

The following options may be used in any combination:

- `-z` Include all symbols in the profile range, even if associated with zero number of calls and zero time.
- `-h` Suppress the heading normally printed on the report. This is useful if the report is to be processed further.
- `-s` Print a summary of several of the monitoring parameters and statistics on the standard error output.
- `-m mdata`
Use file `mdata` instead of `mon.out` as the input profile file.
- `-V` Print `prof` version information on the standard error output.

A program creates a profile file if it has been link edited with the `-p` option of `cc`. This option to the `cc` command arranges for calls to `monitor` at the beginning and end of execution. It is the call to `monitor` at the end of execution that causes the system to write a profile file. The number of calls to a function is tallied if

the `-p` option was used when the file containing the function was compiled.

The name of the file created by a profiled program is controlled by the environmental variable `PROFDIR`. If `PROFDIR` is not set, `mon.out` is produced in the directory current when the program terminates. If `PROFDIR=string, string/pid.progname` is produced, where *progname* consists of `argv[0]` with any path prefix removed, and *pid* is the process ID of the program. If `PROFDIR` is set, but null, no profiling output are produced.

A single function may be split into subfunctions for profiling by means of the `MARK` macro.

NOTES

The times reported in successive identical runs may show variances because of varying cache-hit ratios that result from sharing the cache with other processes. Even if a program seems to be the only one using the machine, hidden background or asynchronous processes may blur the data. In rare cases, the clock ticks initiating recording of the program counter may "beat" with loops in a program, grossly distorting measurements. Call counts are always recorded precisely, however.

Only programs that call `exit` or return from `main` are guaranteed to produce a profile file, unless a final call to the function `monitor` is explicitly coded.

The times for `static` functions are attributed to the preceding external text symbol if the `-g` option is not used. However, the call counts for the preceding function are still correct; that is, the `static` function call counts are not added to the call counts of the external function.

If more than one of the options `-t`, `-c`, `-a`, and `-n` is specified, the last option specified is used and the user is warned.

Care must be taken when creating profiles for dynamically linked programs. Although this is fundamentally possible, the following restrictions must be borne in mind.

In order to be able to receive any output from `prof`, the library `libc` must also be linked statically to the program, from which a profile is to be created, when creating a dynamic link. This can be done with "`cc□... -Bstatic -lc -Bdynamic...`" without statically linking the entire program in the process.

It is not normally possible to create profiles using `prof` for two shared objects at present. Thus, when a profiled, dynamically linked program is executed, only the "main" portion of the image is sampled. This means that all time spent outside of the "main" object, that is, time spent in a shared object, will not be included in the profile summary; the total time reported for the program may be less than the total time used by the program.

Because the time spent in a shared object cannot be accounted for, the use of shared objects should be minimized whenever a program is profiled with `prof`. If possible, the program should be linked statically before being profiled.

Consider an extreme case. A profiled program dynamically linked with the shared library (e.g. `libsocket`) spends 100 units of time in some `libsocket` routine, say, `bind`. Suppose `bind` is called only from routine `B` and `B` consumes only 1 unit of time. Suppose further that routine `A` consumes 10 units of time, more than any other routine in the "main" profiled portion of the image. In this case, `prof` will conclude that most of the time is being spent in `A` and almost no time is being spent in `B`. From this it will be almost impossible to tell that the greatest improvement could be made by looking at routine `B` and not routine `A`. The value of the profiler in this case is severely degraded. When you want to profile a program, it is in your interest to use archives instead of dynamically linked objects.

FILES

`mon.out`
default profile file

`a.out`
default name list (object) file

SEE ALSO

`cc(1)`, `exit(2)`, `profil(2)`, `monitor(3C)`.

NAME

`prs` - print an SCCS file

SYNOPSIS

```
prs [-a ] [-r[SID ] ] [-d dataspec ] [ — ] file ...
prs [-e | -l ] -c date-time [ -ddataspec ] [ — ] file ...
prs [-e | -l ] -r[ SID ] [ -ddataspec ] [ — ] file ...
```

DESCRIPTION

`prs` (print SCCS-file) prints, on the standard output, parts or all of an SCCS file [see [sccsfile\(4\)](#)] in a user-supplied format. If a directory is named, `prs` prints the files in that directory, except the non-SCCS files (last component of the pathname does not begin with `s.`) and unreadable files. If a name of `-` is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory. `prs` silently ignores non-SCCS files and unreadable files.

OPTIONS

Arguments to `prs`, which may appear in any order, consist of keyletters and filenames.

All the specified keyletters apply independently to each named file:

`-d dataspec`

Specifies the output format. *dataspec* is a string consisting of SCCS file data keywords (see the **Data keywords** section) interspersed with optional user-supplied text.

`-r[SID]`

Specifies the SCCS identification (SID) string of a delta for which information is desired. The default is the top delta.

`-e` Requests information for all deltas created earlier than and including the delta designated via the `-r` keyletter or the date given by the `-c` option.

`-l` Requests information for all deltas created later than and including the delta designated via the `-r` keyletter or the date given by the `-c` option.

`-c date-time`

The cutoff date-time in the form:

`YY[MM[DD[HH[MM[SS]]]]]`

Units omitted from the date-time default to their maximum possible values; for example, `-c 7502` is equivalent to `-c 750228235959`. Any number of non-numeric characters may separate the fields of the cutoff date; for example, `"-c 77/2/2 9:22:25"`.

Note: If the two-digit year notation *YY* is in the range 70 to 99, the 20th century is assumed; if it is in the range 00 to 37, the 21st century is assumed. All other entries are rejected.

`-a` Requests information for both removed, i.e., delta type = R, [see [rmDEL\(1\)](#)] and existing, i.e., delta type = D, deltas. If the `-a` keyletter is not specified, information for existing deltas only is provided.

`—` If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `—`.

Data keywords

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file [see [sccsfile\(4\)](#)] have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by `prs` consists of the user-supplied text, and appropriate values extracted from the

SCCS file, substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either "Simple" (S), in which keyword substitution is direct, or "Multi-line" (M), in which keyword substitution is followed by a RETURN.

User-supplied text is any text other than recognized data keywords. A tab is specified by \t and carriage return/newline is specified by \n. The default data keywords are:

Keyword	Data item	File Section	Value	Format
:Dt:	Delta information	Delta table	See below *	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by delta	"	nnnn	S
:Ld:	Lines deleted by delta	"	nnnn	S
:Lu:	Lines unchanged by delta	"	nnnn	S
:DT:	Delta type	"	D or R	S
:I:	SCCS ID String (SID)	"	:R:..L:..B:..S :	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year delta created	"	nn	S
:Dm:	Month delta created	"	nn	S
:Dd:	Day delta created	"	nn	S
:T:	Time delta created	"	:Th:..:Tm:..:Ts:	S
:Th:	Hour delta created	"	nn	S
:Tm:	Minutes delta created	"	nn	S
:Ts:	Seconds delta created	"	nn	S
:P:	Programmer who created delta	"	logname	S
:DP:	Predecessor delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	deltas included (seq#)	"	:DS: :DS:...	S

Keyword	Data item	File Section	Value	Format
:Dx:	Deltas excluded (seq#)	"	:DS: :DS:...	S
:Dg:	Deltas ignored (seq#)	"	:DS: :DS:...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S

:MF:	MR validation flag	"	yes or no	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	yes or no	S
:KV:	Keyword validation string	"	text	S
:BF:	Branch flag	"	yes or no	S
:J:	Joint edit flag	"	yes or no	S
:LK:	Locked release	"	:R:...	S
:Q:	User-defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	yes or no	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of what(1) string	N/A	:Z::M:\t:I:	S
:A:	A form of what(1) string	N/A	:Z: :Y::M: :I::Z:	S
:Z:	what(1) string delimiter	N/A	@(#)	S
:F:	SCCS filename	N/A	text	S
:PN:	SCCS file pathname	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

EXAMPLES

The command

```
prs -d "Users and/or user IDs for :F: are:\n:UN:"s.file
```

may produce on the standard output:

```
Users and/or user IDs for s.file are:□
```

```
xyz□
```

```
131□
```

```
abc
```

The command

```
prs -d "Newest delta for pgm :M:: :I: Created :D: By :P:"-r s.file
```

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas
```

The default case:

```
prs s.file
```

produces the following on the standard output, for each "D" type entry in the delta table:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000□
```

```
MRs:□
```

```
b178-12345□
```

```
b179-54321□  
COMMENTS:□  
this is the comment line for s.file initial delta  
The only keyletter allowed to be used for this is the -a keyletter.
```

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`. When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts. Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization – Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

FILES

```
/var/tmp/pr????
```

SEE ALSO

```
admin(1), delta(1), get(1), help(1), what(1), sccsfile(4).
```

NAME

`prt` - display the delta and commentary history of an SCCSfile

SYNOPSIS

`/usr/ucb/prt [-abdefistu] [-y[SID]] [-c[cutoff]] [-r[rev-cutoff]] filename ...`

DESCRIPTION

Note: the `prt` command is an older version of `prs(1)` that in most circumstances is more convenient to use, but is less flexible than `prs`.

`prt` prints part or all of an SCCS file in a useful format. If a directory is named, `prt` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with `s.`) and unreadable files are silently ignored. If a name of `"-"` is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

OPTIONS

- a Print those types of deltas normally not printed by the `d` keyletter. These are types `R` (removed). This keyletter is effective only if the `d` keyletter is also specified (or assumed).
- b Print the body of the SCCS file.
- d This keyletter normally prints delta table entries of the `D` type.
- e This keyletter implies the `d`, `i`, `u`, `f`, and `t` keyletters and is provided for convenience.
- f Print the flags of the named file.
- i Print the serial numbers of those deltas included, excluded, and ignored. This keyletter is effective only if the `d` keyletter is also specified (or assumed).

The following format is used to print those portions of the SCCS file as specified by the above keyletters. The printing of each delta table entry is preceded by a NEWLINE.

- Type of delta (`D` or `R`).
- SPACE.
- SCCS identification string (SID).
- TAB.
- Date and time of creation (in the form `YY/MM/DD HH:MM:SS`).
- SPACE.
- Creator.
- TAB.
- Serial number.
- SPACE.
- Predecessor delta's serial number.
- TAB.
- Statistics (in the form `inserted/deleted/unchanged`).
- NEWLINE.
- "Included:TAB", followed by SID's of deltas included, followed by NEWLINE (only if there were any such deltas and if `i` keyletter was supplied).
- "Excluded:TAB", followed by SID's of deltas excluded, followed by NEWLINE (see note above).

- "Ignored:TAB", followed by SID's of deltas ignored, followed by NEWLINE (see note above).
 - "MRs:TAB", followed by MR numbers related to the delta, followed by NEWLINE (only if any MR numbers were supplied).
 - Lines of comments (delta commentary), followed by newline (if any were supplied).
- s Print only the first line of the delta table entries; that is, only up to the statistics. This keyletter is effective only if the `d` keyletter is also specified (or assumed).
- t Print the descriptive text contained in the file.
- u Print the login-names and/or numerical group IDs of those users allowed to make deltas.
- y[SID]
Print the delta table entries to stop when the delta just printed has the specified SID. If no delta in the table has the specified SID, the entire table is printed. If no SID is specified, the first delta in the delta table is printed. This keyletter will print the entire delta table entry for each delta as a single line (the NEWLINE in the normal multi-line format of the `d` keyletter are replaced by SPACE characters) preceded by the name of the SCCS file being processed, followed by a `:`, followed by a TAB. This keyletter is effective only if the `d` keyletter is also specified (or assumed).
- c[cutoff]
Stop printing the delta table entries if the delta about to be printed is older than the specified cutoff date-time (see `get(1)` for the format of date-time). If no date-time is supplied, the epoch 0000 GMT Jan. 1, 1970 is used. As with the `y` keyletter, this keyletter will cause the entire delta table entry to be printed as a single line and to be preceded by the name of the SCCS file being processed, followed by a `:`, followed by a TAB. This keyletter is effective only if the `d` keyletter is also specified (or assumed).
- r[rev-cutoff]
Begin printing the delta table entries when the delta about to be printed is older than or equal to the specified cutoff date-time (see `get(1)` for the format of date-time). If no date-time is supplied, the epoch 0000 GMT Jan. 1, 1970 is used. (In this case, nothing will be printed.) As with the `y` keyletter, this keyletter will cause the entire delta table entry to be printed as a single line and to be preceded by the name of the SCCS file being processed, followed by a `:`, followed by a TAB. This keyletter is effective only if the `d` keyletter is also specified (or assumed).
- If any keyletter but `y`, `c`, or `r` is supplied, the name of the file being processed (preceded by one NEWLINE and followed by two NEWLINE characters) is printed before its contents.
- If none of the `u`, `f`, `t`, or `b` keyletters is supplied, the `d` keyletter is assumed.
- Note: the `s` and `i` keyletters, and the `c` and `r` keyletters are mutually exclusive; therefore, they may not be specified together on the same `prt` command.
- The form of the delta table as produced by the `y`, `c`, and `r` keyletters makes it easy to sort multiple delta tables in chronological order.
- When both the `y` and `c` or the `y` and `r` keyletters are supplied, `prt` will stop printing when the first of the two conditions is met.

SEE ALSO

`admin(1)`, `delta(1)`, `get(1)`, `prs(1)`, `sccs(1)`, `what(1)`, `sccsfile(4)`.

regcmp(1)**NAME**

regcmp - regular expression compile

SYNOPSIS

regcmp [-] *file* ...

DESCRIPTION

The `regcmp` (regular expression compile) command compiles the regular expressions in *file* and places the output in *file.i*. If the `-` option is used, the output is placed in *file.c*. The format of entries in *file* is a name (C variable) followed by one or more blanks followed by one or more regular expressions enclosed in double quotes. The output of `regcmp` is C source code. Compiled regular expressions are represented as `extern char` vectors. *file.i* files may thus be `#included` in C programs, or *file.c* files may be compiled and later loaded. In the C program that uses the `regcmp` output, `regex(abc, line)` applies the regular expression named `abc` to `line`. Diagnostics are self-explanatory.

EXAMPLES

```
name      "([A-Za-z][A-Za-z0-9\\_]*)$0"
telno     "\\({0,1}\\)[2-9][01][1-9])$0\\){0,1} *"
          "([2-9][0-9]{2})$1[-]{0,1}"
          "[0-9]{4})$2"
```

The three arguments to `telno` shown above must all be entered on one line.

In the C program that uses the `regcmp` output,

```
regex(telno, line, area, exch, rest)
```

applies the regular expression named `telno` to `line`.

SEE ALSO

[regcomp\(3C\)](#), [regex\(3C\)](#), [regcmp\(3G\)](#), [regexpr\(3G\)](#), [expressions\(5\)](#), [regex\(5\)](#), [regexp\(5\)](#).

NAME

rmDEL - remove a delta from an SCCS file

SYNOPSIS

rmDEL -r *SID* [—] *file*...

DESCRIPTION

rmDEL (remove delta) removes the delta specified by the *SID* (SCCS identification string) from each named SCCS file, *file*. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the *SID* specified must not be that of a version being edited for the purpose of making a delta; that is, if a p-file exists for the named SCCS file [see [get\(1\)](#)], the delta specified must not appear in any entry of the p-file.

The -r option specifies that the delta with the number *SID* is to be removed.

It is necessary to specify — if the first filename begins with -.

If a directory is named, rmDEL behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed.

The rules governing the removal of a delta are as follows: if you make a delta and have appropriate file permissions, you can remove it; if you own the file and directory in which a new delta file resides, you can remove the delta.

LOCALE

The language of the message texts is governed by the environment variable LC_ALL, LC_MESSAGES or LANG.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization. Refer also to [environ\(5\)](#) for information on setting the user environment.

FILES

x-file

See [delta\(1\)](#)

z-file

See [delta\(1\)](#)

SEE ALSO

[delta\(1\)](#), [get\(1\)](#), [prs\(1\)](#), [sccsfile\(4\)](#).

NAME

`sact` - print SCCS file editing activity

SYNOPSIS

`sact file...`

DESCRIPTION

`sact` (SCCS file activity) informs the user of any impending deltas to a named SCCS file. This situation occurs when `get(1)` with the `-e` option has been previously executed without a subsequent execution of `delta`. If a directory is named on the command line, `sact` behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of `-` is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

The output for each named file consists of five fields separated by spaces.

SID specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta.

new SID
specifies the SID for the new delta to be created.

login
contains the logname of the user who will make the delta, i.e. who has executed a `get` for editing.

date contains the date that `get -e` was executed.

time contains the time that `get -e` was executed.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to `environ(5)` for information on setting the user environment.

SEE ALSO

`delta(1)`, `get(1)`, `help(1)`, `unget(1)`.

sccs(1)**NAME**

sccs - shell command for the Source Code Control System(SCCS)

SYNOPSIS

sccs [**-r**] [**-d** *rootpath*] [**-p** *finalpath*] *command* [*SCCS-option* ...] [*filename* ...]

DESCRIPTION

The **sccs** command is a front end to the utility program of the Source Code Control System (SCCS).

sccs normally prefixes each *filename* or the last component of each *filename* with the string "SCCS/s.", because you normally save your SCCS database files in a directory called **SCCS**, and each database file starts with an "s." prefix.

The **PROJECTDIR** environment variable specifies in which directory the SCCS should be sought, if they are not in the current directory. SCCS files are searched for in a **SCCS** subdirectory. If relative pathnames are specified, the **HOME** directory is searched first for the **src** or **source** subdirectories. If these directories do not exist, the path is set relative to the current directory.

sccs program options must appear before the *command* argument. Options to be passed to the actual SCCS command (utility program) must appear after the *command* argument. These options are specific to the *command* argument being used.

sccs can be installed as a "set user ID" program to provide additional protection. This means that direct access to SCCS files is no longer possible for every user, but only those with the suitable **sccs** call. To stop administrator commands [such as **admin(1)**] being called by anyone, these commands always run as the real user.

OPTIONS

-r **sccs** calls *command* as the real user ID and not as the effective user ID, under which **sccs** itself runs.

-d *rootpath*

Defines the prefix portion of the pathname for the SCCS database files. The current directory is the default. The full pathname is prefixed by *rootpath* (see **EXAMPLES**).

The pathnames take precedence over the directories defined by the **PROJECTDIR** environment variable.

-p *finalpath*

Defines the name of the directory that contains the SCCS files. The default is **SCCS**. *finalpath* is appended before the last component of a pathname specified using *command* (see **EXAMPLES**).

ADDITIONAL SCCS COMMANDS

The following pseudo-commands are available in addition to the usual SCCS commands. These are:

check

Checks files which are currently being edited. Like **info**, but returns an exit code rather than a list. If nothing is being edited, there is no output, and if a file was being edited, a non-zero status is returned. **check** can thus be included in an install entry in a makefile to ensure that a SCCS file contains all the necessary components before a version is installed.

clean

Deletes all files from the current directory which can be recreated from SCCS files. **clean** establishes which files are currently being edited and does not delete these. When "**clean -b**" is used it does *not* check if other branches is being edited at that moment. If you save the branches in the same directory it is dangerous to use **-b**.

create

Creates new SCCS files. All options of the **admin** command can be specified. After successful creation

of the SCCS file, the original file is renamed by inserting a comma in front of the base name.

delget

Performs a `delta` on the specified files and then call new versions using `get`. The keywords are expanded so that the new versions cannot be edited.

deledit

Same as `delget`, but creates new versions that can be edited (`get -e`). `deledit` is useful for making a checkpoint of your current editing phase.

diffs

Compares (in `diff`-like format) the current version of the program you have for editing in SCCS format. `diffs` accepts the same arguments as `diff`, except that the `-c` option must be specified as `-C` instead, because the `-c` option is taken as an option of `get`, indicating which version is to be compared with the current version.

edit Gets a file for editing (same as `get -e`).

fix Removes the specified delta, but leaves a copy of the delta with the changes in it. `fix` must follow a `-rSID`. `fix` is useful for recovering small compiler errors. Since `fix` does not leave audit trails, it should be used carefully.

info Displays a list of all files currently being edited. If the `-b` option is given, branches (that is SIDs with two or less components) are ignored. If the `-u user` option is given, only files edited by the specified user are listed. `-U` is the same as `-u current_user`.

print

Prints detailed information about the specified files, in the same way as `sccs -prs`.

tell Displays a list of files being edited on the standard output. Filenames are separated by NEWLINE characters. The same applies for the `-b` and `-u` or `-U` characters as for `info` and `check`.

unedit

Undoes the last `edit` or `get -e` and returns a file to its previous condition. If you `unedit` a file being edited, all changes made since the beginning of the editing session are lost.

EXAMPLES

The command

```
sccs -d /usr/include get sys/inode.h
```

is converted to:

```
get /usr/include/sys/SCCS/s.inode.h
```

This should allow the creation of aliases such as:

```
alias syssccs = "sccs -d /usr/src"
```

These are used as:

```
syssccs get cmd/who.c
```

The command

```
sccs -p private get usr/include/stdio.h
```

is converted to:

```
get usr/include/private/s.stdio.h
```

To convert a file called `myprogram.c` into SCCS format for the first time, assuming that no SCCS directory exists:

```
$ mkdir SCCS
$ sccs create myprogram.c
$ myprogram.c:
1.1
14 lines
```

After you have verified that everything is in order, remove the version of the file that starts with a comma:

```
$ rm ,myprogram.c
```

To get a copy of `myprogram.c` for editing, edit the file and place it back in the SCCS database:

```
$ sccs edit myprogram.c
1.1
new delta 1.2
14 lines
$ vi myprogram.c
```

Your editing session begins.

```
$ sccs delget myprogram.c
comments? Added abusive responses for compatibility
1.2
7 inserted
7 deleted
7 unchanged
1.2
14 lines
```

To get a file from another directory:

```
sccs -p /usr/src/sccs/ get cc.c
```

Alternatively you can also enter the following:

```
sccs get /usr/src/sccs/cc.c
```

To create a delta of a large number of files in the current directory:

```
sccs delta *.c
```

To get a list of files being edited that are not branches:

```
sccs info -b
```

To create a delta version of all files being edited:

```
sccs delta `sccs tell -u`
```

In a makefile, to get source files from an SCCS file if it does not exist:

```
SRCS = <list_of_source_files>
$(SRCS):
    sccs get $(REL) $@
```

LOCALE

The language of the message texts is governed by the environment variables `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the *Programmer's Guide: Internationalization – Localization*. Refer also to [environ\(5\)](#) for information on setting the user environment.

FILES

```
/usr/sccs/*
```

SEE ALSO

[admin\(1\)](#), [cdc\(1\)](#), [comb\(1\)](#), [delta\(1\)](#), [get\(1\)](#), [help\(1\)](#), [make\(1\)](#), [prs\(1\)](#), [rmDEL\(1\)](#), [sact\(1\)](#), [sccsdiff\(1\)](#), [unget\(1\)](#), [val\(1\)](#), [what\(1\)](#), [sccsfile\(4\)](#).

NAME

sccs - front end for the Source Code Control System (SCCS)

SYNOPSIS

`/usr/ucb/sccs [-r] [-dprefixpath] [-pfinalpath] command [SCCS-flags ...] [filename ...]`

DESCRIPTION

The **sccs** command is a front end to the utility programs of the Source Code Control System (SCCS).

sccs normally prefixes each *filename*, or the last component of each *filename*, with the string "SCCS/s.", because you normally keep your SCCS database files in a directory called **SCCS**, and each database file starts with an "s." prefix. If the environment variable **PROJECTDIR** is set, and is an absolute pathname (that is, begins with a slash) **sccs** will search for SCCS files in the directory given by that variable. If it is a relative pathname (that is, does not begin with a slash), it is treated as the name of a user, and **sccs** will search in that user's home directory for a directory named **src** or **source**. If that directory is found, **sccs** will search for SCCS files in the directory given by that variable.

sccs program options must appear before the *command* argument. Flags to be passed to the actual SCCS command (utility program) must appear after the *command* argument. These flags are specific to the *command* being used.

sccs also includes the capability to run "set user ID" to another user to provide additional protection. Certain commands [such as **admin(1)**] cannot be run "set user ID" by all users, since this would allow anyone to change the authorizations. Such commands are always run as the real user.

OPTIONS

-r Run **sccs** as the real user rather than as whatever effective user **sccs** is "set user ID" to.

-dprefixpath

Define the prefix portion of the pathname for the SCCS database files. The default prefix portion of the pathname is the current directory. *prefixpath* is prefixed to the entire pathname. See **EXAMPLES**.

This flag overrides any directory specified by the **PROJECTDIR** environment variable.

-pfinalpath

Define the name of a lower directory in which the SCCS files will be found; **SCCS** is the default. *finalpath* is appended before the final component of the pathname. See **EXAMPLES**.

USAGE**Additional sccs Commands**

Several pseudo-commands are available in addition to the usual SCCS commands. These are:

create

create is used when creating new **s.** files. For example, given a C source language file called **obscure.c**, the **create** command would perform the following actions: (1) create the **s.** file called **s.obscure.c** in the SCCS directory; (2) rename the original source file to **,obscure.c** (3) do an **sccs get** on **obscure.c**. Compared to the SCCS **admin** command, **create** does more of the startup work for you and should be used in preference to **admin**.

enter

enter is just like **create**, except that it does not do the final **sccs get**. It is usually used if an **sccs edit** is to be performed immediately after the **enter**.

edit Get a file for editing.

delget

Perform a `delta` on the named files and then `get` new versions. The new versions have ID keywords expanded, and so cannot be edited.

`deledit`

Same as `delget`, but produces new versions suitable for editing. `deledit` is useful for making a checkpoint of your current editing phase.

fix Remove the named delta, but leaves you with a copy of the delta with the changes that were in it. `fix` must be followed by an `-r` flag. `fix` is useful for fixing small compiler bugs, etc. Since `fix` does not leave audit trails, use it carefully.

`clean`

Remove everything from the current directory that can be recreated from SCCS files. `clean` checks for and does not remove any files being edited. If "`clean -b`" is used, branches are *not* checked to see if they are currently being edited. Note: `-b` is dangerous if you are keeping the branches in the same directory.

`unedit`

Undo the last `edit` or `get -e` and return a file to its previous condition. If you `unedit` a file being edited, all changes made since the beginning of the editing session are lost.

info Display a list of all files being edited. If the `-b` flag is given, branches (that is, SID's with two or fewer components) are ignored. If the `-u` flag is given (with an optional argument), only files being edited by you (or the named user) are listed.

`check`

Check for files currently being edited, like `info`, but returns an exit code rather than a listing: nothing is printed if nothing is being edited, and a non-zero exit status is returned if anything is being edited. `check` may thus be included in an install entry in a makefile, to ensure that everything is included in an SCCS file before a version is installed.

tell Display a list of files being edited on the standard output. Filenames are separated by NEWLINE characters. Take the `-b` and `-u` flags like `info` and `check`.

`diffs`

Compare (in `diff`-like format) the current version of the program you have out for editing and the versions in SCCS format. `diffs` accepts the same arguments as `diff`, except that the `-c` flag must be specified as `-C` instead, because the `-c` flag is taken as a flag to `get` indicating which version is to be compared with the current version.

`print`

Print verbose information about the named files. `print` does an "`sccs□prs□-e`" followed by an "`sccs□get□-p□-m`" on each file.

EXAMPLES

The command:

```
sccs -d/usr/include get sys/inode.h
```

converts to:

```
get /usr/include/sys/SCCS/s.inode.h
```

The intent here is to create aliases such as:

```
alias syssccs sccs -d/usr/src
```

which will be used as:

```
syssccs get cmd/who.c
```

The command:

```
sccs -pprivate get usr/include/stdio.h
```

converts to:

```
get usr/include/private/s.stdio.h
```

To put a file called `myprogram.c` into SCCS format for the first time, assuming also that there is no SCCS

directory already existing:

```
$ mkdir SCCS
$ sccs create myprogram.c
$ myprogram.c:
1.1
14 lines
```

After you have verified that everything is all right you remove the version of the file that starts with a comma:

```
$ rm ,myprogram.c
```

To get a copy of `myprogram.c` for editing, edit that file, then place it back in the SCCS database:

```
$ sccs edit myprogram.c
1.1
new delta 1.2
14 lines
$ vi myprogram.c
```

Your editing session begins.

```
$ sccs delget myprogram.c
comments? Added abusive responses for compatibility
1.2
7 inserted
7 deleted
7 unchanged
1.2
14 lines
```

To get a file from another directory:

```
sccs -p/usr/src/sccs/ get cc.c
```

or:

```
sccs get /usr/src/sccs/cc.c
```

To make a delta of a large number of files in the current directory:

```
sccs delta *.c
```

To get a list of files being edited that are not on branches:

```
sccs info -b
```

To delta everything that you are editing:

```
$ sccs delta 'sccs tell -u'
```

In a makefile, to get source files from an SCCS file if it does not already exist:

```
SRCS = <list_of_source_files>
$(SRCS):
    sccs get $(REL) $@
```

Regular sccs Commands

The regular SCCS commands are described very briefly below. It is unlikely that you ever need to use these commands because the user interface is so complicated, and the `sccs` front end command does 99.9% of the interesting tasks for you.

admin

Create new SCCS files and changes parameters of existing SCCS files. You can use "`sccs create`" to create new SCCS files, or use "`sccs admin`" to do other things.

cdc Change the commentary material in an SCCS delta.

comb Combine SCCS deltas and reconstructs the SCCS files.

delta

Permanently introduces changes that were made to a file previously retrieved using "`sccs get`". You

can use `"sccs delget"` as the more useful version of this command since `"sccs delget"` does all of the useful work and more.

- `get` Extract a file from the SCCS database, either for compilation, or for editing when the `-e` option is used. Use `"sccs get"` if you really need it, but `"sccs delget"` will normally have done this job for you. Use `sccs edit` instead of `get` with the `-e` option.
- `help` Supposed to help you interpret SCCS error messages.
- `prs` Display information about what is happening in an SCCS file.
- `rmDEL`
Remove a delta from an SCCS file.
- `sccsdiff`
Compare two versions of an SCCS file and generates the differences between the two versions.
- `val` Determine if a given SCCS file meets specified criteria. If you use the `sccs` command, you should not need to use `val`, because its user interface is unbelievable.
- `what` Display SCCS identification information.

NOTE

The `help` command usually just parrots SCCS error messages and is generally not considered very helpful.

FILES

`/usr/sccs/*`

SEE ALSO

`admin(1)`, `cdc(1)`, `comb(1)`, `delta(1)`, `get(1)`, `help(1)`, `prs(1)`, `rmDEL(1)`, `sact(1)`, `sccsdiff(1)`, `unget(1)`, `val(1)`, `what(1)`, `sccsfile(4)`.

sccsdiff(1)**NAME**

`sccsdiff` - compare two versions of an SCCS file

SYNOPSIS

`sccsdiff -r SID1 -rSID2 [ -p ] [ -sn ] file...`

DESCRIPTION

`sccsdiff` compares two versions of an SCCS file and generates the differences between the two versions. Any number of SCCS files may be specified, but arguments apply to all files.

OPTIONS

`-rSID1 -rSID2`

SID1 and *SID2* specify the deltas of an SCCS file that are to be compared. Versions are passed to `bdiff` in the order given.

`-p` pipes output for each file through `pr`.

`-sn` *n* is the file segment size that `bdiff` will pass to `diff`. This option is useful when `diff` cannot be successfully executed due to a high system load.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Reliant UNIX Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

FILES

`/var/tmp/get?????`
temporary files

SEE ALSO

[bdiff\(1\)](#), [diff\(1\)](#), [get\(1\)](#), [help\(1\)](#), [pr\(1\)](#).

NAME

`size` - print size of an object file

SYNOPSIS

`size [-F -f -n -o -V -x] file...`

DESCRIPTION

`size` outputs the size of an object file, i.e. an object module or an executable program, in bytes.

The number of bytes occupied by the text, data and bss segments is output on the standard output.

`size` processes ELF and COFF object files entered on the command line. If an archive file is input to the `size` command, the information for each object file in the archive is displayed.

When calculating segment information, the `size` command prints out the file size for the writable and non-writable segments and the memory size of the writable segments minus their file size.

If it cannot calculate segment information, `size` recalculates the section information. When calculating section information, it prints out the size of sections that are allocatable, non-writable, and not NOBITS, the size of the sections that are allocatable, writable, and not NOBITS, and the size of the writable sections of type NOBITS. (NOBITS sections do not actually take up space in the *file*.)

If `size` cannot recalculate either segment or section information, it prints an error message and stops processing the file.

OPTIONS

- `-F` Prints out the size and permission flags of each loadable segment and the sum of the loadable segment sizes. If there is no segment data, `size` prints an error message and stops processing the file.
- `-f` Prints out the size of each allocatable section, the name of the section, and the total of the section sizes. If there is no section data, `size` prints out an error message and stops processing the file.
- `-n` Prints out non-loadable segment or non-allocatable section sizes. If segment data exists, `size` prints out the memory size of each loadable segment or file size of each non-loadable segment, the permission flags, and the total size of the segments. If there is no segment data, `size` prints out, for each allocatable and non-allocatable section, the memory size, the section name, and the total size of the sections. If there is no segment or section data, `size` prints an error message and stops processing.
- `-o` Prints numbers in octal, not decimal.
- `-V` Prints the version information for the `size` command on the standard error output.
- `-x` Prints numbers in hexadecimal; not decimal.

EXAMPLES

The examples below are typical `size` output.

```
size file          2724 + 88 + 0 = 2812
size -f file       26(.txt) + 5(.init) + 5(.fini) = 36
size -F file       2724(r-x) + 88(rwx) = 2812
```

NOTES

Since the size of bss sections is not known until link-edit time, the `size` command will not give the true total size of pre-linked objects.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`. When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Reliant UNIX Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

SEE ALSO

`cc(1)`, `ld(1)`, `a.out(4)`, `ar(4)`.

NAME

`strip` - strip symbol table

SYNOPSIS

`strip [-blrVxm] [—] file...`

DESCRIPTION

The `strip` command strips the symbol table, debugging information, and line number information from ELF object files; COFF object files can no longer be processed using `strip`. Once this stripping process has been done, no symbolic debugging access will be available for that file; therefore, this command is normally run only on production modules that have been debugged and tested.

If `strip` is executed on a common archive file [see [ar\(1\)](#)] in addition to processing the members, `strip` will remove the archive symbol table. The archive symbol table must be restored by executing the [ar\(1\)](#) command with the `-s` option before the archive can be linked by the [ld\(1\)](#) command. `strip` will produce appropriate warning messages when this situation arises.

OPTIONS

The amount of information stripped from the ELF object file can be controlled by using any of the following options:

- `-b` Same effect as the default behavior. This option is obsolete and will be removed in the next release.
- `-l` Strips line number information only; does not strip the symbol table or debugging information.
- `-r` Same effect as the default behavior. This option is obsolete and will be removed in the next release.
- `-V` Print, on standard error, the version number of `strip`.
- `-x` Does not strip the symbol table; debugging and line number information may be stripped.
- `-m` Do not strip the ".mdebug section", as the symbol table and information on debugging and line numbers may also be stripped. This option can be combined with the other options in order to contain the symbol table as well, for example (`-xm`).
- `—` If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `—`.

`strip` is used to reduce the file storage overhead taken by the object file.

NOTES

The symbol table section will not be removed if it is contained within a segment, or the file is either a relocatable or dynamic shared object.

The line number and debugging sections will not be removed if they are contained within a segment, or their associated relocation section is contained within a segment.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`. When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

FILES

TMPDIR/strip*
temporary files

TMPDIR
is usually `/var/tmp`, but can be redefined by setting the environment variable `TMPDIR` [see `tempnam()` in [tmpnam\(3S\)](#)].

SEE ALSO

[ar\(1\)](#), [cc\(1\)](#), [ld\(1\)](#), [tmpnam\(3S\)](#), [a.out\(4\)](#), [ar\(4\)](#).

NAME

`tsort` - topological sort

SYNOPSIS

`tsort [ file ]`

DESCRIPTION

The `tsort` command (topological sort) produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

The `tsort` sorting sequence is not changed by the `LC_COLLATE` environment variable.

DIAGNOSTICS

Odd data:
there is an odd number of fields in the input file.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

SEE ALSO

[lorder\(1\)](#).

NAME

`unget` - undo a previous `get` of an SCCS file

SYNOPSIS

`unget` [`-r` *SID*] [`-s`] [`-n`] [`—`] *file*

DESCRIPTION

`unget` (undo `get`) undoes the effect of a `get -e` done prior to creating the intended new delta. `unget` must be invoked by the same user as the one who executed the `get -e`. If a directory is named, `unget` behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of `-` is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

OPTIONS

Keyletter arguments apply independently to each named file.

`-r` *SID*

Uniquely identifies which delta is no longer intended. (This would have been specified by `get` as the "new delta"). The use of this keyletter is necessary only if two or more outstanding `gets` for editing on the same SCCS file were done by the same person (login name). An error message results if the specified *SID* is ambiguous, or if it is necessary and omitted on the command line.

`-s` Suppresses the printout, on the standard output, of the intended delta's *SID*.

`-n` Retains the gotten file, which would normally be removed from the current directory.

`—` If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `—`.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

FILES

p-file

see [delta\(1\)](#)

q-file

see [delta\(1\)](#)

z-file

see [delta\(1\)](#)

SEE ALSO

[delta\(1\)](#), [get\(1\)](#), [help\(1\)](#), [sact\(1\)](#).

NAME

`unifdef` - resolve and remove `ifdef`'ed lines from C program source

SYNOPSIS

```
/usr/ucb/unifdef [-clt] [-Dname] [-Uname] [-iDname] [-iUname] ... [filename]
```

DESCRIPTION

`unifdef` removes `ifdef`'ed lines from a file while otherwise leaving the file alone. It is smart enough to deal with the nested `ifdefs`, comments, single and double quotes of C syntax, but it does not do any including or interpretation of macros. Neither does it strip out comments, though it recognizes and ignores them. You specify which symbols you want defined with `-D` options, and which you want undefined with `-U` options. Lines within those `ifdefs` will be copied to the output, or removed, as appropriate. Any `ifdef`, `ifndef`, `else`, and `endif` lines associated with *filename* will also be removed.

`ifdefs` involving symbols you do not specify are untouched and copied out along with their associated `ifdef`, `else`, and `endif` lines.

If an `ifdefX` occurs nested inside another `ifdefX`, then the inside `ifdef` is treated as if it were an unrecognized symbol. If the same symbol appears in more than one argument, only the first occurrence is significant.

`unifdef` copies its output to the standard output and will take its input from the standard input if no *filename* argument is given.

OPTIONS

- `-c` Complement the normal operation. Lines that would have been removed or blanked are retained, and vice versa.
- `-l` Replace lines removed with blank lines.
- `-t` Plain text option. `unifdef` refrains from attempting to recognize comments and single and double quotes.
- `-iDname`
Ignore, but print out, lines associated with the defined symbol *name*. If you use `ifdefs` to delimit non-C lines, such as comments or code which is under construction, then you must tell `unifdef` which symbols are used for that purpose so that it will not try to parse for quotes and comments within them.
- `-iUname`
Ignore, but print out, lines associated with the undefined symbol *name*.

DIAGNOSTICS

Premature EOF

Inappropriate `else` or `endif`.

Exit status is 0 if output is exact copy of input, 1 if not, 2 if `unifdef` encounters problems.

SEE ALSO

`cc(1)`, `diff(1)`.

NAME

`val` - validate an SCCS file

SYNOPSIS

```
val -
val [-s] [-r SID] [-m name] [-ytype] [—] file...
```

DESCRIPTION

`val` (validate) determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to `val` may appear in any order.

For `val`, the meaning of the argument `-` is different from its use with other SCCS commands. It allows `val` to read from the standard input instead of from the command line.

`val` generates diagnostic messages on the standard output, and also returns an 8-bit code on exit.

OPTIONS

The effects of each keyletter argument apply independently to each file named on the command line.

`-s` Suppresses the diagnostic message normally generated on the standard output for any error.

`-r SID`

Checks whether the *SID* is ambiguous (e.g., `-r□1` is ambiguous because it physically does not exist but implies 1.1, 1.2, etc., which may exist) or invalid (e.g., `-r□1.0` or `-r□1.1.0` are invalid because neither can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

`-m name`

The argument value *name* is compared with the SCCS `%M%` keyword in *file*.

`-y type`

The argument value *type* is compared with the SCCS `%Y%` keyword in *file*.

— If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `—`.

The 8-bit code returned by `val` is a disjunction of the possible errors; it can be interpreted as a bit string in which the bits which are set are interpreted as follows, taken from left to right:

bit 0

=missing *file* name

bit 1

=unknown or duplicate option

bit 2

=corrupted SCCS file

bit 3

=cannot open file or file not SCCS

bit 4

=*SID* is invalid or ambiguous

bit 5

=*SID* does not exist

bit 6

=`%Y%`, `-y` mismatch

bit 7

=`%M%`, `-m` mismatch

`val` can process between two and 49 files on a given command line and in turn can process multiple command lines when reading the standard input. In these cases an aggregate code is returned: a logical OR of the codes generated for each command line and file processed.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

SEE ALSO

[admin\(1\)](#), [delta\(1\)](#), [get\(1\)](#), [help\(1\)](#), [prs\(1\)](#).

NAME

vc - version control

SYNOPSIS

vc [*option...*] [*keyword=value...*]

DESCRIPTION

vc (version control) is obsolete and will be removed in the next release.

The vc command copies lines from the standard input to the standard output under control of its arguments and of control statements encountered in the standard input. In the process of performing the copy operation, user-declared *keywords* may be replaced by their string *value* when they appear in plain text and/or control statements.

The copying of lines from the standard input to the standard output is conditional, based on tests of keyword values specified in control statements or as vc command arguments.

A control statement is a single line beginning with a control character. The exception to this is -t. The default control character is a colon :, but this can be modified using the -c keyletter. Input lines beginning with a backslash \ followed by a control character are not control lines and are copied to the standard output with the backslash removed. Lines beginning with a backslash followed by a non-control character are copied in their entirety.

A keyword is composed of 9 or less alphanumeric; the first must be alphabetic. A value is any ASCII string that can be created with ed; a numeric value is an unsigned string of digits. Keyword values may not contain blanks or tabs.

Keywords are replaced by values whenever a keyword enclosed between control characters is encountered in a control statement. The -a keyletter forces replacement of keywords throughout the text. An uninterpreted control character may be included in a value by preceding it with \. If a literal \ is desired, then it too must be preceded by \.

OPTIONS

- a Replaces keywords enclosed between control characters with their assigned value in all text lines and not just in vc statements.
- t All characters from the beginning of a line up to and including the first tab character are ignored for the purpose of detecting a control statement. If a control statement is found, all characters up to and including the tab are discarded.
- cchar Specifies a control character to be used in place of the : default.
- s Silences warning messages that are normally printed on the diagnostic output; but not error messages.

vc recognizes the following version control statements:

:dcl *keyword*[, ..., *keyword*]

Declare keywords. All keywords must be declared.

:asg *keyword=value*

Assign values to keywords. An asg statement overrides the assignment for the corresponding keyword on the vc command line and all previous asg statements for that keyword. Keywords that are declared but are not assigned values have null values.

:if *condition*

...

:end

Skip lines of the standard input. If the condition is true, all lines between the if statement and the

matching `end` statement are copied to the standard output. If the condition is false, all intervening lines are discarded, including control statements. Note that intervening `if` statements and matching `end` statements are recognized solely for the purpose of maintaining the proper `if-end` matching.

The syntax of a condition is:

`<cond> ::= [ 'not' ] <or>`

`<or> ::= <and> | <and> ' | ' <or>`

`<and> ::= <expr> | <expr> '&' <and>`

`<expr> ::= ' ( ' <or> ' ) ' | <value>`

`<op> <value>`

`<op> ::= '=' | '!=' | '<' | '>'`

`<value> ::= <arbitrary ASCII string> | <numeric string>`

The available operators and their meanings are:

`=` equal

`!=` not equal

`&` and

`|` or

`>` greater than

`<` less than

`()` used for logical groupings

`not` may only occur immediately after an `if` and, when present, inverts the value of the entire condition

The comparisons `>` and `<` operate only on unsigned integer values (e.g., `:012>012` is false). All other operators take strings as arguments (e.g., `:012!=012` is true).

The precedence of the operators is as follows:

`= != > <` all of equal precedence

`&`

`|`

Parentheses may be used to alter the order of precedence.

Values must be separated from operators or parentheses by at least one blank or tab.

`::text`

Replace keywords on lines that are copied to the standard output. The two leading control characters are removed, and keywords surrounded by control characters in text are replaced by their value before the line is copied to the output file. This action is independent of the `-a` keyletter.

`:on` Turn on keyword replacement on all lines.

`:off` Turn off keyword replacement on all lines.

`:ctl char`

Change the control character to *char*.

`:msg message`

Prints the specified *message* on the error output.

`:err message`

Prints the specified *message* on the error output:

ERROR: err statement on line ... (915)

`vc` halts execution, and returns an exit status of 1.

SEE ALSO

`ed(1)`, `help(1)`.

NAME

`what` - identify files

SYNOPSIS

`what [-s] [—] file...`

DESCRIPTION

`what` searches the given *files* for all occurrences of the pattern that the `get(1)` command substitutes for `%Z%` (this is `@(#)` at this printing) and prints out the characters which follow it, up to the first `"`, `>`, newline, `\` or null character.

For example, if the C program contains the following statement line in the file `f.c`:

```
#ident "@(#) identification information" ;
```

and if `f.c` is compiled to yield `f.o` and `a.out`, then the command

```
what f.c f.o a.out
```

prints

```
f.c:  Identification information
```

```
f.o:  Identification information
```

```
a.out:
```

```
    Identification information
```

`what` is intended to be used in conjunction with the SCCS `get` command, which automatically inserts identifying information, but it can also be used where the information is inserted by another means (e.g. with an editor).

OPTIONS

`-s` Terminate the search as soon as the first occurrence of the pattern is found.

`—` If the first filename begins with a dash (`-`), the end of the command-line options must be marked with `—`.

EXIT STATUS

Exit status is 0 if any matches are found, otherwise 1.

LOCALE

The language of the message texts is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English. You must change one of these variables in order to change the language of the message texts.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to `environ(5)` for information on setting the user environment.

SEE ALSO

`get(1)`, `help(1)`, `mcs(1)`.

NAME

xstr - extract strings from C programs to implement shared strings

SYNOPSIS

```
/usr/ucb/xstr [-v] [-c] [-l name] [file]
```

DESCRIPTION

xstr maintains a `strings` file into which strings in component parts of a large program are hashed. The strings are replaced with references to this common area. This serves to implement shared constant strings.

The

```
xstr -c file
```

command will extract the strings from the C source in *file* and replace string references with expressions of the form `(&xstr[number])`. A declaration of `xstr` is prepended to the file. The resulting C source code is placed in the `x.c` file for compiling. The strings from the *file* are placed in the `strings` file if they are not there already. Repeated strings and strings which are suffixes of existing strings do not initiate changes in the database.

Once all components of a large program have been compiled, an `xs.c` file can be created with

```
xstr
```

which defines the common `xstr` array. This `xs.c` file should then be compiled and loaded with the rest of the program.

xstr does not touch the `strings` file unless new items are added. `make` can therefore avoid remaking `xs.o` unless absolutely necessary.

xstr can also be used on a single file. The

```
xstr file
```

command creates `x.c` and `xs.c` files as before, without using or touching any `strings` file in the same directory.

It may be useful to run **xstr** after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings. **xstr** reads from its standard input when the `"-"` argument is given instead of *file*. An appropriate command sequence for running **xstr** after the C preprocessor is:

```
cc -E name.c | xstr -c -
cc -c x.c
mv x.o name.o
```

OPTIONS

- `-v` (verbose) The changes to the `strings` file are logged to standard error.
- `-c` Handles individual source files of a large program (similar to `cc -cfile`).
- `-l name`
The array bears the name *name* in `x.c` and `xs.c` instead of `xstr`.

NOTES

If a string is a suffix of another string in the `strings` file, but the shorter string is seen first by **xstr**, both strings will be placed in the `strings` file, whereas just the longer one would suffice.

FILES

```
strings
Strings file
```

`x.c` Massaged C source

`xs.c` Csource for definition of array `xstr`

SEE ALSO

`cc(1-ucb)` .

NAME

`yacc` - generate parser

SYNOPSIS

`yacc` [*option...*] *file*

DESCRIPTION

The `yacc` command (yet another compiler-compiler) converts a context-free grammar into a set of tables for a simple automaton that executes an `LALR(1)` parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

All `yacc` output files have the default prefix `y`, all external names have the prefix `yy`. These prefixes can be changed using options (e.g. if a number of `yacc` processes are to run simultaneously in the same directory).

The output file, `y.tab.c`, must be compiled by the C compiler to produce a program `yyparse`. This program must be loaded with the lexical analyzer program, `yylex`, as well as `main` and `yyerror`, an error handling routine. These routines must be supplied by the user; the `lex(1)` command is useful for creating lexical analyzers usable by `yacc`.

OPTIONS

- `-v` Prepares the file `y.output`, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.
- `-d` Generates the file `y.tab.h` with the `#define` statements that associate the `yacc`-assigned "token codes" with the user-declared "token names". This association allows source files other than `y.tab.c` to access the token codes.
- `-l` Specifies that the code produced in `y.tab.c` will not contain any `#line` constructs. This option should only be used after the grammar and the associated actions are fully debugged, i.e. are free of errors.
- `-Q[o]` Specifies whether version information is to be written to `y.tab.c`
`o` stands for a yes/no specification in the language environment set. In an English-language environment `-Qy` should be specified to write version information to `y.tab.c`, and `-Qn` to suppress this. In a German-language environment, for example, `-Qj` or `-Qn` should be specified.
 By default, no version information is output.
- `-i` Permits the direct creation of localized programs.
 If `yacc` is called with the option `-i`, calls of the `gettext()` function are written to the `yacc`-generated part of the file created. `gettext()` has access to the localized message catalogs [see `gettext(3C)`].
- `-t` Compiles runtime debugging code by default. Runtime debugging code is always generated in `y.tab.c` under conditional compilation control. By default, this code is not included when `y.tab.c` is compiled. Whether or not the `-t` option is used, the runtime debugging code is under the control of `YYDEBUG`, a preprocessor symbol. If `YYDEBUG` has a nonzero value, then the debugging code is included. If its value is zero, then the code will not be included. The size and execution time of a program produced without the runtime debugging code will be smaller or shorter.
- `-b file prefix`
 Uses the prefix *file prefix* instead of `y` for the output files (e.g. `fileprefix.tab.c`).
- `-p symbol prefix`
 Uses the prefix *symbol prefix* instead of `yy` for external names (e.g. `symbolprefixparse`).
- `-V` Prints on the standard error output the version information for `yacc`.
- `--` If the first filename begins with a dash (`-`), the end of the command-line options must be marked with

—.

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard error output; a more detailed report is found in the `y.output` file. Similarly, if some rules are not reachable from the start symbol, this instance is also reported.

LOCALE

The language of the message texts and yes/no specifications is governed by the environment variable `LC_ALL`, `LC_MESSAGES` or `LANG`.

When the default is set, the system behaves as if it were not internationalized, i.e. the message texts are in English and yes/no specifications must also be made in English (`y` or `n`). You must change one of these variables in order to change the language of the message texts.

The environment variable `LC_CTYPE` determines which characters are defined as letters and digits in the currently valid environment. This is important for symbol names.

If variable `LC_CTYPE` has the value `De_DE.646`, the vertical bar `|` and the opening brace `{`, for example, are interpreted as letters (their code corresponds to the umlauts `ö` and `ä` in the German variant of the ISO 646 character set). This means that `yacc` can no longer interpret the grammar correctly. In this case, the variable `LC_CTYPE` must be set to `En_US.ASCII` or an empty string must be allocated to the `LANG` variable and, if appropriate, also to the `LC_CTYPE` variable.

Detailed information on the dependencies of the environment variables and on internationalization in general can be found in the manual "Programmer's Guide: Internationalization - Localization". Refer also to [environ\(5\)](#) for information on setting the user environment.

FILES

`y.output`, `y.tab.c`, `y.tab.h`
definitions of token names

`yacc.tmp`, `yacc.debug`, `yacc.acts`
temporary files

`LIBDIR/yaccpar`
prototype parser for C programs

`LIBDIR`
usually `/usr/ccs/lib`

SEE ALSO

[lex\(1\)](#).

The "yacc" chapter in the "Guide to Tools for Programming in C".

System calls(2)

Introduction

NAME

`intro_prm2`, `intro_prm3`, `intro_prm3m` - introduction to library functions

DESCRIPTION

Notes on conversion to XPG4

The introduction of XPG4 has brought with it changes in some library functions. This can, in certain circumstances, lead to modified behavior of existing programs and libraries. To limit the resultant problems to a minimum, the functions involved run as standard in XPG3 mode, and therefore behave just as before. Please note the following should you wish to use XPG4 functionality with these functions:

- The `c89` compiler [see `cc(1)`] must be used for generation purposes, as this activates XPG4 mode automatically.
- The `_XPG_IV` environment variable must be exported to the environment at runtime.

The following functions behave differently in the two compilation modes. In the mathematical functions input and results are checked more strictly for the occurrence of NaN and under/overflow in XPG4 mode, and appropriate `errno` values are set.

`acos()`, `asin()`, `atan()`, `atan2()`, `ceil()`, `cosh()`, `erf()`, `erfc()`, `exp()`, `fabs()`, `floor()`, `fprintf()`, `fscanf()`, `gamma()`, `hypot()`, `j0()`, `j1()`, `jn()`, `lgamma()`, `log()`, `log10()`, `modf()`, `nl_langinfo()`, `pow()`, `printf()`, `scanf()`, `sinh()`, `sprintf()`, `sqrt()`, `sscanf()`, `strftime()`, `sysconf()`, `system()`, `y0()`, `y1()`, `yn()`.

The XPG4 functionality of the `sigaction` routine can only be implemented with the new `c89` compiler [see `sigaction(2)`]

C LIBRARIES AND FUNCTION GROUPS

Each function is assigned to a function group, according to the library in which it occurs. The function group is indicated by the number in parentheses after the function name.

The functions identified by (2), (3C) and (3S) form the C standard library. Those marked with (3E) comprise the ELF library, and those marked with (3M) the math library. These three libraries are described in detail below.

The functions marked with (3G) comprise the "general-purpose" library `libgen`. This library is not implemented as a shared library and is not automatically linked by the C compilation system. Specify `-lgen` when calling the compiler in order to link this library.

The functions marked with (3X) occur in various special libraries. The files in which these libraries can be found are specified in the appropriate sections.

The function declarations are provided in the `#include` files specified in each section.

If you redefine these functions, external variables or macros, the new definitions (and not those predefined in the system) are used when they are called. All other names can be redefined without affecting the behavior of other library functions. Note however that new definitions can cause conflicts with declarations in other header files.

The header files in `INCDIR` make prototypes (function declarations including the types of arguments) available for most of the functions described in this manual. Function prototypes enable the compiler to check the correct usage of these functions in the user program. You can also use `lint`, a program checker, which reports discrepancies even if the header files involved are not linked. Definitions for the functions marked with (2), (3C) and (3S) are checked automatically. Other definitions can be included in the `lint` check by specifying the `-l` option; for example, `-lm` includes definitions for `libm`. You should therefore always use the `lint` program checker to check your programs for syntax and semantics.

The C standard library

The C standard library which comprises the functions labeled with (2), (3C) and (3S) is automatically loaded by the C compilation system. It is implemented as a shared library named `libc.so`, and is also available as a static library under the name `libc.a`. Unless instructions to the contrary are given in the compiler command line, C programs are linked with the shared version of the C standard library. To invoke static linking you should specify the option `-dn` in the compiler command line (see description of the `cc(1)` command).

libelf

Using the functions in the ELF access library, a program can manipulate object files, archive files, and archive members which are in the ELF format (Executable and Linking Format). The header file `libelf.h` provides

type and function declarations for all library services. Specify `-lelf` when calling the compiler in order to link this library.

Programs communicate with many of the higher-level routines using an *ELF descriptor*. That is, when the program starts working with a file, `elf_begin()` creates an ELF descriptor through which the program manipulates the structures and information in the file. These ELF descriptors can be used both to read and to write files. After the program establishes an ELF descriptor for a file, it may then obtain *section descriptors* to manipulate the sections of the file [see `elf_getscn(3E)`]. Sections hold the bulk of an object file's real information, such as text, data, the symbol table, and so on. A section descriptor "belongs" to a particular ELF descriptor, just as a section belongs to a file. Finally, *data descriptors* are available through section descriptors, allowing the program to manipulate the information associated with a section. A data descriptor "belongs" to a section descriptor.

Descriptors provide private handles to a file and its pieces. In other words, a data descriptor is associated with one section descriptor, which is associated with one ELF descriptor, which is associated with one file.

Although descriptors are private, they also give access to shared data. Consider programs that combine input files, using incoming data to create or update another file. Such a program might get data descriptors for an input and an output section. To update the output descriptor, the data obtained via the input descriptor could be reused. That is, the descriptors are distinct, but they could share the associated data bytes. This sharing avoids the space overhead for duplicate buffers and the performance overhead for copying data unnecessarily.

File classes

ELF provides a framework in which to define a family of object files, supporting multiple processors and architectures. An important distinction among object files is the *class*, or capacity, of the file. The 32-bit class supports architectures in which a 32-bit object can represent addresses, file sizes, etc., as in the following.

Name	Purpose
<code>Elf32_Addr</code>	Address (unsigned)
<code>Elf32_Half</code>	Medium integer (unsigned)
<code>Elf32_Off</code>	File offset (unsigned)
<code>Elf32_Sword</code>	Large integer with sign
<code>Elf32_Word</code>	Large integer (unsigned)
<code>unsigned char</code>	Small integer (unsigned)

Other classes will be defined as necessary to support larger (or smaller) machines. Some library services deal only with data objects for a specific class, while others are class-independent. To make this distinction clear, library function names reflect their status, as described below.

Data representations

Conceptually, two parallel sets of objects support cross compilation environments. One set corresponds to file contents, while the other set corresponds to the native memory image of the program manipulating the file. Type definitions supplied by the header files work on the native machine, which may have different data encodings (size, byte order, etc.) than the target machine. Although native memory objects should be at least as big as the file objects (to avoid information loss), they may be bigger if that is more natural for the host machine.

Translation facilities exist to convert between file and memory representations. Some library routines convert data automatically, while others leave conversion as the program's responsibility. Either way, programs that create object files must write file-typed objects to those files; programs that read object files must take a

similar view. See `elf_xlate(3E)` and `elf_fsize(3E)` for more information.

Programs may translate data explicitly, taking full control over the object file layout and semantics. If the program prefers not to have and exercise complete control, the library provides a higher-level interface that hides many object file details. `elf_begin()` and related functions let a program deal with the native memory types, converting between memory objects and their file equivalents automatically when reading or writing an object file.

Elf versions

Object file versions allow ELF to adapt to new requirements. Three (independent) versions can be important to a program. First, an application program knows about a particular version by virtue of being compiled with certain header files. Second, the access library similarly is compiled with header files that control what versions it understands. Third, an ELF object file holds a value identifying its version, determined by the ELF version known by the file's creator. Ideally, all three versions would be the same, but they may differ.

- If a program's version is newer than the access library, the program might use information unknown to the library. Translation routines might not work properly, leading to undefined behavior. This condition merits installing a new library.
- The library's version might be newer than the program's and the file's. The library understands old versions, thus avoiding compatibility problems in this case.
- Finally, a file's version might be newer than either the program or the library understands. The program might or might not be able to process the file properly, depending on whether the file has extra information and whether that information can be safely ignored. Again, the safe alternative is to install a new library that understands the file's version.

To accommodate these differences, a program must use `elf_version()` to pass its version to the library, thus establishing the *working version* for the process. Using this, the library accepts data from and presents data to the program in the proper representations. When the library reads object files, it uses each file's version to interpret the data. When writing files or converting memory types to the file equivalents, the library uses the program's working version for the file data.

System services

As mentioned above, `elf_begin()` and related routines provide a higher-level interface to ELF files, performing input and output on behalf of the application program. These routines assume a program can hold entire files in memory, without explicitly using temporary files. When reading a file, the library routines bring the data into memory and perform subsequent operations on the memory copy. Programs that wish to read or write large object files with this model must execute on a machine with a large process virtual address space. If the underlying operating system limits the number of open files, a program can use `elf_cntl()` to retrieve all necessary data from the file, allowing the program to close the file descriptor and reuse it.

Although the `elf_begin()` interfaces are convenient and efficient for many programs, they might be inappropriate for some. In those cases, an application may invoke the `elf_xlate()` data translation routines directly. These routines perform no input or output, leaving that as the application's responsibility. By assuming a larger share of the job, an application controls its input and output model.

Library names

Names associated with the library take several forms:

`elf_name`

These class-independent names perform some service, *name*, for the program.

`elf32_name`

Service names with an embedded class, 32 here, indicate they work only for the designated class of files.

`Elf_Type`

Data types can be class-independent as well, distinguished by *Type*.

`Elf32_Type`

Class-dependent data types have an embedded class name, 32 here.

ELF_C_CMD

Several functions take commands that control their actions. These values are members of the `Elf_Cmd` enumeration; they range from zero through `ELF_C_NUM-1`.

ELF_F_FLAG

Several functions take flags that control library status and/or actions. Flags are bits that may be combined.

ELF32_FSZ_TYPE

These constants give the file sizes in bytes of the basic ELF types for the 32-bit class of files. See `elf_fsize()` for more information.

ELF_K_KIND

The function `elf_kind()` identifies the *KIND* of file associated with an ELF descriptor. These values are members of the `Elf_Kind` enumeration; they range from zero through `ELF_K_NUM-1`.

ELF_T_TYPE

When a service function, such as `elf_xlate()`, deals with multiple types, names of this form specify the desired *TYPE*. Thus, for example, `ELF_T_EHDR` is directly related to `Elf32_Ehdr`. These values are members of the `Elf_Type` enumeration; they range from zero through `ELF_T_NUM-1`.

Information in the ELF header files is separated into common parts and processor-specific parts. A program can make a processor's information available by including the appropriate header file: `sys/elf_NAME.h` where *NAME* matches the processor name as used in the ELF file header.

Symbol	Processor
M32	AT&T WE 32100
SPARC	SPARC
386	Intel 80386
486	Intel 80486
860	Intel 80860
68K	Motorola 68000
88K	Motorola 88000

Other processors will be added to this table as necessary. For example, a program could use the following code to "see" the processor-specific information for the WE 32100.

```
include <libelf.h>
#include <sys/elf_M32.h>
```

Without the `sys/elf_M32.h` definition, only the common ELF information would be visible.

The math library libm

Declarations for the mathematical functions are obtained by linking the header file `math.h` into your source text. Several generally useful mathematical constants are also defined there.

The math library is not automatically loaded by the C Development System. When calling the compiler, specify option `-l` in order to use the above library (`-lm`).

`libm` contains the full set of double-precision routines and some single-precision routines (designated by the suffix `f`) that give better performance with less precision. Selected routines are optimized for performance. The optimized routines are `sin()`, `cos()`, `tan()`, `atan()`, `atan2()`, `exp()`, `log()`, `log10()`, `pow()` and `sqrt()`, and their single-precision equivalents.

GLOSSARY OF TERMS

This section defines some of the terms used in the further course of this manual. The terms are listed alphabetically.

Background process group

Any process group that is not the foreground process group of a session that has established a connection with a controlling terminal.

Controlling process

A controlling process is a session leader that established a connection to a controlling terminal.

Controlling terminal

A terminal that is associated with a session. Each session may have, at most, one controlling terminal associated with it and a controlling terminal may be associated with only one session. Certain input sequences from the controlling terminal cause signals to be sent to process groups in the session associated with the controlling terminal [see `termio(7)`].

Directory

Directories organize files into a hierarchical system where directories are the nodes in the hierarchy. A directory is a file that catalogs the list of files, including directories (sub-directories), that are directly beneath it in the hierarchy. Entries in a directory file are called links. A link associates a file identifier with a filename. A directory contains at least two links, "." and "..". The link "." refers to the directory itself while ".." refers to its parent directory. The root directory, which is the top-most node of the hierarchy, has itself as its parent directory. The pathname of the root directory is / and the parent directory of the root directory is /.

Driver

In a stream, the driver provides the interface between peripheral hardware and the stream. A driver can also be a pseudo-driver, such as a multiplexer or log driver [see `log(7)`], which is not associated with a hardware device.

Effective user ID and effective group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process' real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the s bit (the set-user-ID bit or set-group ID bit) set [see also `exec(2)`].

File access permissions

Permissions to read, write, and execute/search through a file are granted to a process if one or more of the following are true:

- The effective user ID of the process is superuser.
- The effective user ID of the process matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.
- The effective user ID of the process does not match the user ID of the owner of the file, but the effective group ID of the process matches the group ID of the file and the appropriate access bit of the "group" portion (0070) of the file mode is set.
- The effective user ID of the process does not match the user ID of the owner of the file, the effective group ID of the process does not match the group ID of the file, but the appropriate access bit of the "other" portion (0007) of the file mode is set.

File descriptor

A file descriptor is a small integer used to do I/O on a file. The value of a file descriptor is from 0 to (NOFILES-1). A process may have no more than NOFILES file descriptors open simultaneously. A file descriptor is returned by system calls such as `open()` or `pipe()`. The file descriptor is used as an argument by calls such as `read()`, `write()`, `ioctl()`, and `close()`.

Filename

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

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

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

A filename is sometimes referred to as a pathname component. The interpretation of a pathname component is dependent on the value of `NAME_MAX` associated with the path prefix of that component. If any pathname component is longer than `NAME_MAX`, it shall be considered an error condition in that implementation. Otherwise, the implementation shall use the first `NAME_MAX` bytes of the pathname component.

Foreground process group

Each session that has established a connection with a controlling terminal will distinguish one process group of the session as the foreground process group of the controlling terminal. This group has certain privileges when accessing its controlling terminal that are denied to background process groups.

Message

In a stream, one or more blocks of data or information, with associated STREAMS control structures.

Messages can be of several defined types, which identify the message contents. Messages are used to transfer communication data within a stream.

Message queue

In a stream, a linked list of messages awaiting processing by a module or driver.

Message queue access permissions

In the `msgop` and `msgctl` system call descriptions, the permission required for an operation is interpreted as follows:

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others
00002	Write by others

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

- The effective user ID of the process is superuser.
- The effective user ID of the process matches `msg_perm.cuid` or `msg_perm.uid` in the data structure associated with `msqid` and the appropriate bit of the "user" portion (0600) of `msg_perm.mode` is set.
- The effective group ID of the process matches `msg_perm.cgid` or `msg_perm.gid` and the appropriate bit of the "group" portion (060) of `msg_perm.mode` is set.
- The appropriate bit of the "other" portion (006) of `msg_perm.mode` is set.

Otherwise, the corresponding permissions are denied.

Message queue identifier

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

```
struct    ipc_perm    msg_perm;
```

```

struct    msg          *msg_first;
struct    msg          *msg_last;
ulong     msg_cbytes;
ulong     msg_qnum;
ulong     msg_qbytes;
pid_t     msg_lspid;
pid_t     msg_lrpid;
time_t    msg_stime;
long      msg_pad1;      /* reserved */
time_t    msg_rtime;
long      msg_pad2;      /* reserved */
time_t    msg_ctime;
long      msg_pad3;      /* reserved */
long      msg_pad4[4];   /* reserved */

```

There follows a description of the fields of the `msgqid_ds` structure:

`msg_perm`

Is an `ipc_perm` structure that specifies the message operation permissions (see below). This structure includes the following members:

```

uid_t     cuid;          /* creator user id */
gid_t     cgid;          /* creator group id */
uid_t     uid;           /* user id */
gid_t     gid;           /* group id */
mode_t    mode;          /* r/w permission */
ulong     seq;           /* slot usage sequence # */
key_t     key;           /* key */
long      pad[4]         /* reserved */

```

`*msg_first`

Pointer to the first message on the queue.

`*msg_last`

Pointer to the last message on the queue.

`msg_cbytes`

Current number of bytes on the queue.

`msg_qnum`

Number of messages currently on the queue.

`msg_qbytes`

Maximum number of bytes allowed on the queue.

`msg_lspid`

Process ID of the last process that performed a `msgsnd` operation.

`msg_lrpid`

Process ID of the last process that performed a `msgrcv` operation.

`msg_stime`

Time of the last `msgsnd` operation.

`msg_rtime`

Time of the last `msgrcv` operation

`msg_ctime`

Time of the last `msgctl` operation that changed a member of the above structure.

Module

A module is an entity containing processing routines for input and output data. It always exists in the middle of a stream, between the stream's head and a driver. A module is the STREAMS counterpart to the commands in a shell pipeline except that a module contains a pair of functions which allow independent bidirectional (downstream and upstream) data flow and processing.

Multiplexer

A multiplexer is a driver that allows streams associated with several user processes to be connected to a single driver, or several drivers to be connected to a single user process. STREAMS does not provide a general multiplexing driver, but does provide the facilities for constructing them and for connecting multiplexed configurations of streams.

Parent process ID

A new process is created by a currently active process [see also `fork(2)`]. The parent process ID of a process is the process ID of its creator.

Pathname

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

If a pathname begins with a slash, the path search begins at the root directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

Process group

Each process in the system is a member of a process group that is identified by a process group ID. Any process that is not a process group leader may create a new process group and become its leader. Any process that is not a process group leader may join an existing process group that shares the same session as the process. A newly created process joins the process group of its parent.

Process group ID

Each active process is a member of a process group and is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes [see `kill(2)`].

Process group leader

A process group leader is a process whose process ID is the same as its process group ID.

Process group lifetime

A process group lifetime begins when the process group is created by its process group leader, and ends when the lifetime of the last process in the group ends or when the last process in the group leaves the group.

Process ID

Each process in the system is uniquely identified during its lifetime by a positive integer called a process ID. A process ID may not be reused by the system until the process lifetime, process group lifetime and session lifetime ends for any process ID, process group ID and session ID equal to that process ID.

Process lifetime

A process lifetime begins when the process is forked and ends after it exits, when its termination has been acknowledged by its parent process [see `wait(2)`].

Read queue

In a stream, the message queue in a module or driver containing messages moving upstream.

Real user ID and real group ID

Each user allowed on the system is identified by a positive integer (0 to `MAXUID`) called a real user ID.

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

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

Root directory and current working directory

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

Semaphore identifier

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

```
struct ipc_perm  sem_perm;      /* operation permission struct */
struct sem       *sem_base;    /* ptr to first semaphore in set */
ushort          sem_nsems;     /* number of sems in set */
time_t          sem_otime;     /* last operation time */
long            sem_pad1;      /* reserved */
time_t          sem_ctime;     /* last change time */
                                /* Times measured in secs since */
                                /* 00:00:00 GMT, Jan. 1, 1970 */
long            sem_pad2;      /* reserved */
long            sem_pad3[4];    /* reserved */
```

`sem_perm`

Is an `ipc_perm` structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
uid_t          uid;           /* user id */
gid_t          gid;           /* group id */
uid_t          cuid;          /* creator user id */
gid_t          cgid;          /* creator group id */
mode_t         mode;          /* read/amend permission */
ulong          seq;           /* slot usage sequence number */
key_t          key;           /* key */
long           pad[4];        /* reserved */
```

`sem_nsems`

The number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a `sem_num`. `sem_num` values run sequentially from 0 to the value of `sem_nsems` minus 1.

`sem_otime`

Time of the last semop operation.

`sem_ctime`

Time of the last semctl operation that changed a member of the above structure.

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

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

`semval`

Non-negative integer that is the actual value of the semaphore.

`sempid`

Process ID of the last process that performed a semaphore operation on this semaphore.

`semncnt`

Count of the number of processes that are currently suspended awaiting this semaphore's `semval` to become greater than its current value.

`semzcnt`

Count of the number of processes that are currently suspended awaiting this semaphore's `semval` to

become 0.

Semaphore operation access permissions

In the `semop()` and `semctl()` system call descriptions, the permission required for an operation is interpreted as follows:

00400	Read by user
00200	Alter by user
00040	Read by group
00020	Alter by group
00004	Read by others
00002	Alter by others

Read and alter permissions on a `semid` are granted to a process if one or more of the following are true:

- The effective user ID of the process is superuser.
- The effective user ID of the process matches `sem_perm.cuid` or `sem_perm.uid` in the data structure associated with `semid` and the appropriate bit of the "user" portion (0600) of `sem_perm.mode` is set.
- The effective group ID of the process matches `sem_perm.cgid` or `sem_perm.gid` and the appropriate bit of the "group" portion (060) of `sem_perm.mode` is set.
- The appropriate bit of the "other" portion (06) of `sem_perm.mode` is set.

Otherwise, the corresponding permissions are denied.

Session

A session is a group of processes identified by a common ID called a session ID, capable of establishing a connection with a controlling terminal. Any process that is not a process group leader may create a new session and process group, becoming the session leader of the session and process group leader of the process group. A newly created process joins the session of its creator.

Session ID

Each session in the system is identified during its lifetime by a positive integer called a session ID, the process ID of its session leader.

Session leader

A session leader is a process whose session ID is the same as its process and process group ID.

Session lifetime

A session lifetime begins when the session is created by a logon process, and ends when the lifetime of the last process that is a member of the session ends, or when the last process that is a member in the session leaves the session.

Shared memory access permissions

In the `shmop()` and `shmctl()` system call descriptions, the permission required for an operation is interpreted as follows:

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others

00002

Write by others

Read and write permissions on a shared memory identifier (`shmid`) are granted to a process if one or more of the following are true:

- The effective user ID of the process is superuser.
- The effective user ID of the process matches `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with `shmid` and the appropriate bit of the "user" portion (0600) of `shm_perm.mode` is set.
- The effective group ID of the process matches `shm_perm.cgid` or `shm_perm.gid` and the appropriate bit of the "group" portion (060) of `shm_perm.mode` is set.
- The appropriate bit of the "other" portion (06) of `shm_perm.mode` is set.
- Otherwise, the corresponding permissions are denied.

Shared memory identifier

A shared memory identifier (`shmid`) is a positive integer created by a `shmget()` system call. Each `shmid` has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed. The data structure is referred to as `shmid_ds` and contains the following members:

```
struct    ipc_perm  shm_perm;          /* operation permission struct */
int       shm_segsz; /* size of segment in bytes */
struct    anon-map  *shm_amp;         /* segment anon_map pointer */
ushort    shm_lkcnt; /* number of times it is being
                        locked */

pid_t     shm_lpid; /* pid of last shmop */
pid_t     shm_cpid; /* pid of creator */
ulong     shm_nattch; /* used only for shminfo */
ulong     shm_cnattch; /* used only for shminfo */
time_t    shm_atime; /* last shmat time */
long      shm_pad1; /* reserved */
time_t    shm_dtime; /* last shmdt time */
long      shm_pad2; /* reserved */
time_t    shm_ctime; /* last change time */
long      shm_pad3; /* reserved */
long      shm_pad4[4]; /* reserved */
```

There follows a description of the fields of the `shmid_ds` structure:

`shm_perm`

Is an `ipc_perm` structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
uid_t     cuid;      /* creator user id */
gid_t     cgid;      /* creator group id */
uid_t     uid;       /* user id */
gid_t     gid;       /* group id */
mode_t    mode;      /* r/w permission */
ulong     seq;       /* slot usage sequence # */
key_t     key;       /* key */
long      pad[4];    /* reserved */
```

`shm_segsz`

specifies the size of the shared memory segment in bytes.

`shm_cpid`

is the process ID of the process that created the shared memory identifier.

`shm_lpid`

the process ID of the last process that performed a `shmop()` operation.

`shm_nattch`

the number of processes that currently have this segment attached.

`shm_atime`

the time of the last `shmat()` operation [see [shmop\(2\)](#)].

`shm_dtime`

the time of the last `shmdt()` operation [see [shmop\(2\)](#)].

`shm_ctime`

the time of the last `shmctl()` operation that changed one of the members of the above structure.

Special processes

The process with ID 0 and the process with ID 1 are special processes referred to as `proc0` and `proc1` [see [kill\(2\)](#)]. `proc0` is the process scheduler. `proc1` is the initialization process (`init`); `proc1` is the ancestor of every other process in the system and is used to control the process structure.

STREAMS

A set of kernel mechanisms that support the development of network services and data communication drivers. It defines interface standards for character input/output within the kernel and between the kernel and user level processes. The STREAMS mechanism is composed of utility routines, kernel facilities and a set of data structures.

Stream

A stream is a full-duplex data path within the kernel between a user process and driver routines. The primary components are a stream head, a driver and zero or more modules between the stream head and driver. A stream is analogous to a shell pipeline except that data flow and processing are bidirectional; downstream is the direction from the stream head to the driver, upstream the direction from the driver to the stream head.

Stream head

In a stream, the stream head is the end of the stream that provides the interface between the stream and a user process. The principle functions of the stream head are processing STREAMS-related system calls, and passing data and information between a user process and the stream.

Superuser

A process is recognized as a superuser process and is granted special privileges, such as immunity from file permissions, if its effective user ID is 0.

Write queue

In a stream, the message queue in a module or driver containing messages moving downstream.

ERROR NUMBERS

Most of the functions described below have one or more error status values.

In most cases an error is indicated by a return value, almost always `-1` or the `NULL` pointer. Details are provided with each description. In addition an error number is placed in the external variable `errno`. If a call is successful, `errno` is not reset. In some cases, the value of `errno` must be queried to ascertain whether the function has completed successfully.

Below is a complete list of error numbers and their names, as defined in `errno.h` [see [errno\(5\)](#)]. If the meaning of the error number for a particular function differs from that given here, it will be explained in the description of the function.

1 EPERM

Not owner. Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or the superuser. It is also returned for attempts by ordinary users to do things allowed only to the superuser.

2 ENOENT

No such file or directory. A filename is specified and the file should exist but doesn't, or one of the

directories in a *path* name does not exist.

3 ERCH

No such process. No process can be found corresponding to that specified by PID in the `kill()` or `ptrace()` routine.

4 EINTR

Interrupted system call. An asynchronous signal (such as `interrupt` or `quit`), which should not be ignored, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO

I/O error. Some physical I/O error has occurred. This error may in some cases not appear until the next system call [e.g. an error during `write()` may be reported after `close()`].

6 ENXIO

No such device or address. An I/O operation refers to a device which cannot be accessed.

7 E2BIG

Arg list too long. An argument list longer than `ARG_MAX` bytes is presented to a member of the `exec` family of routines. The argument list limit is the sum of the size of the argument list plus the size of the environment's exported shell variables.

8 ENOEXEC

Exec format error. A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid format [see `a.out(4)`].

9 EBADF

Bad file number. A file descriptor refers to no open file.

10 ECHILD

No child processes. A `wait()` routine was executed by a process which had no existing or awaited for child processes.

11 EAGAIN

Out of system resources. The `fork()` call failed because the system's process table is full or the user is not allowed to start any more processes. It is also possible that a system call failed because of insufficient memory or swap space.

12 ENOMEM

Not enough space. During execution of an `exec()`, `brk()`, or `sbrk()` routine, a program asks for more space than the system is able to supply. The maximum usable size is defined by a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during the `fork()` routine.

13 EACCES

Permission denied. A file has been accessed in a way forbidden by the file access permissions.

14 EFAULT

Bad address. The system encountered a hardware fault in attempting to use an argument of a routine. For example, `errno` potentially may be set to `EFAULT` any time a routine that takes a pointer argument is passed an invalid address, if the system can detect the condition. Because systems will differ in their ability to reliably detect a bad address, on some implementations passing a bad address to a routine will result in undefined behavior.

15 ENOTBLK

Block device required. A non-block file was mentioned where a block device was required (e.g. in a call to the `mount()` routine).

16 EBUSY

Resource not available. An attempt was made to mount a device that was already mounted or an attempt was made to unmount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled.

17 EEXIST

- File exists. An existing file was mentioned in an inappropriate context (e.g., call to the `link()` routine).
- 18 **EXDEV**
Cross-device link. A link to a file on another device was attempted.
- 19 **ENODEV**
No such device. An attempt was made to apply an inappropriate operation to a device (e.g. read a write-only device).
- 20 **ENOTDIR**
Not a directory. A non-directory was specified where a directory is required (e.g., in a path prefix or as an argument to the `chdir(2)` routine).
- 21 **EISDIR**.
Is a directory. An attempt was made to write on a directory.
- 22 **EINVAL**
Invalid argument. An invalid argument was specified; an undefined signal was used in a call to the `signal(2)` or `kill(2)` routine.
- 23 **ENFILE**
File table overflow. The system file table is full, i.e. `SYS_OPEN` files are open, and temporarily no more files can be opened.
- 24 **EMFILE**
Too many open files. No process may have more than `OPEN_MAX` file descriptors open at a time.
- 25 **ENOTTY**
Not a typewriter. A call was made to the `ioctl()` routine specifying a file that is not a character special file.
- 26 **ETXTBSY**
Text file busy. An attempt was made to execute a pure-procedure program that is currently open for writing. Also an attempt to open for writing or to remove a pure-procedure program that is being executed.
- 27 **EFBIG**
File too large. The size of a file exceeded the maximum file size, `FCHR_MAX` [see `getrlimit(2)`].
- 28 **ENOSPC**
No space left on device. While writing a file, there is no free space left on the data medium. In the `fcntl()` routine, the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.
- 29 **ESPIPE**
Illegal seek. A call to the `lseek()` routine was issued to a pipe.
- 30 **EROFS**
Read-only file system. An attempt to modify a file or directory in a read-only file system.
- 31 **EMLINK**
Too many links. An attempt to make more than the maximum number of links (`LINK_MAX`) to a file.
- 32 **EPIPE**
Broken pipe. A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.
- 33 **EDOM**
Math argument out of domain of function. The argument of a mathematical function (3M) is out of the domain of the function.
- 34 **ERANGE**
Math result not representable. The value of a mathematical function (3M) is not representable within machine precision.
- 35 **ENOMSG**
No message of desired type. An attempt was made to receive a message of a type that does not exist on the specified message queue [see `msgop(2)`].
- 36 **EIDRM**

Identifier removed. This error is returned to processes that resume execution due to the removal of an identifier from the interprocess communication system [see `msgctl(2)`, `semctl(2)`, and `shmctl(2)`].

- 37 ECHRNG
Channel number out of range.
- 38 EL2NSYNC
Level 2 not synchronized.
- 39 EL3HLT
Level 3 halted.
- 40 EL3RST
Level 3 reset.
- 41 ELNRNG
Link number out of range.
- 42 EUNATCH
Protocol driver not attached.
- 43 ENOCSI
No CSI structure available.
- 44 EL2HLT
Level 2 halted.
- 45 EDEADLK
Risk of a deadlock. A deadlock situation was detected and avoided. This error pertains to file and record locking.
- 46 ENOLCK
No record locks available. There are no more locks available. The system lock table is full [see `fcntl(2)`].
- 47-49
Reserved numbers.
- 50 EBADE
Bad exchange descriptor.
- 51 EBADR
Bad request descriptor.
- 52 EXFULL
Message tables full.
- 53 ENOANO
Anode table overflow.
- 54 EBADRQC
Bad request code.
- 55 EBADSLT
Invalid slot.
- 56 EDEADLOCK
File locking deadlock.
- 57 EBFONT
Bad font file format.
- 58-59
Reserved numbers.
- 60 ENOSTR
Device not a stream. A `putmsg()` or `getmsg()` system call was attempted on a file that is not a STREAMS device.
- 61 ENODATA
No data available.

- 62 **ETIME**
Timer expired. The timer set for a STREAMS `ioctl()` call has expired. The cause of this error is device-specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the `ioctl()` operation is indeterminate.
- 63 **ENOSR**
Out of stream resources. During a STREAMS `open()`, either no STREAMS queues or no STREAMS head data structures were available. This is a temporary condition; one may recover from it if other processes release resources.
- 64 **ENONET**
Machine is not on the network.
- 65 **ENOPKG**
Package not installed. This error occurs when users attempt to use a system call from a package which has not been installed.
- 66 **EREMOTE**
Object is remote.
- 67 **ENOLINK**
Link has been severed.
- 68 **EADV**
Advertise error.
- 69 **ESRMNT**
Srmount error.
- 70 **ECOMM**
Communication error when sending.
- 71 **EPROTO**
Protocol error. This error is device-specific, but is generally not related to a hardware failure.
- 72-73
Reserved numbers.
- 74 **EMULTIHOP**
Multihop attempted.
- 75-76
Reserved numbers.
- 77 **EBADMSG**
Not a data message. During a `read()`, `getmsg()`, or `ioctl()` system call to a STREAMS device, something has come to the head of the queue that can't be processed. That something depends on the system call:
`read()`
Control information or a passed file descriptor.
`getmsg()`
Passed file descriptor.
`ioctl()`
Control or data information.
- 78 **ENAMETOOLONG**
Filename too long. The length of the pathname argument exceeds `PATH_MAX` or its length exceeds `NAME_MAX` [see [limits\(4\)](#)].
- 79 **EOVERFLOW**
Value too large for defined data type.
- 80 **ENOTUNIQ**
Name not unique on network.
- 81 **EBADFD**
File descriptor in bad state. Either a file descriptor refers to no open file or a read request was made to

a file that is open only for writing.

82 EREMCHG

Remote address changed.

83 ELIBACC

Cannot access a needed shared library. Trying to use `exec()` to start an `a.out` that requires a static shared library and the static shared library doesn't exist or the user doesn't have permission to use it.

84 ELIBBAD

Accessing a corrupted shared library. Trying to use `exec()` to start an `a.out` that requires a static shared library (to be linked in) and `exec()` could not load the static shared library. The static shared library is probably corrupted.

85 ELIBSCN

`.lib` section in `a.out` corrupted. Trying to `exec()` an `a.out` that requires a static shared library (to be linked in) and there was erroneous data in the `.lib` section of the `a.out`. The `.lib` section tells `exec` what static shared libraries are needed.

86 ELIBMAX

Attempting to link in more shared libraries than system limit. Trying to `exec()` an `a.out` that requires more static shared libraries than is allowed on the current configuration of the system.

87 ELIBEXEC

Cannot `exec()` a shared library directly. Attempting to `exec()` a shared library directly.

88 EILSEQ

Illegal byte sequence. Handle multiple characters as a single character.

89 ENOSYS

Operation not applicable.

90 ELOOP

Number of symbolic links encountered during *path* name traversal exceeds `MAXSYMLINKS`.

91 ERESTART

Interrupted system call should be restarted.

92 ESTRPIPE

If pipe/FIFO, don't sleep in stream head.

93 ENOTEMPTY

Directory not empty.

94 EUSERS

Too many users (for UFS).

95 ENOTSOCK

Socket operation on non-socket.

96 EDESTADDRREQ

Destination address required. A required address was omitted from an operation on a transport endpoint.

97 EMSGSIZE

Message too long. A message sent on a transport provider was larger than the internal message buffer or some other network limit.

98 EPROTOTYPE

Protocol wrong type for socket. A protocol was specified that does not support the semantics of the socket type requested.

99 ENOPROTOOPT

Protocol not available. A bad option or level was specified when getting or setting options for a protocol.

100-109

Reserved numbers.

- 110 **ENOLIMFILE**
Inode database not open. If Sharell is active all the processes are attached to the root Inode. Refer to the manual page for the Sharell `charge` command which is used to open and close the database. Sharell closes the database automatically if the current license expires.
- 111 **EPROCLIM**
Process limit reached. A `fork()` can fail if the Sharell process limit is reached for the Inode to which the forking process is attached. Note that process limits are applied up the Inode hierarchy.
- 112 **EDISJOINT**
Inode hierarchy is disjointed. Either by data corruption or forced removal of an Inode with children, the Sharell Inode database contains an orphan Inode. Processes cannot attach to an orphan Inode.
- 113 **ENOLOGIN**
No logins allowed for user. The user has a Sharell `flag.nologin` flag which evaluates as set.
- 114 **ELOGINLIM**
Login limit reached. The user has a Sharell `flag.onelogin` flag which evaluates as set and the Sharell logins count attribute for this user is already one.
- 115 **EGROUPLOOP**
Loop in Inode hierarchy. When activating an Inode, the depth of the Inode tree was found to be deeper than the configured maximum. When changing the parent of an Inode, the depth of the relocated Inode would have exceeded the configured maximum.
- 116 **ENOATTACH**
Attach to Inode not allowed. The Inode has a Sharell `flag.noattach` flag which evaluates as set.
- 117-119
Reserved numbers.
- 120 **EPROTONOSUPPORT**
Protocol not supported. The protocol has not been configured into the system or no implementation for it exists.
- 121 **ESOCKTNOSUPPORT**
Socket type not supported. The support for the socket type has not been configured into the system or no implementation for it exists.
- 122 **EOPNOTSUPP**
Operation not supported on transport endpoint. For example, trying to accept a connection on a datagram transport endpoint.
- 123 **EPFNOSUPPORT**
Protocol family not supported. The protocol family has not been configured into the system or no implementation for it exists. This error arises with Internet protocols.
- 124 **EAFNOSUPPORT**
Address family not supported by protocol family. An address incompatible with the requested protocol was used.
- 125 **EADDRINUSE**
Address already in use. User attempted to use an address already in use, and the protocol does not allow this.
- 126 **EADDRNOTAVAIL**
Cannot assign requested address. Results from an attempt to create a transport endpoint with an address not on the current machine.
- 127 **ENETDOWN**
Network is down. Operation encountered a dead network.
- 128 **ENETUNREACH**
Network is unreachable. Operation was attempted to an unreachable network.
- 129 **ENETRESET**
Network dropped connection because of reset. The host you were connected to crashed and rebooted.
- 130 **ECONNABORTED**

Software caused connection abort. A connection abort was caused internally within your host machine.

131 **ECONNRESET**

Connection reset by connected party. The connection was closed by the connected party. This normally results from a timeout or a reboot of the machine.

132 **ENOBUFS**

No buffer space available. An operation on a transport endpoint or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.

133 **EISCONN**

Transport endpoint is already connected. A connect request has been made, or a `sendto` or `sendmsg` specified a destination address, even though the endpoint was already connected.

134 **ENOTCONN**

Transport endpoint is not connected. A request to send or receive data was disallowed because the transport endpoint is not connected and (when sending a datagram) no address was supplied.

135 **EUCLEAN**

Structure needs cleaning.

136 **Reserved number.**

137 **ENOTNAM**

Not a name file.

138 **ENAVAIL**

Not available.

139 **EISNAM**

Is a name file.

140 **EREMOTEIO**

Remote I/O error.

141 **EINIT**

Reserved number.

142 **EREMDEV**

Reserved number.

143 **ESHUTDOWN**

Cannot send after transport endpoint shutdown. A request to send data was disallowed because the transport endpoint has already been shut down.

144 **ETOOMANYREFS**

Too many references: cannot splice.

145 **ETIMEDOUT**

Connection timed out. A connect or send request failed because the connected party did not properly respond after a period of time. The timeout period is dependent on the communication protocol.

146 **ECONNREFUSED**

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

147 **EHOSDOWN**

Host is down. A transport provider operation failed because the destination host was down.

148 **EHOSTUNREACH**

No route to host. A transport provider operation was attempted to an unreachable host.

149 **EALREADY**

Operation already in progress. An operation was attempted on a non-blocking object that already had an operation in progress.

150 **EINPROGRESS**

Operation now in progress. An operation that takes a long time to complete (such as a `connect`) was attempted on a non-blocking object.

151 **ESTALE**

Ineffective NFS file identifier.

152-157
Reserved numbers.

158 ECANCELED
Operation canceled.

159 EHANDLER
Signal handler exists.

160-199
Reserved numbers.

500 EIORESID
Block not fully transferred.

DIRECTORIES

INCDIR

Standard directory for #include files (usually /usr/include or /usr/include/sys)

SEE ALSO

[intro_prm1\(1\)](#), [intro_prm5\(5\)](#).

Chapter on "Object files" in the "Guide to Tools for Programming in C".

access(2) - getuid(2)

NAME

`access` - determine access permissions for a file

SYNOPSIS

```
#include <unistd.h>

int access(const char *path, int amode);
```

DESCRIPTION

path points to a pathname naming a file. `access()` checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed by an OR of the following constants (defined in `unistd.h`):

R_OK read
W_OK write
X_OK execute (search)
F_OK check existence of file

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `access()` function will fail if:

EACCES

Permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the path prefix.

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.

EROFS

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

The `access()` function may fail if:

EINVAL

The value of the *amode* argument is invalid.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

ETXTBSY

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

RESULT

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

SEE ALSO

`chmod(2)`, `stat(2)`, `unistd(4)`.

Section on "File access permissions" in `intro_prm2(2)`.

NAME

`acct` - enable or disable process accounting

SYNOPSIS

```
#include <unistd.h>
int acct(const char *path);
```

DESCRIPTION

`acct()` enables or disables the system process accounting routine. If the routine is enabled, an accounting record will be written in an accounting file for each process that terminates. The termination of a process can be caused by one of two things: an `exit()` call or a signal [see `exit(2)` and `signal(2)`]. The effective user ID of the process calling `acct()` must be superuser.

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

The accounting routine is enabled if *path* is not equal to `(char *)NULL` and no errors occur during the system call. It is disabled if *path* is `(char *)NULL` and no errors occur during the system call.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`acct()` will fail if one or more of the following apply:

EACCES

The file named by *path* is not an ordinary file.

EBUSY

An attempt is being made to enable accounting using the same file that is currently being used.

EFAULT

path points to an illegal address.

ELOOP

Too many symbolic links were encountered in translating *path*.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a *path* component exceeds `NAME_MAX`.

ENOTDIR

A component of the path prefix is not a directory.

ENOENT

One or more components of the accounting file pathname do not exist.

EPERM

The effective user of the calling process is not superuser.

EROFS

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

RETURN VALUE

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

SEE ALSO

`exit(2)`, `signal(2)`, `acct(4)`, `unistd(4)`.

adjtime(2)**NAME**

`adjtime` - correct the time to allow synchronization of the systemclock

SYNOPSIS

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

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

DESCRIPTION

`adjtime()` adjusts the system's notion of the current time, as returned by `gettimeofday(3C)`, advancing or retarding it by the amount of time specified in the `struct timeval` pointed to by `delta`.

The adjustment is made by speeding up (if that amount of time is positive) or slowing down (if that amount of time is negative) the system's clock. If the amount of time is positive, the time moves twice as fast as before. If the amount of time is negative, the time moves half as fast as before. The time is always a monotonically increasing function. A time correction from an earlier call to `adjtime()` may not be finished when `adjtime()` is called again. If `delta` is 0, then `olddelta` returns the status of the effects of the previous `adjtime()` call and there is no effect on the time correction as a result of this call. If `olddelta` is not a `NULL` pointer, then the structure it points to will contain, upon return, the number of seconds and/or microseconds still to be corrected from the earlier call. If `olddelta` is a `NULL` pointer, the corresponding information will not be returned.

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

Only the superuser may adjust the time of day.

The adjustment value will be rounded to the resolution of the system clock.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`adjtime()` will fail if one or more of the following apply:

EFAULT

`delta` or `olddelta` points outside the process' allocated address space, or `olddelta` points to a region of the process' allocated address space that is not writable.

EPERM

The process' effective user ID is not that of the superuser.

RETURN VALUE

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

SEE ALSO

`date(1)`, `gettimeofday(3C)`, `sys_time(5)`.

alarm(2)**NAME**

alarm - set a process alarm clock

SYNOPSIS

```
#include <unistd.h>
unsigned int alarm(unsigned int sec);
```

DESCRIPTION

`alarm()` instructs the alarm clock of the calling process to send the signal `SIGALRM` to the calling process after the number of real time seconds specified by `sec` have elapsed [see [signal\(2\)](#)].

Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

If `sec` is 0, any previously made alarm request is canceled.

`fork()` sets the alarm clock of a new process to 0 [see [fork\(2\)](#)]. A process created by the `exec` family of routines inherits the time left on the old process' alarm clock.

Interactions between `alarm()` and any of `setitimer()`, `ualarm()` or `usleep()` are unspecified.

RETURN VALUE

`alarm()` returns the amount of time previously remaining in the alarm clock of the calling process.

SEE ALSO

[exec\(2\)](#), [fork\(2\)](#), [pause\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [sigset\(2\)](#), [ualarm\(3\)](#), [usleep\(3\)](#), [getitimer\(3C\)](#), [unistd\(4\)](#).

NAME

`brk`, `sbrk` - change data segment space allocation

SYNOPSIS

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t incr);
```

DESCRIPTION

`brk()` and `sbrk()` are used to change dynamically the amount of memory space allocated for the calling process' data segment [see [exec\(2\)](#)]. The change is made by resetting the process' break value and allocating the appropriate amount of memory space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated memory space increases as the break value increases. Newly allocated memory space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

`brk()` sets the break value to *addr* and changes the allocated space accordingly.

`sbrk()` adds *incr* bytes to the break value and changes the allocated space accordingly. *incr* can be negative, in which case the amount of allocated memory space is decreased. The current value of the program break is returned by `sbrk(0)`.

The behaviour of `brk()` and `sbrk()` is unspecified if an application also uses any other memory functions [such as `malloc()`, `mmap()`, `free()`]. Other functions may use these other memory functions silently.

It is unspecified whether the pointer returned by `sbrk()` is aligned suitably for any purpose.

This interface need not be thread-safe.

RETURN VALUE

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

Upon successful completion, `sbrk()` returns the prior break value. Otherwise, it returns `(void*)-1` and sets `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `brk()` and `sbrk()` functions will fail if:

ENOMEM

The requested change would allocate more space than allowed.

The `brk()` and `sbrk()` functions may fail if:

EAGAIN

The total amount of system memory available for allocation to this process is temporarily insufficient. This may occur even though the space requested was less than the maximum data segment size.

ENOMEM

The requested change would be impossible as there is insufficient swap space available, or would cause a memory allocation conflict.

APPLICATION USAGE

The `brk()` and `sbrk()` functions have been used in specialized cases where no other memory allocation function provided the same capability. The use of `malloc()` is now preferred because it can be used portably with all other memory allocation functions and with any function that uses other allocation functions.

It is unspecified whether the pointer returned by `sbrk()` is aligned suitably for any purpose.

SEE ALSO

`exec(2)`, `mmap(2)`, `malloc(3C)`, `unistd(4)`.

NAME

`chdir`, `fchdir` - change working directory

SYNOPSIS

```
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fildes);
```

DESCRIPTION

`chdir()` and `fchdir()` cause a directory pointed to by *path* or *fildes* to become the current working directory, the starting point for path searches for pathnames not beginning with `/`. *path* points to the pathname of a directory. The *fildes* argument to `fchdir()` is an open file descriptor of a directory.

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`chdir()` will fail and the current working directory will be unchanged if one or more of the following apply:

EACCES

Search permission is denied for any component of the pathname.

EFAULT

path points outside the allocated address space of the process.

EINTER

A signal was caught during the execution of the `chdir()` system call.

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

ELOOP

Too many symbolic links were encountered in translating *path*.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a *path* component exceeds `NAME_MAX`.

ENOTDIR

A component of the pathname is not a directory.

ENOENT

Either a component of the path prefix or the directory named by *path* does not exist or is a null pathname.

EMULTIHOP

Components of *path* require hopping to multiple remote machines.

The `chdir()` function may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

`fchdir()` will fail and the current working directory will be unchanged if one or more of the following apply:

EACCES

Search permission is denied for *fildes*.

EBADF

*fil*des is not an open file descriptor.

EINTR

A signal was caught during the execution of the `fchdir()` system call.

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

ENOLINK

*fil*des points to a remote machine and the link to that machine is no longer active.

ENOTDIR

The open file descriptor *fil*des does not refer to a directory.

RESULT

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

SEE ALSO

`chroot(2)`, `getcwd(3C)`, `unistd(4)`.

NAME

chmod, fchmod - change mode of file

SYNOPSIS

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

DESCRIPTION

chmod() and fchmod() set the access permission portion of the mode of the file whose name is given by *path* or referenced by the descriptor *fildes* to the bit pattern contained in *mode*. Access permission bits are described in <sys/stat.h> [see [stat\(5\)](#)] and are interpreted as follows:

S_ISUID 04000 Set user ID on execution.
 S_ISGID 020#0 Set group ID on execution if # is 7, 5, 3, or 1. Enable mandatory file/record locking if # is 6, 4, 2, or 0
 S_ISVTX 01000 Save text image after execution.
 S_IRWXU 00700 Read, write, execute (search) by owner.
 S_IRUSR 00400 Read by owner.
 S_IWUSR 00200 Write by owner.
 S_IXUSR 00100 Execute (search if a directory) by owner.
 S_IRWXG 00070 Read, write, execute by group.
 S_IRGRP 00040 Read by group.
 S_IWGRP 00020 Write by group.
 S_IXGRP 00010 Execute by group.
 S_IRWXO 00007 Read, write, execute (search) by others.
 S_IROTH 00004 Read by others.
 S_IWOTH 00002 Write by others
 S_IXOTH 00001 Execute by others.

Other modes are constructed by bitwise OR combination of the access permission bits.

The effective user ID of the process must match the owner of the file or the process must have the appropriate privilege to change the mode of a file.

If the process is not a privileged process and the file is not a directory, mode bit 01000 (save text image on execution) is cleared.

If neither the process nor a member of the supplementary group list is privileged, and the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If a 0410 executable file has the sticky bit (mode bit 01000) set, the operating system will not delete the program text from the swap area when the last user process terminates. If a 0413 or `ELF` executable file has the sticky bit set, the operating system will not delete the program text from memory when the last user process terminates. In either case, if the sticky bit is set the text will already be available (either in a swap area or in memory) when the next user of the file executes it, thus making execution faster.

If a directory is writable and has the sticky bit set, files within that directory can be removed (deleted) or renamed only if one or more of the following are true [see `unlink(2)` and `rename(2)`]:

- the user owns the file
- the user owns the directory
- the file is writable by the user
- the user is a privileged user

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record locking will exist on a regular file. This may affect future calls to `open(2)`, `creat(2)`, `read(2)`, and `write(2)` on this file. Mandatory locking is not supported with NFS.

Upon successful completion, `chmod()` and `fchmod()` mark for update the `st_ctime` field of the file.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`chmod()` will fail and the file mode will be unchanged if one or more of the following apply:

EACCES

Search permission is denied on a component of the path prefix of *path*.

EFAULT

path points outside the allocated address space of the process.

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

ELOOP Too many symbolic links were encountered in translating *path*.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and file system type does not allow it.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a *path* component exceeds `NAME_MAX`.

ENOTDIR

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

ENOENT

Either a component of the path prefix, or the file referred to by *path* does not exist or is a null pathname.

ENOLINK

fildev points to a remote machine and the link to that machine is no longer active.

EPERM The effective user ID does not match the owner of the file or the process does not have appropriate privilege.

EROFS The file referred to by *path* resides on a read-only file system.

The `chmod()` function may fail if:

EINTR A signal was caught during execution of the function.

EINVAL

The value of the *mode* argument is invalid.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

The `fchmod()` function will fail and the file mode will be unchanged if:

EBADF *fildev* is not an open file descriptor.

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

ENOLINK

path points to a remote machine and the link to that machine is no longer active.

EPERM The effective user ID does not match the owner of the file or the process does not have appropriate privilege.

EROFS The file referred to by *files* resides on a read-only file system.

The `fchmod()` function may fail if:

EINTR A signal was caught during execution of the `fchmod()` system call.

EINVAL

The value of the *mode* argument is invalid.

EINVAL

The *files* argument refers to a pipe and the implementation disallows execution of `fchmod()` on a pipe.

NOTES

`S_ISUID`, `S_ISGID` and the file permission bits are described in `<sys/stat.h>` [see [stat\(5\)](#)].

If a directory is writable and the mode bit `S_ISVTX` is set on the directory, a process may remove or rename files within that directory only if one or more of the following is true:

- The effective user ID of the process is the same as that of the owner ID of the file.
- The effective user ID of the process is the same as that of the owner ID of the directory.
- The process has appropriate privileges.

If the `S_ISVTX` bit is set on a non-directory file, the behavior is unspecified.

In order to ensure that the `S_ISUID` and `S_ISGID` bits are set, an application requiring this should use `stat()` after a successful `chmod()` in order to verify this.

Any file descriptors currently open by any process on the file may become invalid if the mode of the file is changed to a value which would deny access to that process. One situation where this could occur is on a stateless file system.

SEE ALSO

[chmod\(1\)](#), [chown\(2\)](#), [creat\(2\)](#), [fcntl\(2\)](#), [mknod\(2\)](#), [open\(2\)](#), [read\(2\)](#), [stat\(2\)](#), [write\(2\)](#), [mkfifo\(3C\)](#), [stat\(5\)](#), [types\(5\)](#).

chown(2)**NAME**

`chown`, `lchown`, `fchown` - change owner and group of a file

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>
int chown(const char *path, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
int fchown(int fildes, uid_t owner, gid_t group);
```

DESCRIPTION

The owner ID and group ID of the file specified by *path* or referenced by the descriptor *fildes*, are set to *owner* and *group* respectively. If *owner* or *group* is specified as `(uid_t)-1` or `(gid_t)-1` respectively, the corresponding ID of the file is not changed.

The function `lchown()` sets the owner ID and group ID of the named file just as `chown()` does, except in the case where the named file is a symbolic link. In this case `lchown()` changes the ownership of the symbolic link file itself, while `chown()` changes the ownership of the file or directory to which the symbolic link refers.

If `chown()`, `lchown()`, or `fchown()` is invoked by a process other than superuser, the set-user-ID and set-group-ID bits of the file mode, `S_ISUID` and `S_ISGID` respectively, are cleared [see [chmod\(2\)](#)].

The operating system has a configuration option, `_POSIX_CHOWN_RESTRICTED`, to restrict ownership changes for the `chown()`, `lchown()`, and `fchown()` system calls. When `_POSIX_CHOWN_RESTRICTED` is not in effect, the effective user ID of the process must match the owner of the file or the process must be the superuser to change the ownership of a file. When `_POSIX_CHOWN_RESTRICTED` is in effect, the `chown()`, `lchown()`, and `fchown()` system calls, for users other than superuser, prevent the owner of the file from changing the owner ID of the file and restrict the change of the group of the file to the list of supplementary group IDs.

Upon successful completion, `chown()`, `fchown()` and `lchown()` mark for update the `st_ctime` field of the file.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`chown()` and `lchown()` fail and the owner and group of the named file remain unchanged if one or more of the following apply:

EACCES

Search permission is denied on a component of the path prefix of *path*.

EFAULT

path points outside the allocated address space of the process.

EINTR

A signal was caught during the `chown()` or `lchown()` system calls.

EINVAL

group or *owner* is out of range.

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

ELOOP

Too many symbolic links were encountered in translating *path*.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and file system type does not allow it.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a *path* component exceeds `NAME_MAX`.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

ENOLINK

path points to a remote machine and the link to that machine is no longer active.

ENOTDIR

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

ENOENT

Either a component of the path prefix or the file referred to by *path* does not exist or is a null pathname.

EPERM

The effective user ID does not match the owner of the file or the calling process does not have the appropriate permissions.

EROFS

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

`fchown()` fails and the owner and group of the named file remain unchanged if one or more of the following apply:

EBADF

fildes is not an open file descriptor.

EINVAL

group or *owner* is out of range.

EPERM

The effective user ID does not match the owner of the file or the process is not the superuser and `_POSIX_CHOWN_RESTRICTED` indicates that such privilege is required.

EROFS

The named file referred to by *fildes* resides on a read-only file system.

EINTR

A signal was caught during the `chown()` or `lchown()` system calls.

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

ENOLINK

fildes points to a remote machine and the link to that machine is no longer active.

RESULT

Upon successful completion, a value of 0 is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error. If `-1` is returned, no changes are made in the user ID and group ID of the file.

APPLICATION USAGE

Because `_POSIX_CHOWN_RESTRICTED` is always defined with a value other than `-1` on XSI-compliant systems, the error `EPERM` is always returned if the effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges.

On implementations which support symbolic links as directory entries rather than files, `lchown()` may fail.

SEE ALSO

`chgrp(1)`, `chown(1)`, `chmod(2)`, `symlink(2)`, `unistd(4)`, `types(5)`.

chroot(2)**NAME**

chroot - change root directory

SYNOPSIS

```
#include <unistd.h>
int chroot(const char *path);
```

DESCRIPTION

path points to a pathname naming a directory. `chroot()` causes the named directory to become the root directory, the starting point for path searches for pathnames beginning with `/`. The working directory of the process is unaffected by the `chroot()` system call.

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

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`chroot()` will fail and the root directory will remain unchanged if one or more of the following apply:

ELOOP

Too many symbolic links were encountered in translating *path*.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a *path* component exceeds `NAME_MAX`.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

EFAULT

path points outside the allocated address space of the process.

EINTR

A signal was caught during the `chroot()` system call.

EMULTIHOP

Components of *path* require hopping to multiple remote machines.

ENOLINK

path points to a remote machine and the link to that machine is no longer active.

ENOTDIR

Any component of the pathname is not a directory.

ENOENT

The named directory does not exist or is a null pathname.

EPERM

The effective user ID is not superuser.

RESULT

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

SEE ALSO

`chdir(2)`, `unistd(4)`.

close(2)**NAME**

`close` - close a file descriptor

SYNOPSIS

```
#include <unistd.h>
int close(int fdes);
```

DESCRIPTION

The `close()` function will deallocate the file descriptor indicated by *fdes*. To deallocate means to make the file descriptor available for return by subsequent calls to `open()` or other functions that allocate file descriptors. All outstanding record locks owned by the process on the file associated with the file descriptor will be removed (that is, unlocked).

If `close()` is interrupted by a signal that is to be caught, it will return `-1` with `errno` set to `EINTR` and the state of *fdes* is unspecified.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO will be discarded.

When all file descriptors associated with the open file description have been closed, the open file description is freed.

If the link count of the file is zero, when all file descriptors associated with the file have been closed, the memory space occupied by the file is freed and the file is no longer accessible.

If a STREAMS-based *fdes* is closed, and the calling process had previously registered to receive a `SIGPOLL` signal [see `signal(2)` and `sigset(2)`] for events associated with that stream [see `I_SETSIG` in `streamio(7)`], the calling process will be unregistered for events associated with the stream. The last `close()` for a stream causes the stream associated with *fdes* to be dismantled. If `O_NDELAY` and `O_NONBLOCK` are clear and there have been no signals posted for the stream, and if there are data on the module's write queue, `close()` waits up to 15 seconds (for each module and driver) for any output to drain before dismantling the stream. The time delay can be changed via an `I_SETCLTIME` `ioctl` request [see `streamio(7)`]. If `O_NDELAY` or `O_NONBLOCK` is set, or if there are any pending signals, `close()` does not wait for output to drain, and dismantles the stream immediately.

If *fdes* is associated with one end of a pipe, the last `close()` causes a "hangup" to occur on the other end of the pipe. In addition, if the other end of the pipe has been named [see `fattach(3C)`], the last `close()` forces the named end to be detached [see `fdetach(3C)`]. If the named end has no open processes associated with it and becomes detached, the stream associated with that end is also dismantled.

If *fdes* refers to the master side of a pseudo-terminal and this is the last `close`, a `SIGHUP` signal is sent to the process group, if any, for which the slave side of the pseudo-terminal is the controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal flushes all queued input and output.

If *fdes* refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message may be sent to the master.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

The `close()` function will fail if:

`EBADF`

*fil*des is not a valid open file descriptor.

EINTR

A signal was caught during the `close()` system call.

EWouldBlock

*fil*des is not STREAMS-based, is set blocking and a linger-timer expires before remaining data is sent and acknowledged.

The `close()` function may fail if:

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

ENOLINK

*fil*des is on a remote machine and the link to that machine is no longer active.

NOTES

An application that had used the standard I/O routine `fopen()` to open a file should use the corresponding `fclose()` routine rather than `close()`.

SEE ALSO

`creat(2)`, `dup(2)`, `exec(2)`, `fcntl(2)`, `open(2)`, `pipe(2)`, `signal(2)`, `fattach(3C)`, `fdetach(3C)`, `fclose(3S)`, `unistd(4)`, `signal(5)`.

NAME

`creat`, `creat64` - create a new file or rewrite an existing one

SYNOPSIS

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

int creat(const char *path, mode_t mode);
int creat64(const char *path, mode_t mode);
```

DESCRIPTION

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

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

If the file does not exist the file's owner ID is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process, or if the `S_ISGID` bit is set in the parent directory then the group ID of the file is inherited from the parent directory. The access permission bits of the file mode are set to the value of *mode* modified as follows:

- If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs, the `S_ISGID` bit is cleared.
- All bits set in the process' file mode creation mask are cleared [see `umask(2)`].
- The "save text image after execution bit" of the mode is cleared [see `chmod(2)`].

Upon successful completion, a write-only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across `exec` system calls [see `fcntl(2)`]. A new file may be created with a mode that forbids writing.

The call `creat(path, mode)` is equivalent to:

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

There is no functional difference between `creat()` and `creat64()`, except for the interpretation of `off64_t` [see `lfs(5)`].

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`creat()` fails if one or more of the following apply:

EACCES

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

EACCES

The file does not exist and the directory in which the file is to be created does not permit writing.

EACCES

The file exists and write permission is denied.

EAGAIN

The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see `chmod(2)`].

EFAULT

path points outside the allocated address space of the process.

EISDIR

The named file is an existing directory.

EINTR

A signal was caught during the `creat()` system call.

ELOOP

Too many symbolic links were encountered in translating *path*.

EMFILE

The process has too many open files [see `getrlimit(2)`].

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a *path* component exceeds `NAME_MAX`.

ENOTDIR

A component of the pathname is not a directory.

ENOENT

A component of the pathname does not exist.

ENOENT

The pathname is null.

EROFS

The named file resides or would reside on a read-only file system.

ETXTBSY

The file is a pure procedure (shared text) file that is being executed.

ENFILE

The system file table is full.

ENOSPC

The file system is out of inodes.

RESULT

Upon successful completion a non-negative integer, namely the lowest numbered unused file descriptor, is returned. Otherwise, a value of `-1` is returned, no files are created or modified, and `errno` is set to indicate the error.

SEE ALSO

`chmod(2)`, `close(2)`, `dup(2)`, `fcntl(2)`, `getrlimit(2)`, `lseek(2)`, `open(2)`, `read(2)`, `umask(2)`, `write(2)`, `fcntl(5)`, `lfs(5)`, `stat(5)`, `types(5)`.

NAME

`dup` - duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>
int dup(int fdes);
```

DESCRIPTION

fdes is a file descriptor obtained from a `creat()`, `open()`, `dup()`, `fcntl()`, or `pipe()` system call. `dup()` returns a new file descriptor having the following in common with the original file descriptor.

- Same open file (or pipe).
- Same file pointer (i.e., both file descriptors share one file pointer).
- Same access mode (read, write or read/write).

The new file descriptor is set to remain open across `exec` system calls [see [fcntl\(2\)](#)].

The file descriptor returned is the smallest number available.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`dup()` will fail if one or more of the following apply:

EBADF

fdes is not a valid open file descriptor.

EINTR

A signal was caught during the `dup()` system call.

EMFILE

The process has too many open files [see [getrlimit\(2\)](#)].

ENOLINK

fdes is on a remote machine and the link to that machine is no longer active.

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

SEE ALSO

[close\(2\)](#), [creat\(2\)](#), [exec\(2\)](#), [fcntl\(2\)](#), [getrlimit\(2\)](#), [open\(2\)](#), [pipe\(2\)](#), [dup2\(3C\)](#), [lockf\(3C\)](#), [unistd\(4\)](#).

NAME

exec:
 execl, execv, execl, execve, execlp, execvp, environ - **execute a file**

SYNOPSIS

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg0, ...,
          const char *argn, (char *)0);
int execv(const char *path, char *const *argv);
int execl(const char *path, const char *arg0, ...,
          const char *argn, (char *)0, const char *envp[]);
int execve(const char *path, char *const *argv,
          char *const *envp);
int execlp(const char *file, const char *arg0, ...,
          const char *argn, (char *)0);
int execvp(const char *file, char *const *argv);
```

DESCRIPTION

exec in all its forms overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful **exec** because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

```
#! pathname [arg]
```

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is **exec'd**, the system **execs** the specified interpreter. The *pathname* specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally **exec'd** file.

When a C program is executed, it is called as follows:

```
int main(int argc, char *argv[], char *envp[]);
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As indicated, *argc* is at least one, and the first member of the array points to a string containing the name of the file.

In addition, the following variable:

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

is initialised as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array is not counted in *argc*. Conforming multi-threaded applications will not use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the *environ* variable to access that environment variable.

The arguments specified by a program with one of the **exec** functions are passed on to the new process image in the corresponding `main()` arguments.

path points to a *pathname* that identifies the new process file.

file points to the new process file. If *file* contains a slash character, this argument is used as a pathname for this file. If *file* does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed in the `PATH` environment variable [see `environ(5)`]. The environment is supplied typically by the shell [see `sh(1)`].

If the new process file is not an executable object file, `execlp()` and `execvp()` use the contents of that file as standard input to `sh(1)`.

If the process image file is not a valid executable object, `execlp()` and `execvp()` use the contents of that file as standard input to a command interpreter conforming to `system()`. In this case, the command interpreter becomes the new process image.

The arguments *arg0*, ..., *argn* point to null-terminated character strings. These strings constitute the argument list available to the new process image. Minimally, *arg0* must be present. It will become the name of the process, as displayed by the `ps` command. *arg0* points to a string that is the same as *path* (or the last component of *path*). The list of argument strings is terminated by a `(char *)0` argument.

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process image. *argv* must have at least one member, and it should point to a string that is the same as *path* (or its last component). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. *envp* is terminated by a null pointer. For `execl()`, `execv()`, `execvp()`, and `execlp()`, the C run-time start-off routine places a pointer to the environment of the calling process in the global object `extern char **environ`, and it is used to pass the environment of the calling process to the new process.

The number of bytes available for the new process' combined argument and environment lists is `ARG_MAX`. It is implementation-dependent whether null terminators, pointers, and/or any alignment bytes are included in this total.

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

Directory streams open in the calling process image are closed in the new process image.

The state of conversion descriptors and message catalogue descriptors in the new process image is undefined. For the new process, the equivalent of:

```
setlocale(LC_ALL, "C")
```

is executed at startup.

Signals set to the default action (`SIG_DFL`) in the calling process image are set to the default action in the new process image. Signals set to be ignored (`SIG_IGN`) by the calling process image are set to be ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image [see `signal(5)`]. After a successful call to any of the `exec` functions, alternate signal stacks are not preserved and the `SA_ONSTACK` flag is cleared for all signals.

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

If the `ST_NOSUID` bit is set for the file system containing the new process image file, then the effective user ID, effective group ID, saved set-user-ID and saved set-group-ID are unchanged in the new process image. Otherwise, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the new process image is set to the user ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved set-user-ID and the saved set-group-ID for use by `setuid()`).

Any shared memory segments attached to the calling process image will not be attached to the new process image.

Any mappings established through `mmap()` are not preserved across an `exec`.

If the effective user-ID is `root` or superuser, the set-user-ID and setgroup-ID bits will be honored when the process is being controlled by `ptrace()`.

The shared memory segments attached to the calling process will not be attached to the new process [see [shmop\(2\)](#)].

Any mappings established through `mmap()` are not preserved across an `exec`.

Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.

Profiling is disabled for the new process [see [profil\(2\)](#)].

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

- nice value [see [nice\(2\)](#)]
- scheduler class and priority [see [priocntl\(2\)](#)]
- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- semadj values [see [semop\(2\)](#)]
- session ID [see [exit\(2\)](#) and [signal\(2\)](#)]
- trace flag [see [ptrace\(2\)](#), request 0]
- time left until an alarm clock signal [see [alarm\(2\)](#)]
- current working directory
- root directory
- file mode creation mask [see [umask\(2\)](#)]
- file size limit [see [ulimit\(2\)](#) and [getrlimit\(2\)](#)]
- resource limits [see [getrlimit\(2\)](#)]
- `tms_ftime()`, `tms_stime()`, `tms_cutime()`, and `tms_cstime` [see [times\(2\)](#)]
- file-locks [see [fcntl\(2\)](#) and [lockf\(3C\)](#)]
- controlling terminal
- process signal mask [see [sigprocmask\(2\)](#)]
- pending signals [see [sigpending\(2\)](#)]

All other process attributes will be the same in the new and old process images. The inheritance of process attributes not defined by this specification is implementation-dependent.

A call to any `exec` function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions will be called.

Upon successful completion, the `exec` functions mark for update the `st_atime` field of the file. If an `exec` function failed but was able to locate the process image file, whether the `st_atime` field is marked for update is unspecified. Should the `exec` function succeed, the process image file is considered to have been opened with `open()`. The corresponding `close()` is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the `exec` functions. The `argv[]` and `envp[]` arrays of pointers and the strings to which those arrays point will not be modified by a call to one of the `exec` functions, except as a consequence of replacing the process image.

The saved resource limits in the new process image are set to be a copy of the process's corresponding hard

and soft limits.

RETURN VALUE

If one of the `exec` functions returns to the calling process image, an error has occurred; the return value is `-1`, and `errno` is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

The `exec` functions will fail if:

E2BIG

The number of bytes in the new process' argument list is greater than the system-imposed limit of 20480 bytes. The argument list limit is the sum of the size of the argument list plus the size of the environment's exported shell variables.

EACCES

Search permission is denied for a directory listed in the new process file's path prefix.

EACCES

The new process file is not an ordinary file.

EACCES

The new process file mode denies execution permission.

EAGAIN

Total amount of system memory available when reading via raw I/O is temporarily insufficient.

EFAULT

Required hardware is not present.

EFAULT

An `a.out` that was compiled with the MAU or 32B flag is running on a machine without a MAU or 32B.

EFAULT

An argument points to an illegal address.

EINTR

A signal was caught during the `exec` system call.

ELIBACC

Required shared library does not have execute permission.

ELIBEXEC

Trying to `exec` a shared library directly.

ELOOP

Too many symbolic links were encountered in translating *path* or *file*.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and the file system type does not allow it.

ENAMETOOLONG

The length of the *file* or *path* arguments, or an element of the environment variable `PATH` prefixed to a file, exceeds `PATH_MAX`, or the length of a *file* or *path* component exceeds `NAME_MAX`.

ENOENT

One or more components of the new process pathname of the file do not exist or is a null pathname.

ENOTDIR

A component of the new process path of the file prefix is not a directory.

The `exec` functions, except for `execlp()` and `execvp()`, will fail if:

ENOEXEC

The `exec` is not an `execlp()` or `execvp()`, and the new process file has the appropriate access permission but is not in the proper format.

The `exec` functions may fail if:

`ETXTBSY`

The new process file is a pure procedure (shared text) file that is currently open for writing by some process.

`ENOMEM`

The new process requires more memory than is allowed by the system-imposed maximum `MAXMEM`.

`ENOLINK`

path points to a remote machine and the link to that machine is no longer active.

NOTES

As the state of conversion descriptors and message catalogue descriptors in the new process image is undefined, portable applications should not rely on their use and should close them prior to calling one of the `exec` functions.

If you do not use the default C locale in your application, you must call `setlocale(3C)` with the appropriate arguments to set the locale of the new process.

The `environ` array should not be accessed directly by the application.

SEE ALSO

`ps(1)`, `sh(1)`, `alarm(2)`, `exit(2)`, `fcntl(2)`, `fork(2)`, `getrlimit(2)`, `nice(2)`, `prctl(2)`, `ptrace(2)`, `semop(2)`, `signal(2)`, `sigpending(2)`, `sigprocmask(2)`, `times(2)`, `umask(2)`, `atexit(3C)`, `lockf(3C)`, `system(3S)`, `a.out(4)`, `unistd(4)`, `environ(5)`, `lfs(5)`.

NAME

`exit`, `_exit` - terminate process

SYNOPSIS

```
#include <stdlib.h>
void exit(int status);
#include <unistd.h>
void _exit(int status);
```

DESCRIPTION

The `exit()` function calls all functions registered by the `atexit()` function, in the reverse order of their registration.

If a function registered by a call to `atexit()` fails to return, the remaining registered functions are not called and the rest of the `exit` processing is not completed. If the `exit()` function is called more than once, the results are undefined.

The `exit()` function then flushes all output streams, closes all open streams, and removes all files created by `tmpfile()`. Finally, control is returned to the host environment as described below. The values of `status` can be `EXIT_SUCCESS` or `EXIT_FAILURE`, as described in `<stdlib.h>`, or any implementation-dependent value, although note that only the range 0 through 255 will be available to a waiting parent process.

The `_exit()` function terminates the process without these cleanup measures.

Once all registered functions have been processed, `exit()` flushes all output streams, closes all open streams, and removes all files created by `tmpfile()`.

`exit()` and `_exit()` terminate the calling process with the following consequences:

- All of the file descriptors, directory streams, conversion descriptors and message catalog descriptors open in the calling process are closed.
- If the parent process of the calling process is executing a `wait()`, `wait3()`, `waitid()` or `waitpid()`, and has neither set its `SA_NOCLDWAIT` flag nor set `SIGCHLD` to `SIG_IGN`, it is notified of the calling process' termination and the low-order eight bits (that is, bits 0377) of `status` are made available to it. If the parent is not waiting, the child's status will be made available to it when the parent subsequently executes `wait()`, `wait3()`, `waitid()` or `waitpid()`.
- If the parent process of the calling process is not executing a `wait()`, `wait3()`, `waitid()` or `waitpid()`, and has not set its `SA_NOCLDWAIT` flag, or set `SIGCHLD` to `SIG_IGN`, the calling process is transformed into a *zombie process*. A *zombie process* is an inactive process and it will be deleted at some later time when its parent process executes `wait()`, `wait3()`, `waitid()` or `waitpid()`.
- Termination of a process does not directly terminate its children. The sending of a `SIGHUP` signal as described below indirectly terminates children in some circumstances.
- If the implementation supports the `SIGCHLD` signal, a `SIGCHLD` will be sent to the parent process.
- If the parent process has set its `SA_NOCLDWAIT` flag, or set `SIGCHLD` to `SIG_IGN`, the status will be discarded, and the lifetime of the calling process will end immediately. If `SA_NOCLDWAIT` is set, it is implementation-dependent whether a `SIGCHLD` signal will be sent to the parent process.
- The parent process ID of all of the calling process' existing child processes and zombie processes is set to 1. This means the initialization process inherits each of these processes.
- Each attached shared-memory segment is detached and the value of `shm_nattch` [see [shmget\(2\)](#)] in the data structure associated with its shared memory ID is decremented by 1.
- For each semaphore for which the calling process has set a `semadj` value [see [semop\(2\)](#)], that `semadj` value is added to the `semval` of the specified semaphore.

- If the process is a controlling process, `SIGHUP` is sent to the foreground process group of its controlling terminal and its controlling terminal is deallocated.
- If the process is a controlling process, the controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.
- If the calling process has any stopped children whose process group will be orphaned when the calling process exits, or if the calling process is a member of a process group that will be orphaned when the calling process exits, that process group will be sent `SIGHUP` and `SIGCONT` signals.
- Memory mappings created in the process are unmapped before the process is destroyed.
- Threads terminated by a call to `_exit()` will not invoke their cancellation cleanup handlers or per-thread data destructors.

RETURN VALUE

These functions do not return.

ERRORS

No errors are defined.

NOTES

See **NOTES** section under `signal(2)`.

SEE ALSO

`close(2)`, `semop(2)`, `shmget(2)`, `sigaction(2)`, `signal(2)`, `times(2)`, `wait(2)`, `waitid(2)`, `atexit(3C)`, `wait3(3C)`, `unistd(4)`, `stdlib(5)`.

NAME

fcntl - control an open file

SYNOPSIS

```
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
int fcntl(int fd, int cmd, ... /* arg */);
```

DESCRIPTION

fcntl() provides for control over open files. *fd* is a file descriptor of an open file.

fcntl() may take a third argument, *arg*, whose data type, value and use depend upon the value of *cmd*. *cmd* specifies the operation to be performed by fcntl() and may be one of the following:

F_DUPFD

Returns a new file descriptor with the following characteristics:

- Lowest numbered available file descriptor greater than or equal to the integer value given as the third argument.
- Same open file (or pipe) as the original file.
- Same file pointer as the original file (i.e., both file descriptors share one file pointer).
- Same access mode (read, write, or read/write) as the original file.
- Shares any locks associated with the original file descriptor.
- The close-on-exec flag FD_CLOEXEC (see F_GETFD) associated with the new file descriptor is set to remain open across exec(2) system calls.

F_GETFD

Gets the close-on-exec FD_CLOEXEC flag associated with file descriptor *fd*. If the low-order bit is 0, the file will remain open across exec. Otherwise, the file will be closed upon execution of exec.

F_SETFD

Set the file descriptor flags defined in <fcntl.h> [see fcntl(5)], that are associated with *fd*, to the third argument, *arg*, taken as type int. If the FD_CLOEXEC flag in the third argument is 0, the file will remain open across the exec functions; otherwise the file will be closed upon successful execution of one of the exec functions.

F_GETFL

Get the file status flags and file access modes, defined in <fcntl.h> [see fcntl(5)], for the file description associated with *fd*. The file access modes can be extracted from the return value using the mask O_ACCMODE, which is defined in <fcntl.h>. File status flags and file access modes are associated with the file description and do not affect other file descriptors that refer to the same file with different open file descriptions.

F_SETFL

Set the file status flags, defined in <fcntl.h> [see fcntl(5)], for the file description associated with *fd* from the corresponding bits in the third argument, *arg*, taken as type int. Bits corresponding to the file access mode and the *oflag* values that are set in *arg* are ignored. If any bits in *arg* other than those mentioned here are changed by the application, the result is unspecified.

F_FREESP

Frees storage space associated with a section of the ordinary file *fd*. The section is specified by a variable of data type struct flock pointed to by the third argument *arg*. The data type struct flock is defined in the <fcntl.h> header file [see fcntl(5)] and contains the following members: l_whence is 0, 1, or 2 to indicate that the relative offset l_start will be measured from the start of

the file, the current position, or the end of the file, respectively. `l_start` is the offset from the position specified in `l_whence`. `l_len` is the size of the section. An `l_len` of 0 frees up to the end of the file; in this case, the end of file (i.e., file size) is set to the beginning of the section freed. Any data previously written into this section is no longer accessible.

The following commands are used for record-locking. Locks may be placed on an entire file or on segments of a file.

`F_SETLK`

Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to type `struct flock`, defined in `<fcntl.h>` [see [fcntl\(5\)](#)]. `F_SETLK` is used to establish shared (or read) locks (`F_RDLCK`) or exclusive (or write) locks (`F_WRLCK`), as well as to remove either type of lock (`F_UNLCK`). `F_RDLCK`, `F_WRLCK` and `F_UNLCK` are defined in `<fcntl.h>`. If a shared or exclusive lock cannot be set, `fcntl()` will return immediately with a return value of `-1`.

`F_SETLKW`

This command is the same as `F_SETLK` except that if a shared or exclusive lock is blocked by other locks, the thread will wait until the request can be satisfied. If a signal that is to be caught is received while `fcntl()` is waiting for a region, `fcntl()` will be interrupted. Upon return from the signal handler, `fcntl()` will return `-1` with `errno` set to `EINTR`, and the lock operation will not be done.

`F_GETLK`

If the lock request described by the `flock` structure that *arg* points to could be created, then the structure is passed back unchanged except that the lock type is set to `F_UNLCK` and the `l_whence` field will be set to `SEEK_SET`. If a lock is found that would prevent this lock from being created, then the structure is overwritten with a description of the first lock that is preventing such a lock from being created. The structure also contains the process ID and the system ID of the process holding the lock.

This command never creates a lock; it tests whether a particular lock could be created.

`F_RSETLK`

Used by the network lock daemon, `lockd`, to lock NFS files using the NFS server kernel.

`F_RSETLKW`

Used by the network lock daemon, `lockd`, to lock NFS files using the NFS server kernel.

`F_RGETLK`

Used by the network lock daemon, `lockd`, to lock NFS files using the NFS server kernel.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock and no read locks may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The `flock` structure describes the type (`l_type`), starting offset (`l_whence`), relative offset (`l_start`), size (`l_len`), process ID (`l_pid`), and system ID (`l_sysid`) of the segment of the file to be affected.

The value of `l_whence` is `SEEK_SET`, `SEEK_CUR` or `SEEK_END`, to indicate that the relative offset `l_start` bytes will be measured from the start of the file, current position or end of the file, respectively. The value of `l_len` is the number of consecutive bytes to be locked. The value of `l_len` may be negative (where the definition of `off_t` permits negative values of `l_len`). The `l_pid` field is only used with `F_GETLK` to return the process ID of the process holding a blocking lock. After a successful `F_GETLK` request, that is, one in which a lock was found, the value of `l_whence` will be `SEEK_SET`.

If `l_len` is positive, the area affected starts at `l_start` and ends at `l_start + l_len - 1`. If `l_len` is negative, the area affected starts at `l_start + l_len` and ends at `l_start - 1`. Locks may start and extend beyond the current end of a file, but must not be negative relative to the beginning of the file. A lock will be set to extend to the largest possible value of the file offset for that file by setting `l_len` to 0. If such a lock also has `l_start` set to 0 and `l_whence` is set to `SEEK_SET`, the whole file will be locked.

There will be at most one type of lock set for each byte in the file. Before a successful return from an `F_SETLK` or an `F_SETLKW` request when the calling process has previously existing locks on bytes in the

region specified by the request, the previous lock type for each byte in the specified region will be replaced by the new lock type. As specified above under the descriptions of shared locks an exclusive locks, an `F_SETLK` or an `F_SETLKW` request will (respectively) fail or block when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process created using the `fork()` function.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process' locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, the `fcntl()` function will fail with an `EDEADLK` error.

When mandatory file and record locking is active on a file [see `chmod(2)`], `open(2)`, `read(2)` and `write(2)` system calls issued on the file will be affected by the record locks in effect. Mandatory locking is not supported with NFS.

An unlock (`F_UNLCK`) request in which `l_len` is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type `off_t`, when the process has an existing lock in which `l_len` is 0 and which includes the last byte of the requested segment, will be treated as a request to unlock from the start of the requested segment with an `l_len` equal to 0. Otherwise an unlock (`F_UNLCK`) request will attempt to unlock only the requested segment.

The following additional value may be used in constructing *oflag*:

`O_LARGEFILE`

If set, the offset maximum in the open file description will be the largest value that can be represented correctly in an object of type `off64_t`.

The `O_LARGEFILE` flag may be set or cleared with `F_SETFL`. An incorrectly written program may inadvertently clear this flag. For example, some programs put a file into append mode with:

```
fcntl(fd, F_SETFL, O_APPEND);
```

This is incorrect because it turns off all the other open flags, including `O_LARGEFILE`. Instead, to turn on append mode one should first use `F_GETFL` to get the current flags:

```
int oflag = fcntl(fd, F_GETFL, 0);
```

then include `O_APPEND` in the flags:

```
oflag |= O_APPEND;
```

and then set the new flags:

```
fcntl(fd, F_SETFL, oflag);
```

A more complete example would also check for `fcntl()` failures.

The behavior of the following additional values is equivalent to the corresponding Single UNIX Specification (SUS) value (`FGETLK`, `FSETLK`, `FSETLKW`), but they take a `struct flock64` argument rather than a `struct flock` argument.

`FGETLK64`

`FSETLK64`

`FSETLKW64`

RETURN VALUE

On successful completion, `fcntl()` returns a value that depends on *cmd*, as follows:

`F_DUPFD`

A new file descriptor.

`F_GETFD`

Value of "close-on-exec" status flag (only the low-order bit is defined).

`F_SETFD`

Value other than -1.

`F_FREESP`

Value of 0.

`F_GETFL`

Value of file status flags. The result will not be negative.

`F_SETFL`

Value other than -1.

`F_GETLK`

Value other than -1.

`F_SETLK`

Value other than -1.

`F_SETLKW`

Value other than -1.

On failure, `fcntl()` returns -1 and sets `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `fcntl()` function will fail if:

EACCES or EAGAIN

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

EBADF

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

EFAULT

cmd is `F_FREESP` and the value pointed to by the third argument *arg* resulted in an address outside the process' allocated address space.

EFAULT

cmd is `F_GETLK`, `F_SETLK` or `F_SETLKW` and the value pointed to by the *arg* argument resulted in an address outside the program address space.

EINTR

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

EINVAL

The *cmd* argument is invalid, or the *cmd* argument is `F_DUPFD` and *arg* is negative or greater than or equal to `OPEN_MAX`, or the *cmd* argument is `F_GETLK`, `F_SETLK` or `F_SETLKW` and the data pointed to by *arg* is not valid, or *fildev* refers to a file that does not support locking.

EINVAL

cmd is not a valid value.

EIO

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

EMFILE

The argument *cmd* is `F_DUPFD` and `OPEN_MAX` file descriptors are currently open in the calling process, or no file descriptors greater than or equal to *arg* are available.

ENOLCK

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

ENOLINK

fildev is on a remote machine and the link to that machine is no longer active.

ENOLINK

cmd is `F_FREESP`, the file is on a remote machine, and the link to that machine is no longer active.

EOVERFLOW

One of the values to be returned cannot be represented correctly.

EOVERFLOW

The *cmd* argument is `F_GETLK`, `F_SETLK` or `F_SETLKW` and the smallest or, if *l_len* is non-zero, the largest, offset of any byte in the requested segment cannot be represented correctly in an object of type `off_t`.

The `fcntl()` function may fail if:

EDEADLK

The *cmd* argument is `F_SETLKW`, the lock is blocked by some lock from another process and putting the calling process to sleep, waiting for that lock to become free would cause a deadlock.

EDEADLK

cmd is `F_FREESP`, mandatory record locking is enabled, `O_NDELAY` and `O_NONBLOCK` are clear and a deadlock condition was detected.

SEE ALSO

`close(2)`, `creat(2)`, `dup(2)`, `exec(2)`, `fork(2)`, `open(2)`, `pipe(2)`, `unistd(4)`, `fcntl(5)`, `lfs(5)`, `types(5)`.

NAME

`fork` - create a new process

SYNOPSIS

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

DESCRIPTION

`fork()` causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

- real user ID, real group ID, effective user ID, effective group ID environment
- "close-on-exec" flag [see `exec(2)`]
- signal handling settings (`SIG_DFL`, `SIG_IGN`, `SIG_HOLD`, function address)
- supplementary group IDs
- set-user-ID mode bit
- set-group-ID mode bit
- profiling on/off status
- nice value [see `nice(2)`]
- scheduler class [see `prctl(2)`]
- all attached shared memory segments [see `shmop(2)`]
- process group ID
- session ID [see `exit(2)`]
- current working directory
- root directory
- file mode creation mask [see `umask(2)`]
- resource limits

A 32-bit process may inherit the 64-bit resources of the parent process [see `getrlimit(2)` and the explanations regarding `RLIM_SAVED_CUR` and `RLIM_SAVED_MAX` as well as `ulimit(2)`].

- controlling terminal

Scheduling priority and any per-process scheduling parameters that are specific to a given scheduling class may be inherited, according to the policy of that particular class [see `prctl(2)`].

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

- The child process has a unique process ID which does not match any active process group ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors and directory streams. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.
- All `semadj` values are cleared [see `semop(2)`].
- Process locks, text locks and data locks are not inherited by the child [see `plock(2)`].
- The child process may have its own copy of the parent's message catalogue descriptors.
- The child process' `tms` structure is cleared: `tms_utime`, `stime`, `cutime`, and `cstime` are set to 0 [see `times(2)`].

- The time left until an alarm clock signal is reset to 0.
- The set of signals pending for the child process is initialized to the empty set.
- The child process may have its own copy of the parent's message catalogue descriptors.
- Interval timers are reset in the child process.
- Memory mappings created in the parent are retained in the child process. `MAP_PRIVATE` mappings inherited from the parent will also be `MAP_PRIVATE` mappings in the child, and any modifications to the data in these mappings made by the parent prior to calling `fork()` will be visible to the child. Any modifications to the data in `MAP_PRIVATE` mappings made by the parent after `fork()` returns will be visible only to the parent. Modifications to the data in `MAP_PRIVATE` mappings made by the child will be visible only to the child.

Record locks set by the parent process are not inherited by the child process [see `fcntl(2)`].

A process is created with a single thread. If a multi-threaded process calls `fork()`, the new process contains a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal safe operations until such time as one of the `exec` functions is called.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`fork()` will fail and no child process will be created if one or more of the following apply:

`EAGAIN`

The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user `CHILD_MAX` would be exceeded.

`EAGAIN`

Total amount of system memory available when reading via raw I/O is temporarily insufficient.

The `fork()` function may fail if:

`ENOMEM`

The swap space is too small.

SEE ALSO

`alarm(2)`, `exec(2)`, `fcntl(2)`, `getrlimit(2)`, `nice(2)`, `plock(2)`, `prctl(2)`, `ptrace(2)`, `semop(2)`, `shmop(2)`, `signal(2)`, `times(2)`, `ulimit(2)`, `umask(2)`, `wait(2)`, `system(3S)`, `unistd(4)`, `lfs(5)`, `types(5)`.

NAME

fpathconf, pathconf - get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>
long int fpathconf(int fildes, int name);
long int pathconf(const char *path, int name);
```

DESCRIPTION

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

For `pathconf()`, the *path* argument points to the pathname of a file or directory.

For `fpathconf()`, the *fildes* argument is an open file descriptor.

The *name* argument represents the variable to be queried relative to that file or directory. Implementations will support all of the variables listed in the following table and may support others. The variables in the following table come from `limits.h` or `unistd.h` and the symbolic constants, defined in `unistd.h`, are the corresponding values used for *name*:

Variable	Value of <i>name</i>	See Notes
FILESIZEBITS	_PC_FILESIZEBITS	3, 4
LINK_MAX	_PC_LINK_MAX	1
MAX_CANON	_PC_MAX_CANON	2
MAX_INPUT	_PC_MAX_INPUT	2
NAME_MAX	_PC_NAME_MAX	3, 4
PATH_MAX	_PC_PATH_MAX	4, 5
PIPE_BUF	_PC_PIPE_BUF	6
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3, 4
_POSIX_VDISABLE	_PC_VDISABLE	2

Notes:

- 1) If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.
- 2) The behavior is undefined if *path* or *fildes* does not refer to a terminal file.
- 3) If *path* or *fildes* refers to a directory, the value returned applies to the filenames within the directory.
- 4) The behavior is undefined if *path* or *fildes* does not refer to a directory.
- 5) If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative pathname when the specified directory is the working directory.
- 6) If *path* or *fildes* refers to a pipe or FIFO, the value returned applies to the FIFO itself. If *path* or *fildes* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *path* or *fildes* refer to any other type of file, the behavior is undefined.
- 7) If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.

The value of the configurable system limit or option specified by *name* does not change during the lifetime of the calling process.

RETURN VALUE

If *name* is an invalid value, both `pathconf()` and `fpathconf()` return `-1` and `errno` is set to indicate the error.

If the variable corresponding to *name* has no limit for the *path* or file descriptor, both `pathconf()` and `fpathconf()` return `-1` without changing `errno`. If the implementation needs to use *path* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *path*, or if the process did not have appropriate privileges to query the file specified by *path*, or *path* does not exist, `pathconf()` returns `-1` and `errno` is set to indicate the error.

If the implementation needs to use *fildev* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *fildev*, or if *fildev* is an invalid file descriptor, `fpathconf()` will return `-1` and `errno` is set to indicate the error.

Otherwise `pathconf()` or `fpathconf()` returns the current variable value for the file or directory without changing `errno`. The value returned will not be more restrictive than the corresponding value available to the application when it was compiled with the implementation's `limits.h` or `unistd.h`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`fpathconf()` will fail if the following is true:

EINVAL

name is an invalid value.

`fpathconf()` may fail if the following is true:

EBADF

fildev is not a valid file descriptor.

EINVAL

The implementation does not support an association of the variable *name* with the specified file.

`pathconf()` will fail if the following is true:

EINVAL

name is an invalid value.

ELOOP

Too many symbolic links are encountered while translating *path*.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and the file system type does not allow it.

ENOSYS

An attempt was made to perform an unsupported operation (such as create, remove, link, or unlink) on an entry in a file system.

`pathconf()` may fail if the following is true:

EACCES

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

EINVAL

The implementation does not support an association of the variable *name* with the specified file.

ENAMETOOLONG

The length of a pathname 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`.

ENOENT

path is needed for the command specified and the named file does not exist or the *path* argument points to an empty string.

ENOLINK

path points to a remote machine and the link to that machine is no longer active.

ENOTDIR

A component of the path prefix is not a directory.

NOTES

The `_PC_FILESIZEBITS` option makes it possible for a process to determine how large a file can be created in a given directory. It takes into account implementation limitations in the file system (e.g. due to the size of file size and block count variables), and it takes into account long term policy limitations. It does not take into account dynamic restrictions such as the `RLIM_FSIZE` resource limit or the number of available file blocks, so the process must perform appropriate checks.

When the current directory is on a typical large file capable file system of 4 GB in size,

```
pathconf(".", _PC_FILESIZEBITS);
```

will return 36. This means that in the case of a 4-GB file system (`vxfs`, `ufs`), a file can be written up to position $2^{36}-1 = 32\text{ GB}-1$ (physical block configuration max. 4 GB).

The maximum file size in a `vxfs` file system (Veritas) is $2^{44}-b_{size}-1$, and in a `ufs` file system is $2^{45}-b_{size}-1$.

In general, if the maximum size file that could ever exist on the mounted file system is *maxsize* then the returned value is 2 plus the floor of the base 2 logarithm of *maxsize*.

SEE ALSO

`sysconf(3C)`, `limits(4)`, `unistd(4)`, `lfs(5)`.

NAME

`fsync` - synchronize file status

SYNOPSIS

```
#include <unistd.h>
int fsync(int fildes);
```

DESCRIPTION

`fsync()` moves all modified data and attributes of *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. `fsync()` is different from `sync`, which schedules disk I/O for all files but returns before the I/O completes. `fsync()` should be used by programs that require modifications to a file to be completed before continuing. For example, a program 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.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`fsync()` fails if one or more of the following apply:

EBADF

fildes is not a valid file descriptor open for writing.

ENOLINK

fildes is on a remote machine and the link to that machine is no longer active.

EINTR

A signal was caught during execution of the `fsync()` system call.

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

In the event that any of the queued I/O operations fail, `fsync()` returns the error conditions defined for [read\(2\)](#) and [write\(2\)](#).

NOTES

The way the data reach the physical medium depends on both implementation and hardware. `fsync()` returns when the device driver tells it that the write has taken place.

SEE ALSO

[sync\(2\)](#), [unistd\(4\)](#).

NAME

getcontext, setcontext - amend user context

SYNOPSIS

```
#include <ucontext.h>
int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);
```

DESCRIPTION

These functions, along with those defined in [makecontext\(3C\)](#), are useful for implementing user level context switching between multiple threads of control within a process.

The `getcontext()` function initialises the structure pointed to by `ucp` to the current user context of the calling thread. The `ucontext_t` type that `ucp` points to defines the user context and includes the contents of the calling thread's machine registers, the signal mask, and the current execution stack.

The `setcontext()` function restores the user context pointed to by `ucp`. A successful call to `setcontext()` does not return; program execution resumes at the point specified by the `ucp` argument passed to `setcontext()`. The `ucp` argument should be created either by a prior call to `getcontext()` or `makecontext()`, or by being passed as an argument to a signal handler. If the `ucp` argument was created with `getcontext()`, program execution continues as if the corresponding call of `getcontext` had just returned. If the `ucp` argument was created with `makecontext()`, program execution continues with the function passed to `makecontext()`. When that function returns, the thread continues as if after a call to `setcontext()` with the `ucp` argument that was input to `makecontext()`. If the `uc_link` member of the `ucontext_t` structure pointed to by the `ucp` argument is equal to 0, then this context is the main context, and the thread will exit when this context returns. The effects of passing a `ucp` argument obtained from any other source are unspecified.

RETURN VALUE

On successful completion, `setcontext()` does not return and `getcontext()` returns 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

No errors are defined.

NOTES

When a signal handler is executed, the current user context is saved and a new context is created by the kernel. If the process leaves the signal handler via [longjmp\(3C\)](#) the original context will not be restored, and future calls to `getcontext()` will not be reliable. Signal handlers should use [siglongjmp\(3C\)](#) or `setcontext()` instead.

Portable applications should not modify or access the `uc_mcontext` member of `ucontext_t`. A portable application cannot assume that context includes any process-wide static data, possibly including `errno`. Users manipulating contexts should take care to handle these explicitly when required.

If `getcontext()` and `setcontext()` are used in the same way as [setjmp\(3C\)](#) and [longjmp\(3C\)](#), it is important that the stack area backed up by `getcontext()` and restored by `setcontext()` is not used unintentionally by other functions in between these two calls. In this case, `getcontext()` and `setcontext()` must be called from the same higher-level function.

SEE ALSO

[sigaction\(2\)](#), [sigaltstack\(2\)](#), [sigprocmask\(2\)](#), [makecontext\(3C\)](#), [setjmp\(3C\)](#), [ucontext\(5\)](#).

getdents(2)**NAME**

`getdents` - convert directory entries

SYNOPSIS

```
#include <sys/dirent.h>
int getdents (int fildev, struct dirent *buf, size_t
nbyte) ;
```

DESCRIPTION

fildev is a file descriptor obtained from an `open(2)` or `dup(2)` system call.

`getdents()` attempts to read *nbyte* bytes from the directory associated with *fildev* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be significantly less than *nbyte*. See `dirent(4)` to calculate the number of bytes.

The file system independent directory entry is specified by the `dirent` structure. For a description of this see `dirent(4)`.

On devices capable of seeking, `getdents()` starts at a position in the file given by the file pointer associated with *fildev*. Upon return from `getdents()`, the file pointer is incremented to point to the next directory entry.

This system call was developed in order to implement the `readdir()` routine [for a description, see `directory(3C)`], and should not be used for other purposes.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`getdents()` will fail if one or more of the following apply:

EBADF

fildev is not a valid file descriptor open for reading.

EFAULT

buf points outside the allocated address space.

EINVAL

nbyte is not large enough for one directory entry.

ENOENT

The current file pointer for the directory is not located at a valid entry.

ENOLINK

fildev points to a remote machine and the link to that machine is no longer active.

ENOTDIR

fildev is not a directory.

EIO An I/O error occurred while accessing the file system.

RETURN VALUE

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached. If the system call failed, a -1 is returned and `errno` is set to indicate the error.

SEE ALSO

`directory(3C)`, `dirent(4)`.

getgroups(2)**NAME**

`getgroups`, `setgroups` - get/set group IDs

SYNOPSIS

```
#include <unistd.h>
int getgroups(int gidsetsize, gid_t grouplist[]);
int setgroups(int ngroups, const gid_t grouplist[]);
```

DESCRIPTION

`getgroups()` gets the current supplementary group access list of the calling process and stores the result in the array of group IDs specified by *grouplist*. This array has *gidsetsize* entries and must be large enough to contain the entire list. This list cannot be greater than `NGROUPS_MAX`. If *gidsetsize* equals 0, `getgroups()` will return the number of groups to which the calling process belongs without modifying the array pointed to by *grouplist*.

It is unspecified whether the effective group ID of the calling process is included in, or omitted from, the returned list of supplementary group IDs.

`setgroups()` sets the supplementary group access list of the calling process from the array of group IDs specified by *grouplist*. The number of entries is specified by *ngroups* and can not be greater than `NGROUPS_MAX`. This function may be invoked only by the superuser.

RETURN VALUE

Upon successful completion, `getgroups()` returns the number of supplementary group IDs set for the calling process and `setgroups()` returns the value 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`getgroups()` will fail if:

EINVAL

The value of *gidsetsize* is non-zero and less than the number of supplementary group IDs set for the calling process.

`setgroups()` will fail if:

EINVAL

The value of *ngroups* is greater than `NGROUPS_MAX`.

EPERM

The effective user ID is not superuser.

Either call will fail if:

EFAULT

A referenced part of the array pointed to by *grouplist* is outside of the allocated address space of the process.

SEE ALSO

[groups\(1\)](#), [chown\(2\)](#), [getuid\(2\)](#), [setuid\(2\)](#), [initgroups\(3C\)](#), [unistd\(4\)](#).

getmsg(2)

NAME

getmsg, getpmsg - get message off a stream

SYNOPSIS

```
#include <stropts.h>

int getmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr,
           int *flagsp);

int getpmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr,
            int *bandp, int *flagsp);
```

DESCRIPTION

getmsg() retrieves the contents of a message located at the stream head read queue from a STREAMS file, and places the contents into user specified buffer(s). The message must contain either a data part, a control part, or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the STREAMS module that generated the message.

The function getpmsg() does the same thing as getmsg(), but provides finer control over the priority of the messages received. Except where noted, all information pertaining to getmsg() also pertains to getpmsg().

fd specifies a file descriptor referencing an open stream. ctlptr and dataptr each point to a strbuf structure, which contains the following members:

```
int maxlen;    /* maximum buffer length */
int len;       /* length of data */
char *buf;     /* ptr to buffer */
```

buf points to a buffer in which the data or control information is to be placed, and maxlen indicates the maximum number of bytes this buffer can hold. On return, len contains the number of bytes of data or control information actually received, or 0 if there is a zero-length control or data part, or -1 if no data or control information is present in the message. flagsp should point to an integer that indicates the type of message the user is able to receive. This is described later.

ctlptr is used to hold the control part from the message and dataptr is used to hold the data part from the message. If ctlptr (or dataptr) is NULL or the maxlen field is -1, the control (or data) part of the message is not processed and is left on the stream head read queue. If ctlptr (or dataptr) is not NULL and there is no corresponding control (or data) part of the messages on the stream head read queue, len is set to -1. If the maxlen field is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and len is set to 0. If the maxlen field is set to 0 and there are more than zero bytes of control (or data) information, that information is left on the read queue and len is set to 0. If the maxlen field in ctlptr or dataptr is less than, respectively, the control or data part of the message, maxlen bytes are retrieved. In this case, the remainder of the message is left on the stream head read queue and a non-zero result is provided, as described below under **RETURN VALUE**.

By default, getmsg() processes the first available message on the stream head read queue. However, a user may choose to retrieve only high priority messages by setting the integer pointed to by flagsp to RS_HIPRI. In this case, getmsg() processes the next message only if it is a high priority message. If the integer pointed to by flagsp is 0, getmsg() retrieves any message available on the stream head read queue. In this case, on return, the integer pointed to by flagsp will be set to RS_HIPRI if a high priority message was retrieved, or 0 otherwise.

For getpmsg(), the flags are different. flagsp points to a bitmask with the following mutually-exclusive flags: MSG_HIPRI, MSG_BAND, and MSG_ANY. Like getmsg(), getpmsg() processes the first available message on the stream head read queue. A user may receive high-priority messages only by setting the integers pointed to by flagsp=MSG_HIPRI and by bandp=0 to 0. In this case, getpmsg() will only process the next message if it is a high-priority message. In a similar manner, a user may choose to retrieve a message from a

particular priority band by setting the integer pointed to by *flagsp* to `MSG_BAND` and the integer pointed to by *bandp* to the priority band of interest. In this case, `getpmsg()` will only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *bandp*, or if it is a high-priority message. If a user just wants to get the first message off the queue, the integer pointed to by *flagsp* should be set to `MSG_ANY` and the integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by *flagsp* will be set to `MSG_HIPRI` and the integer pointed to by *bandp* will be set to 0. Otherwise, the integer pointed to by *flagsp* will be set to `MSG_BAND` and the integer pointed to by *bandp* will be set to the priority band of the message.

If `O_NDELAY` and `O_NONBLOCK` are clear, `getmsg()` blocks until a message of the type specified by *flagsp* is available on the stream head read queue. If `O_NDELAY` or `O_NONBLOCK` has been set and a message of the specified type is not present on the read queue, `getmsg()` fails and sets `errno` to `EAGAIN`.

If a hangup occurs on the stream from which messages are to be retrieved, `getmsg()` continues to operate normally, as described above, until the stream head read queue is empty. Thereafter, it returns 0 in the `len` fields of *ctlptr* and *dataptr*.

RETURN VALUE

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A result of `MORECTL` indicates that more control information is waiting for retrieval. A result of `MOREDATA` indicates that more data are waiting for retrieval. A result of `MORECTL | MOREDATA` indicates that both types of information remain. Subsequent `getmsg()` calls retrieve the remainder of the message. However, if a message of higher priority has come in on the stream head read queue, the next call to `getmsg()` will retrieve that higher priority message before retrieving the remainder of the previously received partial message.

If the high priority control part of the message is consumed, the message will be placed back on the queue as a normal message of band 0. Subsequent `getmsg()` and `getpmsg()` calls retrieve the remainder of the message. If, however, a priority message arrives or already exists on the stream head, the subsequent call to `getmsg()` or `getpmsg()` retrieves the higher-priority message before retrieving the remainder of the message that was put back.

Upon failure, `getmsg()` and `getpmsg()` return `-1` and set `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`getmsg()` or `getpmsg()` will fail if one or more of the following apply:

EAGAIN

The `O_NDELAY` or `O_NONBLOCK` flag is set, and no messages are available.

EBADF

fd is not a valid file descriptor open for reading.

EBADMSG

Queued message to be read is not valid for `getmsg()` or `getpmsg()` or a pending file descriptor is at the stream head.

EFAULT

ctlptr, *dataptr*, *bandp*, or *flagsp* points to a location outside the allocated address space.

EINTR

A signal was caught during the `getmsg()` system call.

EINVAL

An illegal value was specified in *flagsp*, or the stream referenced by *fd* is linked under a multiplexer.

ENOSTR

A stream is not associated with *fd*.

In addition, `getmsg()` and `getpmsg()` will fail if the stream head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `getmsg()` or `getpmsg()` but

reflects the prior error.

SEE ALSO

`poll(2)`, `putmsg(2)`, `read(2)`, `write(2)`, `stropts(5)`.

"Programmer's Guide: STREAMS".

NAME

getpid, getpgrp, getppid, getpgid - get processIDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getpgrp(void);
pid_t getppid(void);
pid_t getpgid(pid_t pid);
```

DESCRIPTION

getpid() returns the process ID of the calling process.

getpgrp() returns the process group ID of the calling process.

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

getpgid() returns the process group ID of the process whose process ID is equal to *pid*, or the process group ID of the calling process, if *pid* is equal to zero.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

getpgid() will fail if one or more of the following apply:

EPERM

The process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of that process from the calling process.

ESRCH

There is no process with a process ID equal to *pid*.

The getpgid() function may fail if:

EINVAL

The value of the *pid* argument is invalid.

RESULT

Upon successful completion, getpgid() returns a process group ID. Otherwise, a value of (pid_t)-1 is returned and *errno* is set to indicate the error.

SEE ALSO

[exec\(2\)](#), [fork\(2\)](#), [getpid\(2\)](#), [getsid\(2\)](#), [setpgid\(2\)](#), [setsid\(2\)](#), [setpgrp\(2\)](#), [signal\(2\)](#), [unistd\(4\)](#), [types\(5\)](#).

getrlimit(2)

NAME

getrlimit, setrlimit, getrlimit64, setrlimit64 - control system resource usage

SYNOPSIS

```
#include <sys/resource.h>
int  getrlimit(int resource, struct rlimit *rlp);
int  setrlimit(int resource, const struct rlimit *rlp);
int  getrlimit64(int resource, struct rlimit64 *rlp);
int  setrlimit64(int resource, const struct rlimit64 *rlp);
```

DESCRIPTION

Limits on the consumption of a variety of system resources by a process and each process it creates may be obtained with `getrlimit()` and set with `setrlimit()`.

Each call to either `getrlimit()` or `setrlimit()` identifies a specific resource to be operated upon as well as a resource limit. A resource limit is represented by an `rlimit` structure. The `rlim_cur` member specifies the current or soft limit and the `rlim_max` member specifies the maximum or hard limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a process with appropriate privileges can raise a hard limit. Both hard and soft limits can be changed in a single call to `setrlimit()` subject to the constraints described above.

The value `RLIM_INFINITY`, defined in `<sys/resource.h>`, is considered to be larger than any other limit value. If a call to `getrlimit()` returns `RLIM_INFINITY` for a resource, it means the implementation does not enforce limits on that resource. Specifying `RLIM_INFINITY` as any resource limit value on a successful call to `setrlimit()` inhibits enforcement of that resource limit.

When using the `getrlimit()` function, if a resource limit can be represented correctly in an object of type `rlim_t` then its representation is returned; otherwise if the value of the resource limit is equal to that of the corresponding saved hard limit the value returned is `RLIM_SAVED_MAX`; otherwise the value returned is `RLIM_SAVED_CUR`.

When using the `setrlimit()` function, if the requested new limit is `RLIM_INFINITY` the new limit will be "no limit"; otherwise if the requested new limit is `RLIM_SAVED_MAX` the new limit will be the corresponding saved hard limit; otherwise if the requested new limit is `RLIM_SAVED_CUR` the new limit will be the corresponding saved soft limit; otherwise the new limit will be the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type `rlim_t` then it will be overwritten with the new limit.

The result of setting a limit to `RLIM_SAVED_MAX` or `RLIM_SAVED_CUR` is unspecified unless a previous call to `getrlimit()` returned that value as the soft or hard limit for the corresponding resource limit.

The determination of whether a limit can be correctly represented in an object of type `rlim_t` is implementation-dependent. For example, some implementations permit a limit whose value is greater than `RLIM_INFINITY` and others do not.

The `exec` family of functions also cause resource limits to be saved [see [exec\(2\)](#)].

There is no functional difference between `getrlimit()/setrlimit()` and `getrlimit64()/setrlimit64()`, except for the interpretation of `rlimit/rlimit64` [see [resource\(5\)](#)].

The following resources are defined:

RLIMIT_CORE

This is the maximum size of a core file in bytes that may be created by a process. A limit of 0 will prevent the creation of a core file. If this limit is exceeded, the writing of a core file will terminate at this size.

RLIMIT_CPU

This is the maximum amount of CPU time in seconds used by a process. If this limit is exceeded, `SIGXCPU` is generated for the process. If the process is blocking, catching or ignoring `SIGXCPU`, the behavior is unspecified.

RLIMIT_DATA

This is the maximum size of a process' data segment in bytes. If this limit is exceeded, the `brk()`, `malloc()` and `sbrk()` functions will fail with `errno` set to `ENOMEM`.

RLIMIT_FSIZE

This is the maximum size of a file in bytes that may be created by a process. A limit of 0 will prevent the creation of a file. If a write or truncate operation would cause this limit to be exceeded, `SIGXFSZ` is generated for the process. If the process is blocking, catching or ignoring `SIGXFSZ`, continued attempts to increase the size of a file from end-of-file to beyond the limit will fail with `errno` set to `EFBIG`.

RLIMIT_NOFILE

This is a number one greater than the maximum value that the system may assign to a newly-created descriptor. If this limit is exceeded, functions that allocate new file descriptors may fail with `errno` set to `EMFILE`. This limit constrains the number of file descriptors that a process may allocate.

RLIMIT_STACK

This is the maximum size of a process' stack in bytes. The implementation will not automatically grow the stack beyond this limit. If this limit is exceeded, `SIGSEGV` is generated for the process. If the process is blocking or ignoring `SIGSEGV`, or is catching `SIGSEGV` and has not made arrangements to use an alternate stack, the disposition of `SIGSEGV` will be set to `SIG_DFL` before it is generated.

RLIMIT_AS or RLIMIT_VMEM

This is the maximum size of a process' total available memory, in bytes. If this limit is exceeded, the `brk()`, `malloc()`, `mmap()` and `sbrk()` functions will fail with `errno` set to `ENOMEM`. In addition, the automatic stack growth will fail with the effects outlined above.

Because limit information is stored in the per-process information, the shell builtin `ulimit(1)` must directly execute this system call if it is to affect all future processes created by the shell.

RETURN VALUE

Upon successful completion, `getrlimit()` and `setrlimit()` return 0. Otherwise, these functions return `-1` and set `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

Under the following conditions, the functions `getrlimit()` and `setrlimit()` fail and set `errno` to:

EINVAL

An invalid *resource* was specified; or in a `setrlimit()` call, the new `rlim_cur` exceeds the new `rlim_max`.

EPERM

The limit specified to `setrlimit()` would have raised the maximum limit value, and the calling process does not have appropriate privileges.

The `setrlimit()` function may fail if:

EINVAL

The limit specified cannot be lowered because current usage is already higher than the limit.

NOTES

If a process attempts to set the hard limit or soft limit for `RLIMIT_NOFILE` to less than the value of `_POSIX_OPEN_MAX` from `<limits.h>` unexpected behavior may occur.

These functions map limits that they cannot represent correctly to and from `RLIM_SAVED_MAX` and `RLIM_SAVED_CUR`. These values do not require any special handling by programs. They may be thought of

as tokens that the kernel hands out to programs that cannot handle the real answer, and that remind the kernel, when the tokens come back from the user, of what value is really meant.

If `setrlimit()` fails for any reason (for example, `EPERM`), the resource limits and saved resource limits remain unchanged.

Few, if any, programs will need to refer explicitly to `RLIM_SAVED_MAX` or `RLIM_SAVED_CUR`. Those that do should not use them in C-language switch cases since they may have the same value in some implementations.

A limit that can be represented correctly in an object of type `rlim_t` is either "no limit", which is represented with `RLIM_INFINITY`, or has a value not equal to any of `RLIM_INFINITY` or `RLIM_SAVED_MAX` or `RLIM_SAVED_CUR` and which can be represented correctly in an object of type `rlim_t` and which meets any additional implementation-specific criteria for correct representation.

SEE ALSO

`ulimit(1)`, `brk(2)`, `exec(2)`, `fork(2)`, `open(2)`, `sigaltstack(2)`, `ulimit(2)`, `getdtablesize(3C)`, `malloc(3C)`, `sysconf(3C)`, `lfs(5)`, `resource(5)`, `signal(5)`, `stropts(5)`.

NAME

getrusage - get information about resource utilization

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>
getrusage(int who, struct rusage *rusage);
```

DESCRIPTION

getrusage returns information about the resources utilized by the current process, or all its terminated child processes. The interpretation for some values reported, such as ru_idrss, are dependent on the clock tick interval. This interval is an implementation dependent value.

The who parameter is one of RUSAGE_SELF or RUSAGE_CHILDREN. The buffer to which rusage points will be filled in with the following structure:

```
struct    rusage {
  struct timeval ru_utime;    /* user time used */
  struct timeval ru_stime;    /* system time used */
  int          ru_maxrss;     /* maximum resident set size */
  int          ru_ixrss;      /* currently 0 */
  int          ru_idrss;      /* integral resident set size */
  int          ru_isrss;      /* currently 0 */
  int          ru_minflt;     /* page faults not requiring physical I/O */
  int          ru_majflt;     /* page faults requiring physical I/O */
  int          ru_nswap;      /* swaps */
  int          ru_inblock;    /* block input operations */
  int          ru_oublock;    /* block output operations */
  int          ru_msgsnd;     /* messages sent */
  int          ru_msgrcv;     /* messages received */
  int          ru_nsignals;   /* signals received */
  int          ru_nvcsw;      /* voluntary context switches */
  int          ru_nivcsw;     /* involuntary context switches */
};
```

The fields 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\(3\)](#) system call). Also, see **NOTES**.

ru_ixrss

Currently returns 0.

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. Note: it does not take sharing into account. Also, see **NOTES**.

ru_isrss

Currently returns 0.

ru_minflt

The number of page faults serviced which did not require any physical I/O activity. Also, see **NOTES**.

`ru_majflt`

The number of page faults serviced which required physical I/O activity. This could include page ahead operations by the kernel. Also, see **NOTES**.

`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 VALUE

If successful, the value of the appropriate structure is filled in, and 0 is returned. If the call fails, a `-1` is returned.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`getrusage` will fail if:

EINVAL

The *who* parameter is not a valid value.

EFAULT

The address specified by the *rusage* argument is not in a valid portion of the process' address space.

NOTES

The numbers `ru_inblock` and `ru_oublock` account only for real I/O, and are approximate measures at best. Data supplied by the caching 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)` system calls. 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.

SEE ALSO

`sar(1M)`, `read(2)`, `times(2)`, `wait(2)`, `write(2)`, `getpagesize(3)`, `gettimeofday(3C)`, `resource(5)`, `sys_time(5)`.

NAME

getsid - get session ID

SYNOPSIS

```
#include <sys/types.h>
pid_t getsid(pid_t pid);
```

DESCRIPTION

The function `getsid()` returns the session ID of the process whose process ID is equal to *pid*. If *pid* is equal to `(pid_t)0`, `getsid()` returns the session ID of the calling process.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Under the following conditions, the function `getsid()` fails and sets `errno` to:

EPERM

The process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the session ID of that process from the calling process.

ESRCH

There is no process with a process ID equal to *pid*.

RETURN VALUE

Upon successful completion, the function `getsid()` returns the session ID of the specified process; otherwise, it returns a value of `(pid_t)-1` and sets `errno` to indicate an error.

SEE ALSO

[exec\(2\)](#), [fork\(2\)](#), [getpid\(2\)](#), [setpgid\(2\)](#), [setsid\(2\)](#), [types\(5\)](#).

getuid(2)**NAME**

getuid, geteuid, getgid, getegid - get user/groupIDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

DESCRIPTION

getuid() returns the real user ID of the calling process.
 geteuid() returns the effective user ID of the calling process.
 getgid() returns the real group ID of the calling process.
 getegid() returns the effective group ID of the calling process.

SEE ALSO

setuid(2), unistd(4), types(5).

ioctl(2) - mutex(2)**ioctl(2)****NAME**

ioctl - control devices and STREAMS

SYNOPSIS

```
#include <unistd.h>
int ioctl(int fildev, int request, ... /* arg */);
```

DESCRIPTION

The ioctl() function performs a variety of control functions on STREAMS devices. For non-STREAMS devices, the functions performed by this call are unspecified. The *request* argument and an optional third argument (with varying type) are passed to and interpreted by the appropriate part of the stream associated with *fildev*.

The *fildev* argument is an open file descriptor that refers to a device.

The *request* argument selects the control function to be performed and will depend on the STREAMS device being addressed.

The *arg* argument represents additional information that is needed by this specific STREAMS device to perform the requested function. The type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

The ioctl() commands applicable to STREAMS, their arguments, and error statuses that apply to each individual command are described below.

The following ioctl() commands, with error values indicated, are applicable to all STREAMS files:

I_PUSH

Pushes the module whose name is pointed to by *arg* onto the top of the current stream, just below the stream head. It then calls the `open()` function of the newly-pushed module.

The `ioctl()` function with the `I_PUSH` command will fail if:

- `EINVAL`
Invalid module name.
- `ENXIO`
Open function of new module failed.
- `ENXIO`
Hangup received on *fildes*.

`I_POP`

Removes the module just below the stream head of the stream pointed to by *fildes*. The *arg* argument should be 0 in an `I_POP` request.

The `ioctl()` function with the `I_POP` command will fail if:

- `EINVAL`
No module present in the stream.
- `ENXIO`
Hangup received on *fildes*.

`I_LOOK`

Retrieves the name of the module just below the stream head of the stream pointed to by *fildes*, and places it in a character string pointed to by *arg*. The buffer pointed to by *arg* should be at least `FMNAMESZ+1` bytes long, where `FMNAMESZ` is defined in `<stropts.h>`.

The `ioctl()` function with the `I_LOOK` command will fail if:

- `EINVAL`
No module present in the stream.

`I_FLUSH`

This request flushes read and/or write queues, depending on the value of *arg*. Valid *arg* values are:

- `FLUSHR`
Flush all read queues.
- `FLUSHW`
Flush all write queues.
- `FLUSHRW`
Flush all read and all write queues.

The `ioctl()` function with the `I_FLUSH` command will fail if:

- `EINVAL`
Invalid *arg* value.
- `EAGAIN` or `ENOSR`
Unable to allocate buffers for flush message.
- `ENXIO`
Hangup received on *fildes*.

`I_FLUSHBAND`

Flushes a particular band of messages. The *arg* argument points to a `bandinfo` structure. The `bi_flag` member may be one of `FLUSHR`, `FLUSHW`, or `FLUSHRW` as described above. The `bi_pri` member determines the priority band to be flushed.

`I_SETSIG`

Requests that the STREAMS implementation send the `SIGPOLL` signal to the calling process when a particular event has occurred on the stream associated with *fildes*. `I_SETSIG` supports an asynchronous processing capability in STREAMS. The value of *arg* is a bitmask that specifies the events for which the process should be signaled. It is the bitwise-OR of any combination of the following constants:

S_RDNORM

A normal (priority band set to 0) message has arrived at the head of a stream head read queue. A signal will be generated even if the message is of zero length.

S_RDBAND

A message with a non-zero priority band has arrived at the head of a stream head read queue. A signal will be generated even if the message is of zero length.

S_INPUT

A message, other than a high-priority message, has arrived at the head of a stream head read queue. A signal will be generated even if the message is of zero length.

S_HIPRI

A high-priority message is present on a stream head read queue. A signal will be generated even if the message is of zero length.

S_OUTPUT

The write queue for normal data (priority band 0) just below the stream head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.

S_WRNORM

Same as **S_OUTPUT**.

S_WRBAND

The write queue for a non-zero priority band just below the stream head is no longer full. This notifies the process that there is room on the queue for sending (or writing) priority data downstream.

S_MSG

A STREAMS signal message that contains the **SIGPOLL** signal has reached the front of the stream head read queue.

S_ERROR

Notification of an error condition has reached the stream head.

S_HANGUP

Notification of a hangup has reached the stream head.

S_BANDURG

When used in conjunction with **S_RDBAND**, **SIGURG** is generated instead of **SIGPOLL** when a priority message reaches the front of the stream head read queue.

If *arg* is 0, the calling process will be unregistered and will not receive further **SIGPOLL** signals for the stream associated with *fildev*.

Processes that wish to receive **SIGPOLL** signals must explicitly register to receive them using **I_SETSIG**. If several processes register to receive this signal for the same event on the same stream, each process will be signaled when the event occurs.

The **ioctl()** function with the **I_SETSIG** command will fail if:

EINVAL

The value of *arg* is invalid.

EINVAL

The value of *arg* is 0 and the calling process is not registered to receive the **SIGPOLL** signal.

EAGAIN

There were insufficient resources to store the signal request.

I_GETSIG

Returns the events for which the calling process is currently registered to be sent a **SIGPOLL** signal. The events are returned as a bitmask in an **int** pointed to by *arg*, where the events are those specified in the description of **I_SETSIG** above.

The **ioctl()** function with the **I_GETSIG** command will fail if:

EINVAL

Process is not registered to receive the SIGPOLL signal.

I_FIND

This request compares the names of all modules currently present in the stream to the name pointed to by *arg*, and returns 1 if the named module is present in the stream, or returns 0 if the named module is not present.

The `ioctl()` function with the `I_FIND` command will fail if:

`EINVAL`

arg does not contain a valid module name.

I_PEEK

This request allows a process to retrieve the information in the first message on the stream head read queue without taking the message off the queue. It is analogous to `getmsg()` except that this command does not remove the message from the queue. The *arg* argument points to a `strpeek` structure.

The *maxlen* member in the `ctlbuf` and `databuf` `strbuf` structures must be set to the number of bytes of control information and/or data information, respectively, to retrieve. The *flags* member may be marked `RS_HIPRI` or 0, as described by `getmsg(2)`. If the process sets *flags* to `RS_HIPRI`, for example, `I_PEEK` will only look for a high-priority message on the stream head read queue.

`I_PEEK` returns 1 if a message was retrieved, and returns 0 if no message was found on the stream head read queue, or if the `RS_HIPRI` flag was set in *flags* and a high-priority message was not present on the stream head read queue. It does not wait for a message to arrive. On return, `ctlbuf` specifies information in the control buffer, `databuf` specifies information in the data buffer, and *flags* contains the value `RS_HIPRI` or 0.

I_SRDOPT

Sets the read mode using the value of the argument *arg*. Read modes are described in `read(2)`.

Valid *arg* flags are:

`RNORM`

Byte-stream mode, the default.

`RMSGD`

Message-discard mode.

`RMSGN`

Message-nondiscard mode.

The bitwise inclusive OR of `RMSGD` and `RMSGN` will return `EINVAL`. The bitwise inclusive OR of `RNORM` and either `RMSGD` or `RMSGN` will result in the other flag overriding `RNORM` which is the default.

In addition, treatment of control messages by the stream head may be changed by setting any of the following flags in *arg*:

`RPROTNORM`

Fail `read()` with `EBADMSG` if a message containing a control part is at the front of the stream head read queue.

`RPROTDAT`

Deliver the control part of a message as data when a process issues a `read()`.

`RPROTDIS`

Discard the control part of a message, delivering any data portion, when a process issues a `read()`.

The `ioctl()` function with the `I_SRDOPT` command will fail if:

`EINVAL`

The *arg* argument is not valid.

I_GRDOPT

Returns the current read mode setting as, described above, in an `int` pointed to by the argument *arg*. Read modes are described in `read(2)`.

I_NREAD

Counts the number of data bytes in the data part of the first message on the stream head read queue and places this value in the `int` pointed to by *arg*. The return value for the command is the number of messages on the stream head read queue. For example, if 0 is returned in *arg*, but the `ioctl()` return value is greater than 0, this indicates that a zero-length message is next on the queue.

I_FDINSERT

Creates a message from specified buffer(s), adds information about another stream, and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The *arg* argument points to a `strfdinsert` structure.

The *len* member in the `ctlbuf strbuf` structure must be set to the size of a pointer plus the number of bytes of control information to be sent with the message. The *fildev* member specifies the file descriptor of the other stream, and the *offset* member, which must be suitably aligned for use as a pointer, specifies the offset from the start of the control buffer where `I_FDINSERT` will store a pointer whose interpretation is specific to the stream end. The *len* member in the `databuf strbuf` structure must be set to the number of bytes of data information to be sent with the message, or to 0 if no data part is to be sent.

The *flags* member specifies the type of message to be created. A normal message is created if *flags* is set to 0, and a high-priority message is created if *flags* is set to `RS_HIPRI`. For non-priority messages, `I_FDINSERT` will block if the stream write queue is full due to internal flow control conditions. For priority messages, `I_FDINSERT` does not block on this condition. For non-priority messages, `I_FDINSERT` does not block when the write queue is full and `O_NONBLOCK` is set. Instead, it fails and sets `errno` to `EAGAIN`.

`I_FDINSERT` also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the stream, regardless of priority or whether `O_NONBLOCK` has been specified. No partial message is sent.

The `ioctl()` function with the `I_FDINSERT` command will fail if:

EAGAIN

A non-priority message is specified, the `O_NONBLOCK` flag is set, and the stream write queue is full due to internal flow control conditions.

EAGAIN or ENOSR

Buffers can not be allocated for the message that is to be created.

EINVAL

One of the following:

- The *fd* member of the `strfdinsert` structure is not a valid, open stream file descriptor.
- The size of a pointer plus *offset* is greater than the *len* member for the buffer specified through *ctlptr*.
- The *offset* member does not specify a properly-aligned location in the data buffer.
- An undefined value is stored in *flags*.

ENXIO

Hangup received on *fd* or *fildev*.

ERANGE

The *len* member for the buffer specified through *databuf* does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module or the *len* member for the buffer specified through *databuf* is larger than the maximum configured size of the data part of a message; or the *len* member for the buffer specified through *ctlbuf* is larger than the maximum configured size of the control part of a message.

I_STR

Constructs an internal STREAMS `ioctl()` message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send `ioctl()` requests to downstream modules and drivers. It allows information to be sent with `ioctl()`, and returns to the process any information sent upstream by the downstream recipient. `I_STR` blocks until the system responds with either a positive or negative acknowledgement message, or until the request "times out" after some period of time. If the request times out, it fails with `errno` set to `ETIME`.

At most, one `I_STR` can be active on a stream. Further `I_STR` calls will block until the active `I_STR` completes at the stream head. The default timeout interval for these requests is 15 seconds. The `O_NONBLOCK` flag has no effect on this call.

To send requests downstream, `arg` must point to a `strioc1` structure.

The `ic_cmd` member is the internal `ioctl()` command intended for a downstream module or driver and `ic_timeout` is the number of seconds (`-1` = infinite, `0` = use implementation-dependent timeout interval, `>0` = as specified) an `I_STR` request will wait for acknowledgement before timing out. `ic_len` is the number of bytes in the data argument, and `ic_dp` is a pointer to the data argument. The `ic_len` member has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the process (the buffer pointed to by `ic_dp` should be large enough to contain the maximum amount of data that any module or the driver in the stream can return).

The stream head will convert the information pointed to by the `strioc1` structure to an internal `ioctl()` command message and send it downstream.

The `ioctl()` function with the `I_STR` command will fail if:

`EAGAIN` or `ENOSR`

Unable to allocate buffers for the `ioctl()` message.

`EINVAL`

The `ic_len` member is less than 0 or larger than the maximum configured size of the data part of a message, or `ic_timeout` is less than `-1`.

`ENXIO`

Hangup received on *fildev*.

`ETIME`

A downstream `ioctl()` timed out before acknowledgement was received.

An `I_STR` can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the stream head. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the `ioctl()` command sent downstream fails. For these cases, `I_STR` fails with `errno` set to the value in the message.

`I_SWROPT`

Sets the write mode using the value of the argument `arg`. Valid bit settings for `arg` are:

`SNDZERO`

Send a zero-length message downstream when a `write()` of 0 bytes occurs. To not send a zero-length message when a `write()` of 0 bytes occurs, this bit must not be set in `arg` (for example, `arg` would be set to 0).

The `ioctl()` function with the `I_SWROPT` command will fail if:

`EINVAL`

`arg` is not the above value.

`I_GWROPT`

Returns the current write mode setting, as described above, in the `int` that is pointed to by the argument `arg`.

`I_SENDFD`

`I_SENDFD` creates a new reference to the open file description associated with the file descriptor `arg`, and writes a message on the STREAMS-based pipe *fildev* containing this reference, together with the user ID and group ID of the calling process.

The `ioctl()` function with the `I_SENDFD` command will fail if:

EAGAIN

The sending stream is unable to allocate a message block to contain the file pointer; or the read queue of the receiving stream head is full and cannot accept the message sent by `I_SENDFD`.

EBADF

The *arg* argument is not a valid, open file descriptor.

EINVAL

The *fildev* argument is not connected to a stream pipe.

ENXIO

Hangup received on *fildev*.

I_RECVFD

Retrieves the reference to an open file description from a message written to a STREAMS-based pipe using the `I_SENDFD` command, and allocates a new file descriptor in the calling process that refers to this open file description. The *arg* argument is a pointer to an `strrecvfd` data structure as defined in `<stropts.h>`.

The *fd* member is a file descriptor. The *uid* and *gid* members are the effective user ID and effective group ID, respectively, of the sending process.

If `O_NONBLOCK` is not set `I_RECVFD` blocks until a message is present at the stream head. If `O_NONBLOCK` is set, `I_RECVFD` fails with `errno` set to `EAGAIN` if no message is present at the stream head.

If the message at the stream head is a message sent by an `I_SENDFD`, a new file descriptor is allocated for the open file descriptor referenced in the message. The new file descriptor is placed in the *fd* member of the `strrecvfd` structure pointed to by *arg*.

The `ioctl()` function with the `I_RECVFD` command will fail if:

EAGAIN

A message is not present at the stream head read queue and the `O_NONBLOCK` flag is set.

EBADMSG

The message at the stream head read queue is not a message containing a passed file descriptor.

EMFILE

The process has the maximum number of file descriptors currently open that it is allowed.

ENXIO

Hangup received on *fildev*.

I_LIST

This request allows the process to list all the module names on the stream, up to and including the topmost driver name. If *arg* is a null pointer, the return value is the number of modules, including the driver, that are on the stream pointed to by *fildev*. This lets the process allocate enough space for the module names. Otherwise, it should point to an `str_list` structure.

The *sl_nmods* member indicates the number of entries the process has allocated in the array. Upon return, the *sl_modlist* member of the `str_list` structure contains the list of module names, and the number of entries that have been filled into the *sl_modlist* array is found in the *sl_nmods* member (the number includes the number of modules including the driver). The return value from `ioctl()` is 0. The entries are filled in starting at the top of the stream and continuing downstream until either the end of the stream is reached, or the number of requested modules (*sl_nmods*) is satisfied.

The `ioctl()` function with the `I_LIST` command will fail if:

EINVAL

The *sl_nmods* member is less than 1.

EAGAIN or ENOSR

Unable to allocate buffers.

I_ATMARK

This request allows the process to see if the message at the head of the stream head read queue is marked by some module downstream. The *arg* argument determines how the checking is done when there may be multiple marked messages on the stream head read queue. It may take on the following values:

ANYMARK

Check if the message is marked.

LASTMARK

Check if the message is the last one marked on the queue.

The bitwise inclusive OR of the flags ANYMARK and LASTMARK is permitted.

The return value is 1 if the mark condition is satisfied and 0 otherwise.

The `ioctl()` function with the `I_ATMARK` command will fail if:

EINVAL

Invalid *arg* value.

I_CKBAND

Check if the message of a given priority band exists on the stream head read queue. This returns 1 if a message of the given priority exists, 0 if no message exists, or -1 on error. *arg* should be of type `int`.

The `ioctl()` function with the `I_CKBAND` command will fail if:

EINVAL

Invalid *arg* value.

I_GETBAND

Return the priority band of the first message on the stream head read queue in the integer referenced by *arg*.

The `ioctl()` function with the `I_GETBAND` command will fail if:

ENODATA

No message on the stream head read queue.

I_CANPUT

Check if a certain band is writable. *arg* is set to the priority band in question. The return value is 0 if the band is flow-controlled, 1 if the band is writable, or -1 on error.

The `ioctl()` function with the `I_CANPUT` command will fail if:

EINVAL

Invalid *arg* value.

I_SETCLTIME

This request allows the process to set the time the stream head will delay when a stream is closing and there is data on the write queues. Before closing each module or driver, if there is data on its write queue, the stream head will delay for the specified amount of time to allow the data to drain. If, after the delay, data is still present, they will be flushed. The *arg* argument is a pointer to an integer specifying the number of milliseconds to delay, rounded up to the nearest valid value. If `I_SETCLTIME` is not performed on a stream, an implementation-dependent default timeout interval is used.

The `ioctl()` function with the `I_SETCLTIME` command will fail if:

EINVAL

Invalid *arg* value.

I_GETCLTIME

This request returns the close time delay in the integer pointed to by *arg*.

Multiplexed STREAMS Configurations

The following four commands are used for connecting and disconnecting multiplexed STREAMS configurations. These commands use an implementation-dependent default timeout interval.

I_LINK

Connects two streams, where *fildev* is the file descriptor of the stream connected to the multiplexing driver, and *arg* is the file descriptor of the stream connected to another driver. The stream designated by *arg* gets connected below the multiplexing driver. `I_LINK` requires the multiplexing driver to send an acknowledgement message to the stream head regarding the connection. This call returns a multiplexer ID number (an identifier used to disconnect the multiplexer; see `I_UNLINK`) on success, and `-1` on failure.

The `ioctl()` function with the `I_LINK` command will fail if:

`ENXIO`

Hangup received on *fildev*.

`ETIME`

Time out before acknowledgement message was received at stream head.

`EAGAIN` or `ENOSR`

Unable to allocate STREAMS storage to perform the `I_LINK`.

`EBADF`

The *arg* argument is not a valid, open file descriptor.

`EINVAL`

The *fildev* argument does not support multiplexing; or *arg* is not a stream or is already connected downstream from a multiplexer; or the specified `I_LINK` operation would connect the stream head in more than one place in the multiplexed stream.

An `I_LINK` can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hangup is received at the stream head of *fildev*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, `I_LINK` fails with `errno` set to the value in the message.

`I_UNLINK`

Disconnects the two streams specified by *fildev* and *arg*. *fildev* is the file descriptor of the stream connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the `I_LINK` `ioctl()` command when a stream was connected downstream from the multiplexing driver. If *arg* is `MUXID_ALL`, then all streams that were connected to *fildev* are disconnected. As in `I_LINK`, this command requires acknowledgement.

The `ioctl()` function with the `I_UNLINK` command will fail if:

`ENXIO`

Hangup received on *fildev*.

`ETIME`

Time out before acknowledgement message was received at stream head.

`EAGAIN` or `ENOSR`

Unable to allocate buffers for the acknowledgement message.

`EINVAL`

Invalid multiplexer ID number.

An `I_UNLINK` can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hangup is received at the stream head of *fildev*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, `I_UNLINK` fails with `errno` set to the value in the message.

`I_PLINK`

Creates a *persistent connection* between two streams, where *fildev* is the file descriptor of the stream connected to the multiplexing driver, and *arg* is the file descriptor of the stream connected to another driver. This call creates a persistent connection which can exist even if the file descriptor *fildev* associated with the upper stream to the multiplexing driver is closed. The stream designated by *arg* gets connected via a persistent connection below the multiplexing driver. `I_PLINK` requires the multiplexing driver to send an acknowledgement message to the stream head. This call returns a multiplexer ID number (an identifier that may be used to disconnect the multiplexer, see `I_PUNLINK`) on success, and `-1` on failure.

The `ioctl()` function with the `I_PLINK` command will fail if:

`ENXIO`

Hangup received on *fildev*.

`ETIME`

Time out before acknowledgement message was received at stream head.

`EAGAIN` or `ENOSR`

Unable to allocate STREAMS storage to perform the `I_PLINK`.

`EBADF`

The *arg* argument is not a valid, open file descriptor.

`EINVAL`

The *fildev* argument does not support multiplexing; or *arg* is not a stream or is already connected downstream from a multiplexer; or the specified `I_PLINK` operation would connect the stream head in more than one place in the multiplexed stream.

An `I_PLINK` can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hangup is received at the stream head of *fildev*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, `I_PLINK` fails with `errno` set to the value in the message.

`I_PUNLINK`

Disconnects the two streams specified by *fildev* and *arg* from a persistent connection. The *fildev* argument is the file descriptor of the stream connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the `I_PLINK` `ioctl()` command when a stream was connected downstream from the multiplexing driver. If *arg* is `MUXID_ALL` then all streams which are persistent connections to *fildev* are disconnected. As in `I_PLINK`, this command requires the multiplexing driver to acknowledge the request.

The `ioctl()` function with the `I_PUNLINK` command will fail if:

`ENXIO`

Hangup received on *fildev*.

`ETIME`

Time out before acknowledgement message was received at stream head.

`EAGAIN` or `ENOSR`

Unable to allocate buffers for the acknowledgement message.

`EINVAL`

Invalid multiplexer ID number.

An `I_PUNLINK` can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hangup is received at the stream head of *fildev*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, `I_PUNLINK` fails with `errno` set to the value in the message.

RETURN VALUE

Upon successful completion, `ioctl()` returns a value other than `-1` that depends upon the STREAMS device control function. Otherwise, it returns `-1` and sets `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Under the following general conditions, `ioctl()` will fail if:

`EBADF`

The *fildev* argument is not a valid open file descriptor.

`EINTR`

A signal was caught during the `ioctl()` operation.

EINVAL

The stream or multiplexer referenced by *fildev* is linked (directly or indirectly) downstream from a multiplexer.

If an underlying device driver detects an error, then `ioctl()` will fail if:

EINVAL

The *request* or *arg* argument is not valid for this device.

EIO Some physical I/O error has occurred.**ENOTTY**

The *fildev* argument is not associated with a STREAMS device that accepts control functions.

ENXIO

The *request* and *arg* arguments are valid for this device driver, but the service requested can not be performed on this particular sub-device.

ENODEV

The *fildev* argument refers to a valid STREAMS device, but the corresponding device driver does not support the `ioctl()` function.

If a stream is connected downstream from a multiplexer, any `ioctl()` command except `I_UNLINK` and `I_PUNLINK` will set `errno` to `EINVAL`.

APPLICATION USAGE

The implementation-defined timeout interval for STREAMS has historically been 15 seconds.

SEE ALSO

`close(2)`, `fcntl(2)`, `getmsg(2)`, `open(2)`, `pipe(2)`, `poll(2)`, `putmsg(2)`, `read(2)`, `sigaction(2)`, `write(2)`, `unistd(4)`, `stropts(5)`, `streamio(7)`, `termio(7)`.

NAME

`kill` - send a signal to a process or a group of processes

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

DESCRIPTION

`kill()` sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in `signal.h` [see [signal\(5\)](#)], or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

On XSI-conform systems, `_POSIX_SAVED` is defined by the system. Therefore the real or effective user ID of the sending process must match the real or saved [from [exec\(2\)](#)] user ID of the receiving process unless the effective user ID of the sending process is superuser, or unless *sig* is `SIGCONT` and the sending process has the same session ID as the receiving process.

If *pid* is greater than 0, *sig* will be sent to the process whose process ID is equal to *pid*.

If *pid* is 0, *sig* will be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the process group ID of the sender, and for which the process has permission to send a signal.

If *pid* is -1, *sig* will be sent to all processes (excluding an unspecified set of system processes) for which the process has permission to send that signal.

If *pid* is negative, but not -1, *sig* will be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the absolute value of *pid*, and for which the process has permission to send a signal.

If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked for the calling thread and if no other thread has *sig* unblocked or is waiting in a `sigwait()` function for *sig*, either *sig* or at least one pending unblocked signal will be delivered to the sending thread before `kill()` returns.

The user ID tests described above will not be applied when sending `SIGCONT` to a process that is a member of the same session as the sending process.

An implementation that provides extended security controls may impose further implementation-dependent restrictions on the sending of signals, including the null signal. In particular, the system may deny the existence of some or all of the processes specified by *pid*.

The `kill()` function is successful if the process has permission to send *sig* to any of the processes specified by *pid*. If `kill()` fails, no signal will be sent.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`kill()` will fail and no signal will be sent if one or more of the following apply:

EINVAL

sig is not a valid signal number.

EINVAL

sig is SIGKILL and *pid* is (pid_t) 1 (i.e., *pid* specifies *proc1*).

ESRCH

No process or process group can be found corresponding to that specified by *pid*.

EPERM

The user ID of the sending process is not privileged, and its real or effective user ID does not match the real or saved user ID of the receiving process, and the calling process is not sending SIGCONT to a process that shares the same session ID.

NOTES

`sigsend()` is a more versatile way to send signals to processes. The user is encouraged to use `sigsend()` instead of `kill()`.

SEE ALSO

`kill(1)`, `getpid(2)`, `getsid(2)`, `setpgrp(2)`, `sigaction(2)`, `signal(2)`, `sigsend(2)`, `signal(5)`, `types(5)`.

NAME

`link` - set up link to a file

SYNOPSIS

```
#include <unistd.h>
int link(const char *path1, const char *path2);
```

DESCRIPTION

path1 points to a pathname naming an existing file. *path2* points to a pathname naming the new directory entry to be created. `link()` creates a new link (directory entry) for the existing file and increments the link count by one.

Upon successful completion, `link()` marks for update the `st_ctime` field of the file. Also, the `st_ctime` and `st_mtime` fields of the directory that contains the new entry are marked for update.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`link()` will fail and no link will be created if one or more of the following apply:

EACCES

A component of either path prefix denies search permission, or the requested link requires writing in a directory with a mode that denies write permission, or the calling process does not have permission to access the existing file and this is required by the implementation.

EEXIST

The link named by *path2* exists.

EFAULT

path1 or *path2* points outside the allocated address space of the process.

EINTR

A signal was caught during the `link()` system call.

ELOOP

Too many symbolic links were encountered in translating *path1* or *path2*.

EMLINK

The maximum number of links to a file would exceed `LINK_MAX`.

EMULTIHOP

Components of *path1* or *path2* require hopping to multiple remote machines and the file system type does not allow it.

ENAMETOOLONG

The length of the *path1* or *path2* argument exceeds `PATH_MAX` or the length of a *path1* or *path2* component exceeds `NAME_MAX`.

ENOTDIR

A component of either path prefix is not a directory.

ENOENT

path1 or *path2* is a null pathname.

ENOENT

A component of either path prefix does not exist.

ENOENT

The file named by *path1* does not exist.

ENOLINK

path1 or *path2* points to a remote machine and the link to that machine is no longer active.

ENOSPC

The directory that would contain the link cannot be extended.

EPERM

The file named by *path1* is a directory and the effective user ID is not superuser.

EROFS

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

EXDEV

The link named by *path2* and the file named by *path1* are on different logical devices (file systems) and the implementation does not support links between file systems, or *path1* refers to a named STREAM.

The `link()` function may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

RESULT

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

SEE ALSO

`unlink(2)`, `unistd(4)`.

NAME

`lseek`, `lseek64` - move read/write file pointer

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
off64_t lseek64(int fd, off64_t offset, int whence);
```

DESCRIPTION

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

- If *whence* is `SEEK_SET`, the pointer is set to *offset* bytes.
- If *whence* is `SEEK_CUR`, the pointer is set to its current location plus *offset*.
- If *whence* is `SEEK_END`, the pointer is set to the size of the file plus *offset*.

On success, `lseek()` returns the resulting pointer location, as measured in bytes from the beginning of the file. Note that if *fd* is a remote file descriptor and *offset* is negative, `lseek()` returns the file pointer even if it is negative.

`lseek()` allows the file pointer to be set beyond the existing data in the file. If data are later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes of value 0 until data are written into the gap.

There is no functional difference between `lseek()` and `lseek64()`, except for the interpretation of `off64_t` [see [lfs\(5\)](#)].

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`lseek()` fails and the file pointer remains unchanged if one or more of the following apply:

EBADF

fd is not an open file descriptor.

ESPIPE

fd is associated with a pipe or fifo.

EINVAL

whence is not `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`. The process also gets a `SIGSYS` signal.

EINVAL

fd is not a remote file descriptor, and the resulting file pointer would be negative.

EOVERFLOW

The resulting file offset would be a value which cannot be represented correctly in an object of type `off_t`.

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

RESULT

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

NOTES

`lseek()` will fail if the resulting file offset would exceed the largest value that can be represented correctly in the related data type which is in use for the call, and will set `errno` to `EOVERFLOW` (permitted by PASC Interpretation 1003.1-90 #75).

Programs typically, but incorrectly, fail to check the return value of this function, which renders the error return less useful. On the other hand, returning an incorrect offset can result in serious malfunction as well.

An `lseek()` to the end of a file using

```
lseek(fd, 0, SEEK_END);
```

is quite common. It is unfortunate that these fail on a too-large file since the return value is usually ignored.

Another potentially serious consequence of ignoring the return value of `lseek()` is that programs which extend data files by attempting to seek beyond the end-of-file and then writing may instead overwrite existing data.

For example, typical implementations of the `dbm` and `ndbm` libraries contain code such as:

```
(void) lseek(db->dbm_pagf, blkno*PBLKSIZ, L_SET);  
if (write(db->dbm_pagf, pagebuf, PBLKSIZ) != PBLKSIZ)  
    ... error handling ...
```

The problem is that the return code of `lseek()` is not checked and so if `blkno*PBLKSIZ` overflows the `lseek()` will fail (or will seek to an unintended offset) and the data will be written to an unintended offset.

SEE ALSO

`creat(2)`, `dup(2)`, `fcntl(2)`, `open(2)`, `unistd(4)`, `lfs(5)`, `types(5)`.

NAME

memcntl - control memory management

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>
int memcntl(caddr_t addr, size_t len, int cmd, caddr_t arg,
            int attr, int mask);
```

DESCRIPTION

The function `memcntl()` allows the calling process to apply a variety of control operations over the address space specified by the mappings `[addr, addr + len)`.

`addr` must be a multiple of the pagesize as returned by `sysconf(3C)`. The scope of the control operations can be further defined with additional selection criteria (in the form of attributes) according to the bit pattern contained in `attr`.

The following attributes specify page mapping selection criteria:

`SHARED`

Page is mapped shared.

`PRIVATE`

Page is mapped private.

The following attributes specify page protection selection criteria:

`PROT_READ`

Page can be read.

`PROT_WRITE`

Page can be written.

`PROT_EXEC`

Page can be executed.

The selection criteria are constructed by an OR of the attribute bits and must match exactly.

In addition, the following criteria may be specified:

`PROC_TEXT`

Process text.

`PROC_DATA`

Process data.

where `PROC_TEXT` specifies all privately mapped segments with read and execute permission, and `PROC_DATA` specifies all privately mapped segments with write permission.

Selection criteria can be used to describe various abstract memory objects within the address space on which to operate. If an operation shall not be constrained by the selection criteria, `attr` must have the value 0.

The operation to be performed is identified by the argument `cmd`. The symbolic names for the operations are defined in the file `sys/mman.h` as follows:

`MC_LOCK`

Locks in memory all pages in the range with the attributes `attr`. A given page may be locked multiple times through different mappings; however, within a given mapping, page locks do not nest. Multiple lock operations on the same address in the same process will all be removed with a single unlock operation. A page locked in one process and mapped in another (or visible through a different mapping in the locking process) is locked in memory until the locking process does either an implicit or an explicit unlock operation. If a locked mapping is removed, or a page is deleted through file removal or truncation, an unlock operation is implicitly performed. If a writable `MAP_PRIVATE` page in the address

range is changed, the lock will be transferred to the private page.

At present *arg* is unused, but must be 0 to ensure compatibility with potential future enhancements.

MC_LOCKAS

Locks in memory all pages mapped by the address space with the attributes *attr*. At present *addr* and *len* are unused, but must be `NULL` and 0 respectively, to ensure compatibility with potential future enhancements. *arg* is a bit pattern built from the flags:

MCL_CURRENT

Lock current mappings

MCL_FUTURE

Lock future mappings

The value of *arg* 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. If `MCL_FUTURE` is specified, then all mappings subsequently added to the address space will be locked, provided sufficient memory is available.

MC_SYNC

Saves all modified pages in the range with attributes *attr*. Optionally, invalidates cache copies. The backing storage for a modified `MAP_SHARED` mapping is the file the page is mapped to; the backing storage for a modified `MAP_PRIVATE` mapping is its swap area. *arg* is a bit pattern built from the flags used to control the behavior of the operation:

MS_ASYNC

Perform asynchronous writes

MS_SYNC

Perform synchronous writes

MS_INVALIDATE

Invalidate mappings

`MS_ASYNC` returns immediately once all write operations are scheduled; with `MS_SYNC` the system call will not return until all write operations are completed.

`MS_INVALIDATE` invalidates all cached copies of data in memory, so that further references to the pages will be obtained by the system from their backing storage locations. This operation should be used by applications that require a memory object to be in a known state.

MC_UNLOCK

Unlocks all pages in the range with attributes *attr*. At present *arg* is unused, but must be 0 to ensure compatibility with potential future enhancements.

MC_UNLOCKAS

Removes address space memory locks, and locks on all pages in the address space with attributes *attr*. At present *addr*, *len*, and *arg* are unused, but must be `NULL` and 0 respectively to ensure compatibility with potential future enhancements.

The *mask* argument must be zero; it is reserved for future use.

Locks established with the lock operations are not inherited by a child process after `fork(). memcntl()` fails if it attempts to lock more memory than a system-specific limit.

Due to the potential impact on system resources, all operations, with the exception of `MC_SYNC`, are restricted to processes with superuser effective user ID. The `memcntl()` function subsumes the operations of `plock()` and `mctl()`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Under the following conditions, the function `memcntl()` fails and sets `errno` to:

EAGAIN

If some or all of the memory identified by the operation could not be locked when `MC_LOCK` or

MC_LOCKAS is specified.

EBUSY

If some or all the addresses in the range $[addr, addr + len)$ are locked and the option MC_SYNC is specified with MS_INVALIDATE.

EINVAL

If *addr* is not a multiple of the page size as returned by `sysconf()`.

EINVAL

If *addr* and/or *len* do not have the value 0 when MC_LOCKAS or MC_UNLOCKAS is specified.

EINVAL

If *arg* is not valid for the function specified.

EINVAL

If invalid selection criteria are specified in *attr*.

ENOMEM

If some or all the addresses in the range $[addr, addr + len)$ are invalid for the address space of the process or pages not mapped are specified.

EPERM

If the process' effective user ID is not superuser and one of the options MC_LOCK, MC_LOCKAS, MC_UNLOCK or MC_UNLOCKAS was specified.

RETURN VALUE

Upon successful completion, the function `memcntl()` returns a value of 0; otherwise, it returns a value of -1 and sets `errno` to indicate an error.

SEE ALSO

`mmap(2)`, `mprotect(2)`, `plock(2)`, `mctl(3)`, `mlock(3C)`, `mlockall(3C)`, `msync(3C)`, `sysconf(3C)`, `mman(5)`, `types(5)`.

NAME

`mincore` - determine residency of memory pages

SYNOPSIS

```
#include <unistd.h>
int mincore(caddr_t addr, size_t len, char *vec);
```

DESCRIPTION

`mincore()` returns the primary memory residency status of pages in the address space covered by mappings in the range `[addr, addr + len)`. The status is returned as a character-per-page in the character array referenced by `*vec` (which the system assumes to be large enough to encompass all the pages in the address range). The least significant bit of each character is set to 1 to indicate that the referenced page is in primary memory, 0 if it is not. The settings of other bits in each character are undefined and may contain other information in future implementations.

`mincore()` returns residency information that is accurate at an instant in time. Because the system may frequently adjust the set of pages in memory, this information may quickly be outdated. Only locked pages are guaranteed to remain in memory [see `memcntl(2)`].

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`mincore()` fails if:

EFAULT

`*vec` includes an out-of-range or otherwise inaccessible address.

EINVAL

`addr` is not a multiple of the page size as returned by `sysconf(3C)`.

EINVAL

The argument `len` has a value less than or equal to 0.

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.

RETURN VALUE

`mincore()` returns 0 on success, -1 on failure.

SEE ALSO

`mmap(2)`, `mlock(3C)`, `sysconf(3C)`, `unistd(4)`.

mkdir(2)**NAME**

`mkdir` - make a directory

SYNOPSIS

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

DESCRIPTION

`mkdir()` creates a new directory named by the pathname pointed to by *path*. The mode of the new directory is initialized from *mode* [see [chmod\(2\)](#) for values of mode]. The protection part of the *mode* argument is modified by the process' file creation mask [see [umask\(2\)](#)].

The directory's owner ID is set to the process' effective user ID. The directory's group ID is set to the process' effective group ID, or if the `S_ISGID` bit is set in the parent directory, then the group ID of the directory is inherited from the parent. The `S_ISGID` bit of the new directory is inherited from the parent directory.

If *path* is a symbolic link, it is not followed.

The newly created directory is empty with the exception of entries for itself and its parent directory.

Upon successful completion, `mkdir()` marks for update the `st_atime`, `st_ctime` and `st_mtime` fields of the directory. Also, the `st_ctime` and `st_mtime` fields of the directory that contains the new entry are marked for update.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`mkdir()` fails and creates no directory if one or more of the following are true:

EACCES

Either a component of the path prefix denies search permission, or write permission is denied on the parent directory of the directory to be created.

EEXIST

The named file already exists.

EFAULT

path points outside the allocated address space of the process.

EIO An I/O error has occurred while accessing the file system.

ELOOP

Too many symbolic links were encountered in translating *path*.

EMLINK

The maximum number of links to the parent directory would exceed `LINK_MAX`.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and the file system type does not allow it.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a *path* component exceeds `NAME_MAX`.

ENOENT

A component of the path prefix does not exist or is a null pathname.

ENOLINK

path points to a remote machine and the link to that machine is no longer active.

ENOSPC

No free space is available on the device containing the directory.

ENOTDIR

A component of the path prefix is not a directory.

EROFS

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

The `mkdir()` function may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

RESULT

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

SEE ALSO

`chmod(2)`, `mknod(2)`, `umask(2)`, `stat(5)`, `types(5)`.

NAME

mknod - make directory, special file, or ordinary file

SYNOPSIS

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

DESCRIPTION

mknod() creates a new file named by the pathname pointed to by *path*. The file type and permissions of the new file are initialized from *mode*.

The file type is specified in *mode* by the *S_IFMT* bits, which must be set to one of the following values:

```
S_IFIFO      fifo special
S_IFCHR      character-oriented file
S_IFDIR      directory
S_IFBLK      block-oriented file
S_IFREG      ordinary file
```

The only portable use of mknod() is to create a FIFO-special file. If *mode* is not *S_IFIFO* or *dev* is not 0, the behavior of mknod() is unspecified.

The permissions for the new file are OR-ed into the *mode* argument, and may be selected from any combination of the following symbolic constants:

```
S_ISUID  04000  Set user ID on execution.
S_ISGID  020#0  Set group ID on execution if # is 7, 5, 3, or 1. Enable file/record locking if
                # is 6, 4, 2, or 0
S_ISVTX  01000  Save text image after execution.
S_IRWXU  00700  Read, write, execute by owner.
S_IRUSR  00400  Read by owner.
S_IWUSR  00200  Write by owner.
S_IXUSR  00100  Execute or search by owner.
S_IRWXG  00070  Read, write, execute by group.
S_IRGRP  00040  Read by group.
S_IWGRP  00020  Write by group.
S_IXGRP  00010  Execute by group.
S_IRWXO  00007  Read, write, execute or search by others.
S_IROTH  00004  Read by others.
S_IWOTH  00002  Write by others
S_IXOTH  00001  Execute by others.
```

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process. However, if the `S_ISGID` bit is set in the parent directory, then the group ID of the file is inherited from the parent. If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs, the `S_ISGID` bit is cleared.

The access permission bits of *mode* are modified by the process' file mode creation mask: all bits set in the process' file mode creation mask are cleared [see `umask(2)`]. If *mode* indicates a block or character-oriented file, *dev* is a configuration-dependent specification of this file. If *mode* does not indicate a block or character-oriented device, *dev* is ignored.

Upon successful completion, `mknod()` 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.

Only a process with appropriate privileges may invoke `mknod()` for file types other than FIFO-special.

If *path* is a symbolic link, it is not followed.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`mknod()` fails and creates no new file if one or more of the following are true:

`EEXIST`

The named file exists.

`EINVAL`

dev is invalid.

`EFAULT`

path points outside the allocated address space of the process.

`ELOOP` Too many symbolic links were encountered in translating *path*.

`EMULTIHOP`

Components of *path* require hopping to multiple remote machines and the file system type does not allow it.

`ENAMETOOLONG`

The length of the *path* argument exceeds `PATH_MAX` or the length of a path component exceeds `NAME_MAX`.

`ENOTDIR`

A component of the path prefix is not a directory.

`ENOENT`

A component of the path prefix does not exist or is a null pathname.

`EROFS` The directory in which the file is to be created is located on a read-only file system.

`EINTR` A signal was caught during the `mknod()` system call.

`ENOLINK`

path points to a remote machine and the link to that machine is no longer active.

`EPERM` The invoking process does not have appropriate privileges and the file type is not FIFO-special.

`EACCES`

A component of the path prefix denies search permission, or write permission is denied on the parent directory.

`EIO` An I/O error occurred while accessing the file system.

`ENOSPC`

The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.

The `mknod()` function may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

RESULT

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

SEE ALSO

`mkdir(1)`, `chmod(2)`, `exec(2)`, `umask(2)`, `mkfifo(3C)`, `fs(4)`, `stat(5)`, `types(5)`.

NAME

mmap, mmap64 - map pages of memory

SYNOPSIS

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

void *mmap(void *addr, size_t len, int prot, int flags,
           int fildes, off_t off);

void *mmap64(void *addr, size_t len, int prot, int flags,
             int fd, off64_t offset);
```

DESCRIPTION

The `mmap()` function establishes a mapping between a process' address space and a file. The format of the call is as follows:

```
pa=mmap(addr, len, prot, flags, fildes, off);
```

The `mmap()` function establishes a mapping between the process' address space at an address *pa* for *len* bytes and the file associated with the file descriptor *fildes* at offset *off* for *len* bytes. The value of *pa* is an unspecified function of the argument *addr* and values of *flags*, further described below. A successful `mmap()` call returns *pa* as its result. The address ranges covered by [*pa*, *pa* + *len*) and [*off*, *off* + *len*) must be legitimate for the possible (not necessarily current) address space of a process and the file, respectively.

If the size of the mapped file changes after the call to `mmap()` as a result of some other operation on the mapped file, the effect of references to portions of the mapped region that correspond to added or removed portions of the file is unspecified.

The `mmap()` function is supported for regular files. Support for any other type of file is unspecified.

The *prot* argument determines whether read, write, execute, or some combination of accesses are permitted to the pages being mapped. The protection options are defined in `<sys/mman.h>`:

```
PROT_READ
    Page can be read.

PROT_WRITE
    Page can be written.

PROT_EXEC
    Page can be executed.

PROT_NONE
    Page can not be accessed.
```

Implementations need not enforce all combinations of access permissions. However, writes shall only be permitted when `PROT_WRITE` has been set.

The *flags* argument provides other information about the handling of the mapped pages. The options are defined in `<sys/mman.h>`:

```
MAP_SHARED
    Share changes.

MAP_PRIVATE
    Changes are private.

MAP_FIXED
    Interpret addr exactly.
```

The `MAP_PRIVATE` and `MAP_SHARED` flags control the visibility of write references to the memory region. Exactly one of these flags must be specified. The mapping type is retained across a `fork()`.

If `MAP_SHARED` is set in *flags*, write references to the memory region by the calling process may change the

file and are visible in all `MAP_SHARED` mappings of the same portion of the file by any process.

If `MAP_PRIVATE` is set in *flags*, write references to the memory region by the calling process do not change the file and are not visible to any process in other mappings of the same portion of the file.

It is unspecified whether write references by processes that have mapped the memory region using `MAP_SHARED` are visible to processes that have mapped the same portion of the file using `MAP_PRIVATE`.

It is also unspecified whether write references to a memory region mapped with `MAP_SHARED` are visible to processes reading the file and whether writes to a file are visible to processes that have mapped the modified portion of that file, except for the effect of `msync()`.

When `MAP_FIXED` is set in the *flags* argument, the implementation is informed that the value of *pa* must be *addr*, exactly. If `MAP_FIXED` is set, `mmap()` may return `(void *)-1` and set `errno` to `EINVAL`. If a `MAP_FIXED` request is successful, the mapping established by `mmap()` replaces any previous mappings for the process' pages in the range $[pa, pa + len)$.

When `MAP_FIXED` is not set, the implementation uses *addr* in an unspecified manner to arrive at *pa*. The *pa* so chosen will be an area of the address space which the implementation deems suitable for a mapping of *len* bytes to the file. All implementations interpret an *addr* value of 0 as granting the implementation complete freedom in selecting *pa*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for *pa*, it never places a mapping at address 0, nor does it replace any extant mapping, nor map into dynamic memory allocation areas.

The *off* argument is constrained to be aligned and sized according to the value returned by `sysconf()` when passed `_SC_PAGESIZE` or `_SC_PAGE_SIZE`. When `MAP_FIXED` is specified, the argument *addr* must also meet these constraints. The implementation performs mapping operations over whole pages. Thus, while the argument *len* need not meet a size or alignment constraint, the implementation will include, in any mapping operation, any partial page specified by the range $[pa, \lceil pa \rceil + \lceil len \rceil)$.

The implementation always zero-fills any partial page at the end of the file. Further, the implementation never writes out any modified portions of the last page of a file that are beyond the end of the file. If the mapping established by `mmap()` extends into pages beyond the page containing the last byte of the file, an application reference to any of the pages in the mapping that are beyond the last page results in the delivery of a `SIGBUS` or `SIGSEGV` signal.

The `mmap()` function adds an extra reference to the file associated with the file descriptor *fd* which is not removed by a subsequent `close()` on that file descriptor. This reference is removed when there are no more mappings to the file.

The `st_atime` field of the mapped file may be marked for update at any time between the `mmap()` call and the corresponding `munmap()` call. The initial read or write reference to a mapped region will cause the file's `st_atime` field to be marked for update if it has not already been marked for update.

The `st_ctime` and `st_mtime` fields of a file that is mapped with `MAP_SHARED` and `PROT_WRITE`, will be marked for update at some point in the interval between a write reference to the mapped region and the next call to `msync()` with `MS_ASYNC` or `MS_SYNC` for that portion of the file by any process. If there is no such call, these fields may be marked for update at any time after a write reference if the underlying file is modified as a result.

There may be implementation-dependent limits on the number of memory regions that can be mapped (per process or per system). If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of `shmat()` is implementation-dependent.

There is no functional difference between `mmap()` and `mmap64()`, except for the interpretation of `off64_t` [see [lfs\(5\)](#)].

RETURN VALUE

Upon successful completion, the `mmap()` function returns the address at which the mapping was placed (*pa*); otherwise, it returns a value of `MAP_FAILED` and sets `errno` to indicate the error. The symbol `MAP_FAILED` is defined in the header `<sys/mman.h>`. No successful return from `mmap()` will return the value

MAP_FAILED.

If `mmap()` fails for reasons other than `EBADF`, `EINVAL` or `ENOTSUP`, some of the mappings in the address range starting at *addr* and continuing for *len* bytes may have been unmapped.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `mmap()` function will fail if:

EAGAIN

The mapping could not be locked in memory.

EBADF

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

EACCES

The *fildes* argument is not open for read, regardless of the protection specified, or *fildes* is not open for write and `PROT_WRITE` was specified for a `MAP_SHARED` type mapping.

ENXIO

Addresses in the range [*off*, *off* + *len*) are invalid for *fildes*.

ENXIO

`MAP_FIXED` was specified in *flags* and the combination of *addr*, *len* and *off* is invalid for the object specified by *fildes*.

EINVAL

The *addr* argument (if `MAP_FIXED` was specified) or *off* is not a multiple of the page size as returned by `sysconf()`, or are considered invalid by the implementation.

EINVAL

The value of *flags* is invalid (neither `MAP_PRIVATE` nor `MAP_SHARED` is set).

EINVAL

This executable is a specially preloaded binary. The argument *addr* lies illegally between the text and data regions.

EMFILE

The number of mapped regions would exceed an implementation-dependent limit (per process or per system).

ENODEV

The *fildes* argument refers to a file whose type is not supported by `mmap()`.

ENOMEM

`MAP_FIXED` was specified and the range [*addr*, *addr* + *len*) exceeds that allowed for the address space of a process, or `MAP_FIXED` was not specified and there is insufficient memory space within the address space to effect the mapping.

ENOTSUP

The implementation does not support the combination of accesses requested in the *prot* argument.

EOVERFLOW

The file is a regular file and the value of *off* plus *len* exceeds the offset maximum established in the open file description associated with *fildes*.

NOTES

Use of `mmap()` may reduce the amount of memory available to other memory allocation functions.

Use of `MAP_FIXED` may result in unspecified behavior in further use of `brk()`, `sbrk()`, `malloc()` and `shmat()`. The use of `MAP_FIXED` is discouraged, as it may prevent an implementation from making the most effective use of resources.

The application must ensure correct synchronization when using `mmap()` in conjunction with any other file access method, such as `read()` and `write()`, standard input/output, and `shmat()`.

The `mmap()` function allows access to resources via address space manipulations, instead of `read()/write()`. Once a file is mapped, all a process has to do to access it is use the data at the address to which the file was mapped. So, using pseudo-code to illustrate the way in which an existing program might be changed to use `mmap()`, the following:

```
fildes = open(...)
lseek(fildes, some_offset)
read(fildes, buf, len)
/* use data in buf */
```

becomes:

```
fildes = open(...)
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)
/* use data at address */
```

For consistency with `read()` and `write()`, the `mmap()` function will fail when the request extends beyond the offset maximum.

SEE ALSO

`exec(2)`, `fcntl(2)`, `fork(2)`, `mprotect(2)`, `munmap(2)`, `plock(2)`, `shmat(2)`, `lockf(3C)`, `mlockall(3C)`, `sysconf(3C)`, `lfs(5)`, `mman(5)`, `preload(8)`.

NAME

mount - mount a file system

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mount.h>
int mount(const char *spec, const char *dir, int mflag,
          /* int fstyp, const char *dataptr, size_t datalen */);
```

DESCRIPTION

mount() mounts a removable file system contained in the block special file identified by *spec* in the directory identified by *dir*. *spec* and *dir* are pointers to pathnames. The argument *fstyp* is interpreted by mount() when either MS_FSS or MS_DATA is set in *mflag*. *fstyp* is the file system type number or a pointer to a string containing the file system type. The `sysfs(2)` system call can be used to determine the file system type number. If neither MS_DATA or MS_FSS is set in *mflag*, mount() defaults to the root file system type.

If the MS_DATA flag is set in *mflag* the system expects the *dataptr* and *datalen* arguments to be present. Together they describe a block of file-system specific data at address *dataptr* of length *datalen*. This is interpreted by file-system specific code within the operating system and its format depends on the file system type. If a particular file system type does not require this data, *dataptr* and *datalen* should both be zero. If the file system is to be mounted as read-only, MS_RDONLY must be set in *mflag*.

After a successful call to mount(), the name in *dir* refers to the root directory on the newly mounted file system.

mount() may be invoked only by the superuser. The interface is intended for use only by the `mount(1M)` utility.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

mount() fails if one or more of the following apply:

EBUSY

dir is currently mounted on, is someone's current working directory, or is otherwise busy.

EBUSY

The special file associated with *spec* is currently mounted.

EBUSY

There are no more mount table entries.

EFAULT

spec, *dir*, or *datalen* points outside the allocated address space of the process.

EINVAL

The super block has an invalid magic number or the *fstyp* is invalid.

ELOOP

Too many symbolic links were encountered in compiling *dir*.

ENAMETOOLONG

The length of the *dir* argument exceeds PATH_MAX or NAME_MAX.

ENOENT

One of the named files does not exist.

ENOTDIR

A component of a path prefix is not a directory.

EPERM

The effective user ID is not superuser.

EREMOTE

spec is remote and cannot be mounted.

ENOLINK

dir points to a remote machine and the link to that machine is no longer active.

EMULTIHOP

Components of *dir* require hopping to multiple remote machines.

ENOTBLK

spec is not a block special file.

ENXIO

The special file associated with *spec* does not exist.

ENOTDIR

dir is not a directory.

EROFS

spec is write protected and *mflag* requests write permission.

ENOSPC

The file system state in the super-block is not *FSOKAY* and *mflag* requests write permission.

RETURN VALUE

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

SEE ALSO

`mount(1M)`, `sysfs(2)`, `umount(2)`, `fs(4)`, `types(5)`.

mprotect(2)**NAME**

`mprotect` - amend protection of memory mapping

SYNOPSIS

```
#include <sys/mman.h>
int mprotect(void *addr, size_t len, int prot);
```

DESCRIPTION

The function `mprotect()` changes the access protections to be that specified by *prot* for those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes. The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The *prot* argument should be either `PROT_NONE` (page can not be accessed) or the bitwise inclusive OR of one or more of `PROT_READ` (page can be read), `PROT_WRITE` (page can be written) and `PROT_EXEC` (page can be executed).

If an implementation cannot support the combination of access types specified by *prot*, the call to `mprotect()` fails.

An implementation may permit accesses other than those specified by *prot*; however, no implementation permits a write to succeed where `PROT_WRITE` has not been set or permits any access where `PROT_NONE` alone has been set. Implementations will support at least the following values of *prot*: `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, and the inclusive OR of `PROT_READ` and `PROT_WRITE`. If `PROT_WRITE` is specified, the application must have opened the mapped objects in the specified address range with write permission, unless `MAP_PRIVATE` was specified in the original mapping, regardless of whether the file descriptors used to map the objects have since been closed.

The implementation will require that *addr* be a multiple of the page size as returned by `sysconf()`.

The behaviour of this function is unspecified if the mapping was not established by a call to `mmap()`.

When `mprotect()` fails for reasons other than `EINVAL`, the protections on some of the pages in the range [*addr*, *addr* + *len*) may have been changed.

RETURN VALUE

Upon successful completion the function `mprotect()` returns a value of 0; otherwise, it returns a value of -1 and sets `errno` to indicate an error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Under the following conditions, the function `mprotect()` fails and sets `errno` to:

EACCES

if *prot* specifies a protection that harms the access permission the process has to the underlying memory object.

EAGAIN

if *prot* specifies `PROT_WRITE` over a `MAP_PRIVATE` mapping and there are insufficient memory resources to reserve for locking the private page.

EINVAL

if *addr* is not a multiple of the page size as returned by `sysconf()`.

EINVAL

if the argument *len* has a value less than or equal to 0.

ENOMEM

if 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.

ENOMEM

The *prot* argument specifies `PROT_WRITE` on a `MAP_PRIVATE` mapping, and it would require more space than the system is able to supply for locking the private pages, if required.

ENOTSUP

The implementation does not support the combination of accesses requested in the *prot* argument.

SEE ALSO

`memcntl(2)`, `mmap(2)`, `plock(2)`, `mlock(3C)`, `mlockall(3C)`, `sysconf(3C)`, `mman(5)`.

NAME

msgctl - provide message control operations

SYNOPSIS

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

DESCRIPTION

msgctl() provides message control operations as specified by *cmd*. The following *cmd* control operations are available:

IPC_STAT

Places the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in the introduction to the function descriptions.

IPC_SET

Sets the value of the following members of the data structure associated with *msqid* to the corresponding value from the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only least significant 9 bits */
msg_qbytes
```

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

IPC_RMID

Removes the message queue identifier specified by *msqid* from the system and destroys the message queue and data structure associated with it. This function can only be executed by a process that has an effective user ID equal to either that of superuser, or to the value of *msg_perm.cuid* or *msg_perm.uid* in the data structure associated with *msqid*.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

msgctl() fails if one or more of the following apply:

EACCES

cmd is IPC_STAT and operation permission is denied to the calling process.

EFAULT

buf points to an illegal address.

EINVAL

msqid is not a valid message queue identifier.

EINVAL

cmd is not a valid control operation.

EINVAL

cmd is IPC_SET and *msg_perm.uid* or *msg_perm.gid* is not valid.

EOVERFLOW

cmd is `IPC_STAT` and *uid* or *gid* is too large to be stored in the structure pointed to by *buf*.

EPERM

cmd is `IPC_RMID` or `IPC_SET`. The effective user ID of the calling process is not that of superuser, or the value of `msg_perm.cuid` or `msg_perm.uid` in the data structure associated with *msqid*.

EPERM

cmd is `IPC_SET`, an attempt is being made to increase to the value of `msg_qbytes`, and the effective user ID of the calling process is not that of superuser.

SEE ALSO

`msgget(2)`, `msgop(2)`, `msg(5)`.

msgget(2)**NAME**

`msgget` - get identifier of message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

DESCRIPTION

`msgget()` returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure are created for *key* if one of the following applies:

- *key* is `IPC_PRIVATE`.
- *key* does not already have a message queue identifier associated with it, and $(msgflg \& IPC\_CREAT)$ is true.

On creation, the data structure associated with the new message queue identifier is initialized as follows:

- `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid`, and `msg_perm.gid` are set to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of `msg_perm.mode` are set to the low-order 9 bits of *msgflg*.
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are set to 0.
- `msg_ctime` is set to the current time.
- `msg_qbytes` is set to the system limit.

RETURN VALUE

Upon successful completion, a non-negative integer, namely a message queue identifier, is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`msgget()` fails if one or more of the following apply:

EACCES

A message queue identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *msgflg* is not granted.

ENOENT

A message queue identifier does not exist for *key* and $(msgflg \& IPC\_CREAT)$ is false.

ENOSPC

A message queue identifier is to be created but the system-imposed limit on the maximum number of message queue identifiers allowed in the system will be exceeded.

EEXIST

A message queue identifier exists for *key* but $(msgflg \& IPC\_CREAT)$ and $(msgflg \& IPC\_EXCL)$ are both true.

SEE ALSO

[msgctl\(2\)](#), [msgop\(2\)](#), [stdipc\(3C\)](#), [msg\(5\)](#).

NAME

msgop:
msgsnd, msgrcv - send/receive message

SYNOPSIS

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz, int
msgflg);
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp,
int msgflg);
```

DESCRIPTION

msgsnd() sends a message to the queue specified by the identifier *msqid*.

The argument *msgp* points to a user-defined buffer that must contain first a field of type `long int` that will specify the type of the message, and then a data portion that will hold the text of the message. The following members can be in a user defined buffer.

```
long int mtype;      /* message type */
char      mtext[1]; /* message text */
```

The structure member *mtype* is a positive long integer that can be used by the receiving process for message selection.

The structure member *mtext* is any text of length *msgsz* bytes. *msgsz* can range from 0 to a system imposed maximum.

The argument *msgflg* specifies the action to be taken if one or more of the following apply:

- The number of bytes already on the queue is equal to `msg_qbytes`.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (*msgflg* & `IPC_NOWAIT`) is true, the message is not sent and the calling process returns immediately.
- If (*msgflg* & `IPC_NOWAIT`) is false, the calling process suspends execution until one of the following occurs:
 - The condition responsible for the suspension no longer exists, in which case the message is sent.
 - *msqid* is removed from the system [see `msgctl(2)`]. When this occurs, `errno` is set to `EIDRM`, and a value of `-1` is returned.
 - The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in `signal(2)`.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*.

- `msg_qnum` is incremented by 1.
- `msg_lspid` is set to the process ID of the calling process.
- `msg_stime` is set to the current time.

msgrcv() reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the user defined structure pointed to by *msgp*.

The argument *msgp* points to a user-defined buffer that must contain first a field of type `long int` that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The following members can be in a user defined buffer.

```
long int mtype;      /* message type */
```



```
char    mtext[1]; /* message text */
```

The structure member `mtype` is the received message's type as specified by the sending process.

The structure member `mtext` is the text of the message.

The argument `msgsz` specifies the size in bytes of `mtext`. The received message is truncated to `msgsz` bytes if it is larger than `msgsz` and `(msgflg & MSG_NOERROR)` is non-zero. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

If the value of `msgsz` is greater than `SSIZE_MAX`, the result is implementation-dependent.

The argument `msgtyp` specifies the type of message requested as follows:

- If `msgtyp` is 0, the first message on the queue is received.
- If `msgtyp` is greater than 0, the first message of type `msgtyp` is received.
- If `msgtyp` is less than 0, the first message of the lowest type that is less than or equal to the absolute value of `msgtyp` is received.

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

- If `(msgflg & IPC_NOWAIT)` is true, the calling process returns immediately with a result of `-1` and sets `errno` to `ENOMSG`.
- If `(msgflg & IPC_NOWAIT)` is false, the calling process suspends execution until one of the following occurs:
 - A message of the desired type is placed on the queue.
 - `msqid` is removed from the system. When this occurs, `errno` is set to `EIDRM`, and a value of `-1` is returned.
 - The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in [signal\(2\)](#).

Upon successful completion, the following actions are taken with respect to the data structure associated with `msqid`.

- `msg_qnum` is decremented by 1.
- `msg_lrp_id` is set to the process ID of the calling process.
- `msg_rtime` is set to the current time.

RETURN VALUE

Upon successful completion, `msgsnd()` returns 0. Otherwise, no message will be sent, `msgsnd()` will return `-1` and `errno` will be set to indicate the error.

Upon successful completion, `msgrcv()` returns a value equal to the number of bytes actually placed into the buffer `mtext`. Otherwise, no message will be received, `msgrcv()` will return `(ssize_t)-1` and `errno` will be set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `msgsnd()` function will fail if:

EACCES

Operation permission is denied to the calling process.

EAGAIN

The message cannot be sent for one of the reasons cited above and `(msgflg & IPC_NOWAIT)` is true.

EFAULT

`msgp` points to an illegal address.

EIDRM

The message queue identifier `msqid` is removed from the system.

EINTR

The `msgsnd()` function was interrupted by a signal.

EINVAL

The value of *msqid* is not a valid message queue identifier, or the value of *mtype* is less than 1; or the value of *msgsz* is less than 0 or greater than the system-imposed limit.

The `msgrcv()` function will fail if:

E2BIG

The length of *mtext* is greater than *msgsz* and (*msgflg* & `MSG_NOERROR`) is false.

EACCES

Operation permission is denied to the calling process.

EFAULT

msgp points to an illegal address.

EINVAL

msqid is not a valid message queue identifier; or the value of *msgsz* is less than 0.

EIDRM

The message queue identifier *msqid* is removed from the system.

EINTR

The `msgsnd()` function was interrupted by a signal.

ENOMSG

The queue does not contain a message of the desired type and (*msgtyp* & `IPC_NOWAIT`) is true.

SEE ALSO

`msgctl(2)`, `msgget(2)`, `signal(2)`, `msg(5)`.

NAME

`munmap` - terminate mapping of memory pages

SYNOPSIS

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

DESCRIPTION

The function `munmap()` removes any mappings for those entire pages containing any part of the address space of the process starting at `addr` and continuing for `len` bytes. Further references to these pages result in the generation of a `SIGSEGV` signal to the process. If there are no mappings in the specified address range, then `munmap()` has no effect.

The implementation will require that `addr` be a multiple of the page size `PAGESIZE`.

If a mapping to be removed was private, any modifications made in this address range will be discarded.

The behaviour of this function is unspecified if the mapping was not established by a call to `mmap()`.

RETURN VALUE

Upon successful completion, the function `munmap()` returns a value of 0; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Under the following conditions, the function `munmap()` fails and sets `errno` to:

`EINVAL`

if addresses in the range `[addr, addr + len)` are outside the valid range for the address space of a process.

`EINVAL`

if the argument `len` has a value less than or equal to 0.

`EINVAL`

if `addr` is not a multiple of the page size as returned by `sysconf()`.

SEE ALSO

[mmap\(2\)](#), [sysconf\(3C\)](#), [mman\(5\)](#).

mutex(2)**NAME**

mutex - ABI entry points for mutual exclusion

SYNOPSIS

```
cc [ flag ... ] file ... -lmutex [ library ... ]
#include <abi_mutex.h>
void init_lock(abilock_t *lck);
int acquire_lock(abilock_t *lck);
void release_lock(abilock_t *lck);
int stat_lock(abilock_t *lck);
```

DESCRIPTION

These functions supply a user-level interface to mutual exclusion locks. The parameter *lck* must point to memory shared by all processes wishing to acquire or test the lock.

The contents of the structure `abilock_t` are as follows:

```
typedef struct {
    volatile unsigned long    abi_lock;
} abilock_t;
```

The function `init_lock` must be called on a lock before any of the other functions. It initializes the lock to an unlocked state.

`acquire_lock` attempts to acquire the lock referenced by *lck*. It returns zero if the lock was acquired, otherwise non-zero.

`stat_lock` returns the current state of the lock referenced by *lck* without attempting to acquire the lock. It returns zero if the lock is free, otherwise non-zero.

`release_lock` unconditionally releases the lock pointed to by *lck*.

NOTES

No locks are ever freed automatically by the system. Locks acquired by a process that dies remain locked.

The user is responsible for implementing a back-off policy if `acquire_lock` returns failure.

Even though `stat_lock` returns a status indicating that the lock is available, a call to `acquire_lock` could still fail.

The contents of the lock pointed to by *lck* should not be examined directly.

On RM400-5 systems, these functions are available only to the superuser or to users with the privilege to use preemption control [see `pfenable(1M)`]. On these systems, the loader flags `-lmproc` and `-lxt` must be specified in addition to `-lmutex`.

nice(2) - rmdir(2)**nice(2)****NAME**

nice - change priority of a time-sharing process

SYNOPSIS

```
#include <unistd.h>
int nice(int incr);
```

DESCRIPTION

`nice()` allows a process in the time-sharing scheduling class to change its priority. The `prctl()` system call is a more general interface to scheduler functions.

`nice()` adds the value of *incr* to the *nice* value of the calling process. A process' *nice* value is a non-negative number for which a more positive value results in lower CPU priority.

A maximum *nice* value is 39 and the minimum is 0 (the default value is 20). Calls using values above or below these limits result in the highest value being used. Only a process with appropriate privileges can lower the *nice* value.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `nice()` function will fail if:

EPERM

`nice()` fails and does not change the *nice* value if *incr* is negative or greater than 39 and the effective user ID of the calling process is not superuser.

EINVAL

`nice()` fails if called by a process in a scheduling class other than time-sharing.

RESULT

Upon successful completion, `nice()` returns the new value minus 20. Otherwise, a value of `-1` is returned, the process' *nice* value is not changed, and `errno` is set to indicate the error.

SEE ALSO

[nice\(1\)](#), [exec\(2\)](#), [prctl\(2\)](#), [unistd\(4\)](#).

NAME

`open`, `open64` - open file for reading or writing

SYNOPSIS

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

int open(const char *path, int oflag, ... /* mode_t mode */);
int open64(const char *path, int oflag, ...);
```

DESCRIPTION

The `open()` function establishes the connection between a file and a file descriptor. It creates an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The *path* argument points to a pathname naming the file.

The `open()` function will return a file descriptor for the named file that is the lowest 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 process in the system. The `FD_CLOEXEC` file descriptor flag associated with the new file descriptor will be cleared.

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

The file status flags and file access modes of the open file description will be 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`. Applications must specify exactly one of the first three values (file access modes) below in the value of *oflag*:

`O_RDONLY`
Open for reading only

`O_WRONLY`
Open for writing only

`O_RDWR`
Open for reading and writing.

Any combination of the following may be used:

`O_NONBLOCK`
Affect subsequent reads and writes [see `read(2)` and `write(2)`].

When a FIFO file is opened with `O_RDONLY` or `O_WRONLY`, this has the following effect:

If `O_NONBLOCK` is set:

An `open()` for reading-only will return without delay; an `open()` for writing-only will return an error if no process currently has the file open for reading.

If `O_NONBLOCK` is clear:

An `open()` for reading-only will block until a process opens the file for writing; an `open()` for writing-only will block until a process opens the file for reading.

When a file associated with a terminal line is opened, this has the following effect:

If `O_NONBLOCK` is set:

`open()` will return without waiting for the device to be ready or available; subsequent behavior of the device is device specific.

If `O_NONBLOCK` is clear:

`open()` will block until the device is ready or available.

`O_APPEND`

The file pointer will be set to the end of the file prior to each write.

`O_SYNC`

When opening a regular file, subsequent writes are affected. Each `write(2)` will wait for both the file data and file status to be physically updated.

`O_NOCTTY`

If the file is assigned to a terminal, the terminal will not be allocated as the calling process' controlling terminal.

`O_CREAT`

If the file exists, this flag has no effect except as noted under `O_EXCL` below. Otherwise, the file is created; the user ID of the file is set to the effective user ID of the process; the group ID of the file is set to the group ID of the file's parent directory or to the effective group ID of the process; and the access permission bits [see `stat(5)`] of the file mode are set to the value of the third argument taken as type `mode_t` modified as follows: a bitwise-AND is performed on the file-mode bits and the corresponding bits in the complement of the process' file mode creation mask. Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared. When bits other than the file permission bits are set, the effect is unspecified. The third argument does not affect whether the file is open for reading, writing or for both.

`O_TRUNC`

If the file exists and is a regular file, and the file is successfully opened `O_RDWR` or `O_WRONLY`, its length is truncated to 0 and the mode and owner are unchanged. It will have no effect on FIFO special files or terminal device files. Its effect on other file types is implementation-dependent. The result of using `O_TRUNC` with `O_RDONLY` is undefined.

`O_EXCL`

If `O_EXCL` and `O_CREAT` are set, `open()` will fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other processes executing `open()` naming the same filename in the same directory with `O_EXCL` and `O_CREAT` set.

The following additional value may be used in constructing *oflag*:

`O_LARGEFILE`

If set, the offset maximum in the open file description will be the largest value that can be represented correctly in an object of type `off64_t`.

When opening a STREAMS file, *oflag* may be constructed from `O_NONBLOCK`, OR-ed with `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. Other flag values are not applicable to STREAMS devices and have no effect on them. The value of `O_NONBLOCK` affects the operation of STREAMS drivers and certain system calls [see `read(2)`, `getmsg(2)`, `putmsg(2)`, and `write(2)`]. For drivers, the implementation of `O_NONBLOCK` is device-specific. Each STREAMS device driver may treat these options differently.

If *path* names the master side of a pseudo-terminal device, then it is unspecified whether `open()` locks the slave side so that it cannot be opened. Portable applications must call `unlockpt()` before opening the slave side.

When `open()` is invoked to open a named stream, and the `connld` module [see `connld(7)`] has been pushed on the pipe, `open()` blocks until the server process has issued an `I_RECVFD` ioctl [see `streamio(7)`] to receive the file descriptor.

If *path* is a symbolic link and `O_CREAT` and `O_EXCL` are set, the link is not followed.

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

The new file descriptor is the lowest numbered file descriptor available and is set to remain open across `exec()` system calls [see `fcntl(2)`].

Certain flag values can be set following `open()` as described in `fcntl(2)`.

If `O_CREAT` is set and the file did not previously exist, upon successful completion `open()` marks 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 `O_TRUNC` is set and the file did previously exist, upon successful completion `open()` marks for update the

`st_ctime` and `st_mtime` fields of the file.

Note that using `open64()` is equivalent to using `open()` with `O_LARGEFILE` set in *oflag*.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The named file is opened unless one or more of the following conditions is true:

EACCES

The file does not exist and write permission is denied by the parent directory of the file to be created.

EACCES

`O_TRUNC` is specified and write permission is denied.

EACCES

A component of the path prefix denies search permission.

EACCES

oflag permission is denied for an existing file.

EAGAIN

The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see [chmod\(2\)](#)].

EEXIST

`O_CREAT` and `O_EXCL` are set, and the named file exists.

EFAULT

path points outside the allocated address space of the process.

EINTR

A signal was caught during the `open()` system call.

EIO

A hangup or error occurred during the open of the STREAMS-based device.

EISDIR

The named file is a directory and *oflag* is write or read/write.

ELOOP

Too many symbolic links were encountered in translating *path*.

EMFILE

The process has too many open files [see [getrlimit\(2\)](#)].

EMULTIHOP

Components of *path* require hopping to multiple remote machines and the file system type does not allow it.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a path component exceeds `NAME_MAX`.

ENFILE

The system file table is full.

ENOENT

`O_CREAT` is not set and the named file does not exist.

ENOENT

`O_CREAT` is set and a component of the path prefix does not exist or is the null pathname.

ENOLINK

path points to a remote machine and the link to that machine is no longer active.

ENOSPC

`O_CREAT` and `O_EXCL` are set, and the file system is out of inodes.

ENOSPC

`O_CREAT` is set and the directory for the file cannot be extended.

ENOSR

Unable to allocate a stream.

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.

ENXIO

`O_NONBLOCK` is set, the named file is a FIFO, `O_WRONLY` is set, and no process has the file open for reading.

ENXIO

A STREAMS module or driver open routine failed.

EROFS

The named file resides on a read-only file system and either `O_WRONLY`, `O_RDWR`, `O_CREAT`, or `O_TRUNC` is set in *oflag* (if the file does not exist).

The `open()` function may fail if:

EAGAIN

The *path* argument names the slave side of a pseudo-terminal device that is locked.

EINVAL

The value of the *oflag* argument is not valid.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

ENOMEM

The *path* argument names a STREAMS file and the system is unable to allocate resources.

ETXTBSY

The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.

EOVERFLOW

The named file is a regular file, `O_LARGEFILE` is not set and the size of the file cannot be represented correctly in an object of type `off_t`.

RESULT

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

NOTES

An `open()` will fail if the size of the (regular) file cannot be represented correctly in an object of type `off_t`.

The size of file on which a program is able to operate is determined by the `off_t` in use for the `open()`. The open protection rule ensures that old binaries do not operate on files that are too large to handle correctly, and prevents the binaries from generating incorrect results or corrupting the data in the file.

An argument against open protection is that requiring opens to fail will break some binaries that would have worked perfectly well otherwise.

Another argument against open protection is that it is unnecessary because an error will be returned as soon as a function cannot return the correct result of an operation ("No Lies" rule). However, most programs check for the success of the `open()` call, but many do not check for overflow or error after `lseek()` and other calls. An audit of the standard utilities uncovered numerous examples.

An argument for open protection is that it increases the likelihood of an immediate and informative error message. The error message is likely to include the name of the file that could not be opened. It is much less likely that an `lseek()` error message will be as immediate or as informative. The delay in, or complete lack of, reporting such errors may result in "silent failure".

Another argument for open protection is that there are numerous plausible scenarios in which this rule avoids serious harm. It prevents typical implementations of the `touch` utility from truncating large files to 0 length. It can prevent silent failure, which has been demonstrated to occur in at least one commercial data management system. With open protection a commercial backup/restore system will report errors on files that might otherwise result in a corrupted backup tape. It prevents typical implementations of `dbm/ndbm` from returning incorrect results from a database whose size exceeds the `off_t` in use for the `dbm` routines.

SEE ALSO

`chmod(2)`, `close(2)`, `creat(2)`, `dup(2)`, `exec(2)`, `fcntl(2)`, `getrlimit(2)`, `lseek(2)`, `read(2)`, `getmsg(2)`, `putmsg(2)`, `stat(2)`, `umask(2)`, `write(2)`, `fcntl(5)`, `lfs(5)`, `stat(5)`, `types(5)`.

NAME

`pause` - suspend process until signal

SYNOPSIS

```
#include <unistd.h>
□
□
int pause(void);
```

DESCRIPTION

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

If the signal causes termination of the calling process, `pause()` does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function [see [signal\(2\)](#)], the calling process resumes execution from the point of suspension.

RESULT

Since the `pause()` function suspends a process in an undefined state until a signal is received, there is no successful completion return value. When the calling process receives a signal, `pause()` returns `-1` and `errno` is set to `EINTR`.

SEE ALSO

[alarm\(2\)](#), [kill\(2\)](#), [signal\(2\)](#), [sigpause\(2\)](#), [wait\(2\)](#).

NAME

pipe - create an interprocess communication channel

SYNOPSIS

```
#include <unistd.h>
int pipe(int fildes[2]);
```

DESCRIPTION

pipe() creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. The files associated with *fildes*[0] and *fildes*[1] are streams and are both opened for reading and writing. The `O_NDELAY` and `O_NONBLOCK` flags are cleared.

A read from *fildes*[0] accesses the data written to *fildes*[1] on a FIFO basis and a read from *fildes*[1] accesses the data written to *fildes*[0] also on a FIFO basis. It is unspecified whether *fildes*[0] is also open for writing and whether *fildes*[1] is also open for reading.

The `FD_CLOEXEC` flag will be clear on both file descriptors.

Upon successful completion pipe() marks for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the pipe.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

pipe() fails if:

EMFILE

If `OPEN_MAX-1` or more file descriptors are currently open for this process.

ENFILE

A file table entry could not be allocated.

RESULT

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

NOTES

Since a pipe is bi-directional, there are two separate flows of data. Therefore, the size (*st_size*) returned by a call to [fstat\(2\)](#) with argument *fildes*[0] or *fildes*[1] is the number of bytes available for reading from *fildes*[0] or *fildes*[1] respectively. Previously, the size (*st_size*) returned by a call to *fstat()* with argument *fildes*[1] (the write-end) was the number of bytes available for reading from *fildes*[0] (the read-end).

SEE ALSO

[sh\(1\)](#), [fcntl\(2\)](#), [getmsg\(2\)](#), [poll\(2\)](#), [putmsg\(2\)](#), [read\(2\)](#), [write\(2\)](#), [unistd\(4\)](#), [streamio\(7\)](#).

NAME

plock - lock or unlock process, text, or data

SYNOPSIS

```
#include <sys/lock.h>
int plock(int op);
```

DESCRIPTION

plock() allows the calling process to lock into memory or unlock 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 superuser to use this call. plock() performs the function specified by *op* :

PROCLCK

Lock text and data segments into memory (process lock).

TXTLCK

Lock text segment into memory (text lock).

DATLOCK

Lock data segment into memory (data lock).

UNLOCK

Remove locks.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

plock() fails and does not perform the requested operation if one or more of the following apply:

EPERM

The effective user ID of the calling process is not superuser.

EINVAL

op is equal to PROCLCK and a process lock, a text lock, or a data lock already exists on the calling process.

EINVAL

op is equal to TXTLCK and a text lock, or a process lock already exists on the calling process.

EINVAL

op is equal to DATLOCK and a data lock, or a process lock already exists on the calling process.

EINVAL

op is equal to UNLOCK and no lock exists on the calling process.

EAGAIN

Not enough memory.

RETURN VALUE

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of -1 is returned and [errno](#) is set to indicate the error.

NOTES

memcntl() is the preferred interface to process locking.

SEE ALSO

[exec\(2\)](#), [exit\(2\)](#), [fork\(2\)](#), [memcntl\(2\)](#).

NAME

`poll` - multiplex STREAMS input/output

SYNOPSIS

```
#include <stropts.h>
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

DESCRIPTION

`poll()` provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open files. `poll()` identifies those files on which a user can send or receive messages, or on which certain events have occurred.

`fds` specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one element for each open file descriptor of interest. The array's elements are `pollfd` structures, which contain the following members:

```
int fd;           /* file descriptor */
short events;     /* requested events */
short revents;    /* returned events */
```

`fd` specifies an open file descriptor and `events` and `revents` are bitmasks constructed by an OR of any combination of the following event flags:

POLLIN

Data other than high priority data may be read without blocking. For STREAMS, this flag is set even if the message is of zero length.

POLLRDNORM

Normal data (priority band = 0) may be read without blocking. For STREAMS, this flag is set in `revents` even if the message is of zero length.

POLLRDBAND

Data from a non-zero priority band may be read without blocking. For STREAMS, this flag is set in `revents` even if the message is of zero length.

POLLPRI

High priority data may be received without blocking. For STREAMS, this flag is set in `revents` even if the message is of zero length.

POLLOUT

Normal data may be written without blocking.

This option is also set for BSD sockets when an error has occurred.

POLLWRNORM

The same as **POLLOUT**.

POLLWRBAND

Priority data (priority band > 0) may be written. This event only examines bands that have been written to at least once.

POLLMSG

An `M_SIG` or `M_PCSIG` message containing the `ASIGPOLL` signal has reached the front of the stream head read queue.

POLLERR

An error has occurred on the device or stream. This flag is only valid in the `revents` bitmask; it is not used in the `events` field.

When errors occur, **POLLOUT** is set for BSD sockets instead of this option.

POLLHUP

A hangup has occurred on the stream. This event and `POLLOUT` are mutually exclusive; a stream can never be writable if a hangup has occurred. However, this event and `POLLIN`, `POLLRDNORM`, `POLLRDBAND`, or `POLLPRI` are not mutually exclusive. This flag is only valid in the `revents` bitmask; it is not used in the `events` field.

POLLNVAL

The specified `fd` value does not belong to an open file. This flag is only valid in the `revents` field; it is not used in the `events` field.

If the value of `fd` is less than 0, `events` is ignored and `revents` is set to 0 in that entry on return from `poll()`.

In each `pollfd` structure, `poll()` clears the `revents` member except that where the application requested a report on a condition by setting one of the bits of `events` listed above, `poll()` sets the corresponding bit in `revents` if the requested condition is true. In addition, `poll()` sets the `POLLHUP`, `POLLERR`, and `POLLNVAL` flag in `revents` if the condition is true, even if the application did not set the corresponding bit in `events`.

If none of the defined events have occurred on any selected file descriptor, `poll()` waits at least *timeout* milliseconds for an event to occur on any of the selected file descriptors. If the value of *timeout* is 0, `poll()` returns immediately. If the value of *timeout* is -1, `poll()` blocks until a requested event occurs or until the call is interrupted.

Implementations may place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval will be rounded up to the next supported value.

The `poll()` function is not affected by the `O_NONBLOCK` flag.

The `poll()` function supports regular files, terminal and pseudo-terminal devices, STREAMS-based files, FIFOs, pipes and BSD sockets. The behavior of `poll()` on elements of *fds* that refer to other types of file is unspecified.

Regular files always poll TRUE for reading and writing.

RETURN VALUE

Upon successful completion, a non-negative value is returned. A positive value indicates the total number of file descriptors that has been selected (i.e., file descriptors for which the `revents` field is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`poll()` fails if one or more of the following apply:

EAGAIN

Allocation of internal data structures failed, but the request may be attempted again.

EFAULT

Some argument points to a memory area outside the allocated address space.

EINTR

A signal was caught during the `poll()` system call.

EINVAL

The *nfds* argument is greater than `OPEN_MAX`, or one of the `fd` members refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

SEE ALSO

[getmsg\(2\)](#), [getrlimit\(2\)](#), [putmsg\(2\)](#), [read\(2\)](#), [write\(2\)](#), [pollbunch\(3C\)](#), [poll\(5\)](#), [stropts\(5\)](#).

Programmer's Guide: STREAMS.

NAME

pread, pread64 - atomic position and read

SYNOPSIS

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

ssize_t pread(int fd, void *buf, size_t nbyte, off_t offset);
ssize_t pread64(int fd, void *buf, size_t nbyte, off64_t offset);
```

DESCRIPTION

The `pread()` system call does an atomic position-and-read, eliminating the necessity of using a locking mechanism when both operations are desired and file descriptors are shared. `pread()` is analogous to `read()` but takes a fourth argument, *offset*. The read is done as if an `lseek()` to *offset* (from the beginning of the file) were done first. Note that (though the semantics are analogous) an `lseek()` is not actually performed; the file pointer is not affected by `pread()`. The read of *nbytes* then starts at the specified offset.

The atomicity of `pread()` enables processes or threads that share file descriptors to read from a shared file at a particular offset without using a locking mechanism that would be necessary to achieve the same result in separate `lseek()` and `read()` system calls. Atomicity is required as the file pointer is shared and one thread might move the pointer using `lseek()` after another process completes an `lseek()` but prior to the `read()`.

There is no functional difference between `pread()` and `pread64()`, except for the interpretation of `off64_t` [see [lfs\(5\)](#)].

RETURN VALUE

Upon successful completion, `pread()` returns the number of bytes actually read and placed in *buf*. A value of 0 is returned when an end-of-file has been reached. Otherwise a `-1` and an error is returned.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `pread()` function will fail if:

EAGAIN

Mandatory file/record locking was set, `O_NONBLOCK` was set, and there was a blocking record lock.

EAGAIN

Total amount of system memory available when reading via raw I/O is temporarily insufficient.

EAGAIN

No data is waiting to be read on a file associated with a tty device and `O_NONBLOCK` was set.

EAGAIN

No message is waiting to be read on a stream and `O_NONBLOCK` was set.

EBADF

The *fd* argument is not a valid file descriptor open for reading.

EBADMSG

The file is a stream file that is set to control-normal mode and the message waiting to be read includes a control part.

EDEADLK

The `pread()` was going to go to sleep and cause a deadlock to occur.

EFAULT

buf points outside the allocated address space.

EINTR

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

EINVAL

The stream or multiplexer referenced by *fildev* is linked (directly or indirectly) downstream from a multiplexer.

EIO

A physical I/O error has occurred.

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. This error may also be generated for implementation-dependent reasons.

EISDIR

The *fildev* argument refers to a directory and the implementation does not allow the directory to be read using `pread()`. The `readdir()` function should be used instead.

ENOLCK

The system record lock table was full, so the `pread()` could not go to sleep until the blocking record lock was removed.

ENOLINK

fildev is on a remote machine and the link to that machine is no longer active.

EOVERFLOW

The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildev*.

The `pread()` function may fail if:

ENXIO

A request was made of a non-existent device, or the request was outside the capabilities of the device.

The `pread()` function will fail, and the file pointer remains unchanged, if:

EINVAL

The *offset* argument is invalid. The value is negative.

EOVERFLOW

The file is a regular file and an attempt was made to read or write at or beyond the offset maximum associated with the file.

ENXIO

A request was outside the capabilities of the device.

ESPIPE

fildev is associated with a pipe or FIFO.

A `pread()` from a STREAMS file also fails if an error message is received at the stream head. In this case, `errno` is set to the value returned in the error message. If a hangup occurs on the stream being read, `pread()` continues to operate normally until the stream head read queue is empty. Thereafter, it returns 0.

NOTES

`pread()` updates the time of last access [see `stat(2)`] of the file.

SEE ALSO

`lseek(2)`, `pwrite(2)`, `read(2)`, `unistd(4)`, `lfs(5)`, `types(5)`, `uio(5)`.

priocntl(2)**NAME**

priocntl - process scheduler control

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/procset.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <sys/fppriocntl.h>
long priocntl(idtype_t idtype, id_t id, int cmd, ... /* arg */);
```

DESCRIPTION

priocntl() provides for control over the scheduling of active processes.

Processes fall into distinct classes with a separate scheduling policy applied to each class. The three classes currently supported are the real-time class, the time-sharing class and the fixed-priority class. The characteristics of these classes are described under the corresponding headings below. The class attribute of a process is inherited across the [fork\(2\)](#) and [exec\(2\)](#) system calls. priocntl() can be used to dynamically change the class and other scheduling parameters associated with a running process or set of processes given the appropriate permissions as explained below.

In the default configuration, a runnable real-time process runs before any other process. Therefore, inappropriate use of real-time processes can have a dramatic negative impact on system performance.

priocntl() provides a interface for specifying a process or set of processes to which the system call is to apply. The priocntlset() system call provides the same functions as priocntl(), but allows a more general interface for specifying the set of processes to which the system call is to apply.

For priocntl(), the idtype and id arguments are used together to specify the set of processes. The interpretation of id depends on the value of idtype. The possible values for idtype and corresponding interpretations of id are as follows:

P_PID

id is a process ID specifying a single process to which the priocntl() system call is to apply.

P_PPID

id is a parent process ID. The priocntl() system call applies to all processes with the specified parent process ID.

P_PGID

id is a process group ID. The priocntl() system call applies to all processes in the specified process group.

P_SID

id is a session ID. The priocntl() system call applies to all processes in the specified session.

P_CID

id is a class ID (returned by priocntl() PC_GETCID as explained below). The priocntl() system call applies to all processes in the specified class.

P_UID

id is a user ID. The priocntl() system call applies to all processes with this effective user ID.

P_GID

id is a group ID. The priocntl() system call applies to all processes with this effective group ID.

P_ALL

The priocntl() system call applies to all existing processes. The value of id is ignored. The

permission restrictions described below still apply.

An *id* value of `P_MYID` can be used in conjunction with the *idtype* value to specify the calling process' process ID, parent process ID, process group ID, session ID, class ID, user ID, or group ID.

In order to change the scheduling parameters of a process (using the `PC_SETPARMS` command as explained below) the real or effective user ID of the process calling `prioctl()` must match the real or effective user ID of the receiving process or the effective user ID of the calling process must be superuser. These are the minimum permission requirements enforced for all classes. An individual class may impose additional permissions requirements when setting processes to that class and/or when setting class-specific scheduling parameters.

A special `sys` scheduling class exists for the purpose of scheduling the execution of certain special system processes (such as the swapper process). It is not possible to change the class of any process to `sys`. In addition, any processes in the `sys` class that are included in a specified set of processes are disregarded by `prioctl()`. For example, an *idtype* of `P_UID` and an *id* value of zero would specify all processes with a user ID of zero except processes in the `sys` class and (if changing the parameters using `PC_SETPARMS`) the `init` process.

The `init` process is a special case. In order for a `prioctl()` call to change the class or other scheduling parameters of the `init` process (process ID 1), it must be the only process specified by *idtype* and *id*. The `init` process may be assigned to any class configured on the system, but the time-sharing class is almost always the appropriate choice.

The data type and value of *arg* are specific to the type of command specified by *cmd*.

The following structure is used by the `PC_GETCID` and `PC_GETCLINFO` commands.

```
typedef struct {
    id_t    pc_cid;           /* Class id */
    char    pc_clname[PC_CLNMSZ]; /* Class name */
    long    pc_clinfo[PC_CLINFOSZ]; /* Class information */
} pcinfo_t;
```

`pc_cid` is a class ID returned by `prioctl()` `PC_GETCID`. `pc_clname` is a buffer of size `PC_CLNMSZ` (defined in `sys/prioctl.h`) used to hold the class name (RT for real-time, TS for time-sharing, or FP for fixed-priority).

`pc_clinfo` is a buffer of size `PC_CLINFOSZ` (defined in `sys/prioctl.h`) used to return data describing the attributes of a specific class. The format of this data is class-specific and is described below under the appropriate heading.

The following structure is used by the `PC_SETPARMS` and `PC_GETPARMS` commands.

```
typedef struct {
    id_t    pc_cid;           /* Process class */
    long    pc_clparms[PC_CLPARMSZ]; /* Class-specific params */
} pcparms_t;
```

`pc_cid` is a class ID (returned by `prioctl PC_GETCID`). The special class ID `PC_CLNULL` can also be assigned to `pc_cid` when using the `PC_GETPARMS` command as explained below.

The `pc_clparms` buffer holds class-specific scheduling parameters. The format of this parameter data for a particular class is described under the appropriate heading below. `PC_CLPARMSZ` is the length of the `pc_clparms` buffer and is defined in `sys/prioctl.h`.

Commands

`PC_GETCID`

Gets class ID and class attributes for a specific class with given class name. The *idtype* and *id* arguments are ignored. If *arg* is not `NULL`, it points to a structure of type `pcinfo_t`. The `pc_clname` buffer contains the name of the class whose attributes you are getting.

On success, the class ID is returned in `pc_cid`, the class attributes are returned in the `pc_clinfo` buffer, and the `prioctl()` call returns the total number of classes configured in the system (including the `sys` class). If the class specified by `pc_clname` is invalid or is not currently configured

the `priocntl()` call returns `-1` with `errno` set to `EINVAL`. The format of the attribute data returned for a given class is defined in the `sys/rtpriocntl.h`, `sys/tspriocntl.h`, or `sys/fppriocntl.h` header file and described under the appropriate heading below.

If `arg` is a `NULL` pointer, no attribute data is returned but the `priocntl()` call still returns the number of configured classes.

PC_GETCLINFO

Gets class name and class attributes for a given class ID. The `idtype` and `id` arguments are ignored. If `arg` is nonnull, it points to a structure of type `pcinfo_t`. `pc_cid` is the class ID of the class whose attributes you are getting.

On success, the class name is returned in the `pc_clname` buffer, the class attributes are returned in the `pc_clinfo` buffer, and the `priocntl()` call returns the total number of classes configured in the system (including the `sys` class). The format of the attribute data returned for a given class is defined in the `sys/rtpriocntl.h`, `sys/tspriocntl.h`, or `sys/fppriocntl.h` header file and described under the appropriate heading below.

If `arg` is a `NULL` pointer, no attribute data is returned but the `priocntl()` call still returns the number of configured classes.

PC_SETPARMS

Sets the class and class-specific scheduling parameters of the specified process(es). `arg` points to a structure of type `pcparms_t`. `pc_cid` specifies the class you are setting and the `pc_clparms` buffer contains the class-specific parameters you are setting. The format of the class-specific parameter data is defined in the `sys/rtpriocntl.h`, `sys/tspriocntl.h`, or `sys/fppriocntl.h` header file and described under the appropriate class heading below.

When setting parameters for a set of processes, `priocntl()` acts on the processes in the set in an implementation-specific order. If `priocntl()` encounters an error for one or more of the target processes, it may or may not continue through the set of processes, depending on the nature of the error. If the error is related to permissions (`EPERM`), `priocntl()` continues through the process set, resetting the parameters for all target processes for which the calling process has appropriate permissions. `priocntl()` then returns `-1` with `errno` set to `EPERM` to indicate that the operation failed for one or more of the target processes. If `priocntl()` encounters an error other than permissions, it does not continue through the set of target processes but returns the error immediately.

PC_GETPARMS

Get the class and/or class-specific scheduling parameters of a process. `arg` points to a structure of type `pcparms_t`.

If `pc_cid` specifies a configured class and a single process belonging to that class is specified by the `idtype` and `id` values or the `procset` structure, then the scheduling parameters of that process are returned in the `pc_clparms` buffer. If the process specified does not exist or does not belong to the specified class, the `priocntl()` call returns `-1` with `errno` set to `ESRCH`.

If `pc_cid` specifies a configured class and a set of processes is specified, the scheduling parameters of one of the specified processes belonging to the specified class are returned in the `pc_clparms` buffer and the `priocntl()` call returns the process ID of the selected process. The criteria for selecting a process to return in this case is class dependent. If none of the specified processes exist or none of them belong to the specified class the `priocntl()` call returns `-1` with `errno` set to `ESRCH`.

If `pc_cid` is `PC_CLNULL` and a single process is specified the class of the specified process is returned in `pc_cid` and its scheduling parameters are returned in the `pc_clparms` buffer.

PC_ADMIN

This command provides functionality needed for the implementation of the `dispadmin(1M)` command. It is not intended for general use by other applications.

REAL-TIME CLASS

The real-time class provides a fixed priority preemptive scheduling policy for those processes requiring fast and deterministic response and absolute user/application control of scheduling priorities. If the real-time class

is configured in the system it should have exclusive control of the highest range of scheduling priorities on the system. This ensures that a runnable real-time process is given CPU service before any process belonging to any other class.

The real-time class has a range of real-time priority (`rt_pri`) values that may be assigned to processes within the class. Real-time priorities range from 0 to x , where the value of x is configurable and can be determined for a specific installation by using the `priocntl()` `PC_GETCID` or `PC_GETCLINFO` command.

The real-time scheduling policy is a fixed priority policy. The scheduling priority of a real-time process is never changed except as the result of an explicit request by the user/application to change the `rt_pri` value of the process.

For processes in the real-time class, the `rt_pri` value is, for all practical purposes, equivalent to the scheduling priority of the process. The `rt_pri` value completely determines the scheduling priority of a real-time process relative to other processes within its class. Numerically higher `rt_pri` values represent higher priorities. Since the real-time class controls the highest range of scheduling priorities in the system it is guaranteed that the runnable real-time process with the highest `rt_pri` value is always selected to run before any other process in the system.

In addition to providing control over priority, `priocntl()` provides for control over the length of the time quantum allotted to processes in the real-time class. The time quantum value specifies the maximum amount of time a process may run assuming that it does not complete or enter a resource or event wait state (`sleep()`). Note that if another process becomes runnable at a higher priority the currently running process may be preempted before receiving its full time quantum.

The system's process scheduler keeps the runnable real-time processes on a set of scheduling queues. There is a separate queue for each configured real-time priority and all real-time processes with a given `rt_pri` value are kept together on the appropriate queue. The processes on a given queue are ordered in FIFO order (that is, the process at the front of the queue has been waiting longest for service and receives the CPU first). Real-time processes that wake up after sleeping, processes which change to the real-time class from some other class, processes which have used their full time quantum, and runnable processes whose priority is reset by `priocntl()` are all placed at the back of the appropriate queue for their priority. A process that is preempted by a higher priority process remains at the front of the queue (with whatever time is remaining in its time quantum) and runs before any other process at this priority. Following a `fork(2)` system call by a real-time process, the parent process continues to run while the child process (which inherits its parent's `rt_pri` value) is placed at the back of the queue.

The following structure (defined in `sys/rtpriocntl.h`) defines the format used for the attribute data for the real-time class.

```
typedef struct {
    short rt_maxpri;      /* Maximum real-time priority */
} rtinfo_t;
```

The `priocntl PC_GETCID` and `PC_GETCLINFO` commands return real-time class attributes in the `pc_clinfo` buffer in this format.

`rt_maxpri` specifies the configured maximum `rt_pri` value for the real-time class (if `rt_maxpri` is x , the valid real-time priorities range from 0 to x).

The following structure (defined in `sys/rtpriocntl.h`) defines the format used to specify the real-time class-specific scheduling parameters of a process.

```
typedef struct {
    short rt_pri;         /* Real-Time priority */
    ulong rt_tqsecs;      /* Seconds in time quantum */
    long  rt_tqnsecs;     /* Additional nanoseconds in quantum */
} rtparms_t;
```

When using the `priocntl PC_SETPARMS` or `PC_GETPARMS` commands, if `pc_cid` specifies the real-time class, the data in the `pc_clparms` buffer is in this format.

The above commands can be used to set the real-time priority to the specified value or get the current `rt_pri` value. Setting the `rt_pri` value of a process that is currently running or runnable (not sleeping)

causes the process to be placed at the back of the scheduling queue for the specified priority. The process is placed at the back of the appropriate queue regardless of whether the priority being set is different from the previous `rt_pri` value of the process. Note that a running process can voluntarily release the CPU and go to the back of the scheduling queue at the same priority by resetting its `rt_pri` value to its current real-time priority value. In order to change the time quantum of a process without setting the priority or affecting the process' position on the queue, the `rt_pri` field should be set to the special value `RT_NOCHANGE` (defined in `sys/rtpriocntl.h`). Specifying `RT_NOCHANGE` when changing the class of a process to real-time from some other class results in the real-time priority being set to zero.

For the `priocntl PC_GETPARMS` command, if `pc_cid` specifies the real-time class and more than one real-time process is specified, the scheduling parameters of the real-time process with the highest `rt_pri` value among the specified processes are returned and the process ID of this process is returned by the `priocntl()` call. If there is more than one process sharing the highest priority, the one returned is implementation-dependent.

The `rt_tqsecs` and `rt_tqnsecs` fields are used for getting or setting the time quantum associated with a process or group of processes. `rt_tqsecs` is the number of seconds in the time quantum and `rt_tqnsecs` is the number of additional nanoseconds in the quantum. For example setting `rt_tqsecs` to 2 and `rt_tqnsecs` to 500,000,000 (decimal) would result in a time quantum of two and one-half seconds. Specifying a value of 1,000,000,000 or greater in the `rt_tqnsecs` field results in an error return with `errno` set to `EINVAL`. Although the resolution of the `tq_nsecs` field is very fine, the specified time quantum length is rounded up by the system to the next integral multiple of the system clock's resolution. For example, the finest resolution currently available on the 3B2 is 10 milliseconds (1 "tick"). Setting `rt_tqsecs` to 0 and `rt_tqnsecs` to 34,000,000 would specify a time quantum of 34 milliseconds, which would be rounded up to 4 ticks (40 milliseconds) on the 3B2. The maximum time quantum that can be specified is implementation-specific and equal to `LONG_MAX` ticks (defined in `limits.h`). Requesting a quantum greater than this maximum results in an error return with `errno` set to `ERANGE` (although infinite quanta may be requested using a special value as explained below). Requesting a time quantum of zero (setting both `rt_tqsecs` and `rt_tqnsecs` to 0) results in an error return with `errno` set to `EINVAL`.

The `rt_tqnsecs` field can also be set to one of the following special values (defined in `sys/rtpriocntl.h`), in which case the value of `rt_tqsecs` is ignored.

`RT_TQINF`

Sets an infinite time quantum.

`RT_TQDEF`

Sets the time quantum to the default for this priority [see `rt_dptbl(4)`].

`RT_NOCHANGE`

Don't set the time quantum. This value is useful when you wish to change the real-time priority of a process without affecting the time quantum. Specifying this value when changing the class of a process to real-time from some other class is equivalent to specifying `RT_TQDEF`.

In order to change the class of a process to real-time (from any other class) the process invoking `priocntl()` must have superuser privileges. In order to change the priority or time quantum setting of a real-time process the process invoking `priocntl()` must have superuser privileges or must itself be a real-time process whose real or effective user ID matches the real or effective user ID of the target process.

The real-time priority and time quantum are inherited across the `fork(2)` and `exec(2)` system calls.

TIME-SHARING CLASS

The time-sharing scheduling policy provides for a fair and effective allocation of the CPU resource among processes with varying CPU consumption characteristics. The objectives of the time-sharing policy are to provide good response time to interactive processes and good throughput to CPU-bound jobs while providing a degree of user/application control over scheduling.

The time-sharing class has a range of time-sharing user priority (see `ts_upri` below) values that may be assigned to processes within the class. A `ts_upri` value of zero is defined as the default base priority for the time-sharing class. User priorities range from $-x$ to $+x$ where the value of x is configurable and can be determined for a specific installation by using the `priocntl PC_GETCID` or `PC_GETCLINFO` command.

The purpose of the user priority is to provide some degree of user/application control over the scheduling of processes in the timesharing class. Raising or lowering the `ts_upri` value of a process in the time-sharing class raises or lowers the scheduling priority of the process. It is not guaranteed, however, that a process with a higher `ts_upri` value will run before one with a lower `ts_upri` value. This is because the `ts_upri` value is just one factor used to determine the scheduling priority of a time-sharing process. The system may dynamically adjust the internal scheduling priority of a time-sharing process based on other factors such as recent CPU usage.

In addition to the system-wide limits on user priority (returned by the `PC_GETCID` and `PC_GETCLINFO` commands) there is a per process user priority limit (see `ts_uprilim` below), which specifies the maximum `ts_upri` value that may be set for a given process; by default, `ts_uprilim` is zero.

The following structure (defined in `sys/tspriocntl.h`) defines the format used for the attribute data for the time-sharing class.

```
typedef struct {
    short ts_maxupri;    /* Limits of user priority range */
} tsinfo_t;
```

The `priocntl PC_GETCID` and `PC_GETCLINFO` commands return timesharing class attributes in the `pc_clinfo` buffer in this format.

`ts_maxupri` specifies the configured maximum user priority value for the time-sharing class. If `ts_maxupri` is `x`, the valid range for both user priorities and user priority limits is from `-x` to `+x`.

The following structure (defined in `sys/tspriocntl.h`) defines the format used to specify the time-sharing class-specific scheduling parameters of a process.

```
typedef struct {
    short ts_uprilim;    /* Time-Sharing user priority limit */
    short ts_upri;       /* Time-Sharing user priority */
    short ts_flags;      /* (see below) */
} tsparms_t;
/* tsparms_t.ts_flags values */
#define TS_FIXEDPRI 1
```

When using the `priocntl PC_SETPARMS` or `PC_GETPARMS` commands, if `pc_cid` specifies the time-sharing class, the data in the `pc_clparms` buffer is in this format.

For the `priocntl PC_GETPARMS` command, if `pc_cid` specifies the time-sharing class and more than one time-sharing process is specified, the scheduling parameters of the time-sharing process with the highest `ts_upri` value among the specified processes is returned and the process ID of this process is returned by the `priocntl()` call. If there is more than one process sharing the highest user priority, the one returned is implementation-dependent.

Any time-sharing process may lower its own `ts_uprilim` (or that of another process with the same user ID). Only a time-sharing process with superuser privileges may raise a `ts_uprilim`. When changing the class of a process to time-sharing from some other class, superuser privileges are required in order to set the initial `ts_uprilim` to a value greater than zero. Attempts by a non-superuser process to raise a `ts_uprilim` or set an initial `ts_uprilim` greater than zero fail with a result of `-1` and `errno` set to `EPERM`.

Any time-sharing process may set its own `ts_upri` (or that of another process with the same user ID) to any value less than or equal to the process' `ts_uprilim`. Attempts to set the `ts_upri` above the `ts_uprilim` (and/or set the `ts_uprilim` below the `ts_upri`) result in the `ts_upri` being set equal to the `ts_uprilim`.

The `tsparms_t` structure also has a `ts_flags` value that can be set by a process with super-user privileges. If the flag has the `TS_FIXEDPRI` flag bit set, then the time-share scheduling class will not degrade the priority of the process when it consumes its entire time slice. This feature is useful for processes such as system daemons and database servers. This feature has been replaced by the fixed-priority class. It is left only for compatibility purposes and will be removed in a future release.

Any of the `ts_uprilim`, `ts_upri`, or `ts_flags` fields may be special value `TS_NOCHANGE` (defined in `sys/tspriocntl.h`) in order to set one of the values without affecting the other. Specifying `TS_NOCHANGE` for the `ts_upri` when the `ts_uprilim` is being set to a value below the current `ts_upri` causes the

`ts_upri` to be set equal to the `ts_uprilim` being set. Specifying `TS_NOCHANGE` for a parameter when changing the class of a process to time-sharing (from some other class) causes the parameter to be set to a default value. The default value for the `ts_uprilim` is 0 and the default for the `ts_upri` is to set it equal to the `ts_uprilim` which is being set.

The time-sharing user priority and user priority limit are inherited across the `fork(2)` and `exec(2)` system calls.

FIXED-PRIORITY CLASS

The fixed-priority class provides a fixed priority preemptive scheduling policy for those processes requiring fast and deterministic response and absolute user/application control of scheduling priorities much like the real-time class. However, the fixed-priority class has scheduling priorities that extend only slightly higher than the time-share class. This allows a process to get a great deal of the CPU without preventing the kernel from running.

The fixed-priority class has a range of fixed-priority priority (`fp_umdpr`) values that may be assigned to processes within the class. Fixed-priority priorities range from 0 to x , where the value of x is configurable and can be determined for a specific installation by using the `prioctl PC_GETCID` or `PC_GETCLINFO` command.

The fixed-priority scheduling policy is a fixed priority policy. The scheduling priority of a fixed-priority process is only changed when the process goes to sleep in the kernel, or as the result of an explicit request by the user/application to change the `fp_umdpr` value of the process.

The `fp_umdpr` value is (for processes in the fixed-priority class) the scheduling priority of the process. The `fp_umdpr` value completely determines the scheduling priority of a fixed-priority process relative to other processes. Numerically higher `fp_umdpr` values represent higher priorities.

In addition to providing control over priority, `prioctl` provides for control over the length of the time quantum allotted to processes in the fixed-priority class. The time quantum value specifies the maximum amount of time a process may run assuming that it does not complete or enter a resource or event wait state (sleep). If another process becomes runnable at a higher priority the currently running process may be preempted before receiving its full time quantum.

The system's process scheduler keeps the runnable fixed-priority processes on a set of scheduling queues. There is a separate queue for each configured fixed-priority priority and all fixed-priority processes with a given `fp_umdpr` value are kept together on the appropriate queue. The processes on a given queue are ordered in FIFO order (that is, the process at the front of the queue has been waiting longest for service and receives the CPU first).

Fixed-priority processes that wake up after sleeping, processes which change to the fixed-priority class from some other class, processes which have used their full time quantum, and runnable processes whose priority is reset by `prioctl` are all placed at the back of the appropriate queue for their priority.

A process that is preempted by a higher priority process gets placed in the queue according to whatever time is remaining in its time quantum. Following a `fork(2)` system call by a fixed-priority process, the parent process relinquishes the CPU to the child process and is placed on the back of its scheduler priority queue.

The following structure (defined in `<sys/fpprioctl.h>`) defines the format used for the attribute data for the fixed-priority class.

```
typedef struct {
    short  fp_maxumdpr;    /* configured limits of user priority range */
} fpinfo_t;
```

The `prioctl PC_GETCID` and `PC_GETCLINFO` commands return fixed-priority class attributes in the `pc_clinfo` buffer in this format.

`fp_maxumdpr` specifies the configured maximum `fp_umdpr` value for the fixed-priority class (if `rt_maxumdpr` is x , the valid fixed-priority priorities range from 0 to x).

The following structure (defined in `<sys/fpprioctl.h>`) defines the format used to specify the fixed-priority class-specific scheduling parameters of a process.

```
typedef struct {
    short fp_pri;           /* Real-Time priority */
    ulong fp_tqsecs;        /* Seconds in time quantum */
    long fp_tqnsecs;        /* Additional nanoseconds in quantum */
} fpparms_t;
```

When using the `prioctl` `PC_SETPARMS` or `PC_GETPARMS` commands, if `pc_cid` specifies the fixed-priority class, the data in the `pc_clparms` buffer is in this format.

These commands can be used to set the fixed-priority priority to the specified value or get the current `fp_pri` value. Setting the `fp_pri` value of a process that is currently running or runnable (not sleeping) causes the process to be placed at the back of the scheduling queue for the specified priority. The process is placed at the back of the appropriate queue regardless of whether the priority being set is different from the previous `fp_pri` value of the process.

A running process can voluntarily release the CPU and go to the back of the scheduling queue at the same priority by resetting its `fp_pri` value to its current fixed-priority priority value.

For the `prioctl` `PC_GETPARMS` command, if `pc_cid` specifies the fixed-priority class and more than one fixed-priority process is specified, the scheduling parameters of the fixed-priority process with the highest `fp_umdpr` value among the specified processes are returned and the process ID of this process is returned by the `prioctl` call. If there is more than one process sharing the highest priority, the one returned is implementation-dependent.

The `fp_tqsecs` and `fp_tqnsecs` fields are used for getting or setting the time quantum associated with a process or group of processes. `fp_tqsecs` is the number of seconds in the time quantum and `fp_tqnsecs` is the number of additional nanoseconds in the quantum. For example setting `fp_tqsecs` to 2 and `fp_tqnsecs` to 500,000,000 (decimal) would result in a time quantum of two and one-half seconds. Specifying a value of 1,000,000,000 or greater in the `fp_tqnsecs` field results in an error return with `errno` set to `EINVAL`. Although the resolution of the `tq_nsecs` field is very fine, the specified time quantum length is rounded up by the system to the next integral multiple of the system clock's resolution. For example, the finest resolution currently available on the 3B2 is 10 milliseconds (1 "tick"). Setting `fp_tqsecs` to 0 and `fp_tqnsecs` to 34,000,000 would specify a time quantum of 34 milliseconds, which would be rounded up to 4 ticks (40 milliseconds) on the 3B2. The maximum time quantum that can be specified is implementation-specific and equal to `LONG_MAX` ticks (defined in `limits.h`). Requesting a quantum greater than this maximum results in an error return with `errno` set to `ERANGE` (although infinite quanta may be requested using a special value as explained below). Requesting a time quantum of zero (setting both `fp_tqsecs` and `fp_tqnsecs` to 0) results in an error return with `errno` set to `EINVAL`.

The `fp_tqnsecs` field can also be set to one of the following special values (defined in `sys/rtprioctl.h`), in which case the value of `fp_tqsecs` is ignored.

`FP_TQINF`

Sets an infinite time quantum.

`FP_TQDEF`

Sets the time quantum to the default for this priority [see `fp_dptbl(4)`].

`FP_NOCHANGE`

Don't set the time quantum. This value is useful when you wish to change the real-time priority of a process without affecting the time quantum. Specifying this value when changing the class of a process to real-time from some other class is equivalent to specifying `FP_TQDEF`.

In order to change the class of a process to fixed-priority (from any other class) the process invoking `prioctl()` must have super-user privileges. In order to change the priority setting of a fixed-priority process the process invoking `prioctl()` must have super-user privileges or must itself be a fixed-priority process whose real or effective user ID matches the real or effective user ID of the target process.

The fixed-priority priority is inherited across the `fork(2)` and `exec(2)` system calls.

PROCESSOR AFFINITY AND BINDING CONSIDERATIONS

In order to increase the cache hit rates of a multiprocessor, a process should generally execute on a particular processor. This affinity for a particular processor will reduce the required memory bandwidth for the process

(and for the system as a whole). If a process has not run recently then it probably only has a small cache state in any given processor and therefore, should loose its affinity for any particular processor.

This affinity mechanism can be enabled or disabled on a per-process basis by the super user. See `mpcntl(3X)`. The affinity mechanism defaults to `on` for the `TS` and `FP` classes and to `off` for the `RT` class.

A process can also be bound to a set of processors. If a processor has been so bound, then it will *only* run on one of the processors of that set. If there is a idle processor that is not in the processes binding set – then the process will still wait until there is a processor in its set that is not running a higher priority process.

Process bindings can be established by the process or can be temporarily established by the kernel (if the kernel is executing driver code that must run on a particular CPU). See `mpcntl(3X)`.

RETURN VALUE

Unless otherwise noted, `priocntl` returns a value of 0 on success. `priocntl` returns `-1` on failure and sets `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`priocntl()` fails if one or more of the following are true:

`EPERM`

The calling process does not have the required permissions as explained above.

`EINVAL`

The argument `cmd` was invalid, an invalid or unconfigured class was specified, or one of the parameters specified was invalid.

`EDOM` The requested time quantum is out of range.

`ESRCH`

None of the specified processes exist.

`EFAULT`

All or part of the area pointed to by one of the data pointers is outside the process' address space.

`ENOMEM`

An attempt to change the class of a process failed because of insufficient memory.

`EAGAIN`

An attempt to change the class of a process failed because of insufficient resources other than memory (for example, class-specific kernel data structures).

SEE ALSO

`priocntl(1)`, `dispadm(1M)`, `exec(2)`, `fork(2)`, `nice(2)`, `priocntlset(2)`, `mpcntl(3X)`, `fp_dptbl(4)`, `rt_dptbl(4)`, `ts_dptbl(4)`, `types(5)`.

prioctlset(2)**NAME**

prioctlset, setprocset - control process scheduling

SYNOPSIS

```
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/prioctl.h>
#include <sys/rtprioctl.h>
#include <sys/tsprioctl.h>
long prioctlset(procset_t *psp, int cmd, ... /* arg */);
```

DESCRIPTION

prioctlset() changes the scheduling properties of running processes. prioctlset() has the same functions as the prioctl() system call, but a more general way of specifying the set of processes whose scheduling properties are to be changed.

cmd specifies the function to be performed. arg is a pointer to a structure whose type depends on cmd. See [prioctl\(2\)](#) for the valid values of cmd and the corresponding arg structures.

psp is a pointer to a procset structure, which prioctlset() uses to specify the set of processes whose scheduling properties are to be changed.

```
typedef struct procset {
    idop_t    p_op;        /* operator connecting left/right sets */
    idtype_t  p_lidtype;   /* left set ID type */
    id_t      p_lid;       /* left set ID */
    idtype_t  p_ridtype;   /* right set ID type */
    id_t      p_rid;       /* right set ID */
} procset_t;
```

p_lidtype and p_lid specify the ID type and ID of one ("left") set of processes; p_ridtype and p_rid specify the ID type and ID of a second ("right") set of processes. ID types and IDs are specified just as for the prioctl() system call. p_op specifies the operation to be performed on the two sets of processes to get the set of processes the system call is to apply to. The valid values for p_op and the processes they specify are:

POP_DIFF

Set difference: processes in left set and not in right set

POP_AND

Set intersection: processes in both left and right sets

POP_OR

Set union: processes in either left or right sets or both

POP_XOR

Set exclusive-or: processes in left or right set but not in both

The following macro, which is defined in procset.h, offers a convenient way to initialize a procset structure:

```
#define setprocset(psp, op, ltype, lid, rtype, rid) \
(psp)->p_op      = (op), \
(psp)->p_lidtype = (ltype), \
(psp)->p_lid     = (lid), \
(psp)->p_ridtype = (rtype), \
(psp)->p_rid     = (rid),
```

RETURN VALUE

`priocntlset()` has the same results and errors as `priocntl()`.

SEE ALSO

`priocntl(1)`, `priocntl(2)`, `types(5)`.

NAME

`profil` - execution time profile

SYNOPSIS

```
#include <unistd.h>
void profil(unsigned short *buff, size_t bufsiz, int offset,
            unsigned scale);
```

DESCRIPTION

`profil()` provides CPU-use statistics by profiling the amount of CPU time expended by a program. `profil()` generates the statistics by creating an execution histogram for a current process. The histogram is defined for a specific region of program code to be profiled, and the identified region is logically broken up into a set of equal size subdivisions, each of which corresponds to a count in the histogram. With each clock tick, the current subdivision is identified and its corresponding histogram count is incremented. These counts establish a relative value for the amount of time being spent in each code subdivision. The resulting histogram counts for a profiled region and can be used to identify those functions that consume a disproportionately high percentage of CPU time.

`buff` is a buffer of `bufsiz` bytes in which the histogram counts are stored in an array of unsigned short int. `offset`, `scale`, and `bufsiz` specify the region to be profiled.

`offset` is the start address of the region to be profiled.

`scale` is a contraction factor that indicates how much smaller the histogram buffer is than the region to be profiled. That is, `scale` is interpreted as an unsigned 16-bit fixed-point fraction with the decimal point implied on the left. Its value is the reciprocal of the number of bytes in a subdivision, per byte of histogram buffer. Since there are two bytes per histogram counter, the effective ratio of subdivision bytes per counter is double the `scale`.

- The maximum value of `scale`, 0xffff (approximately 1), maps subdivisions 2 bytes long to each counter.
- The minimum value of `scale` (for which profiling is performed), 0x0002 (1/32768), maps a subdivision 65536 bytes long to each counter.
- The default value of `scale` (currently used by `cc -gp`), 0x4000, maps a subdivision 8 bytes long to each counter.

The values are used within the kernel as follows: when the process is interrupted for a clock tick, the value of `offset` is subtracted from the current value of the program counter, and the remainder is multiplied by `scale` to derive a result. That result is used as an index into the histogram array to locate the cell to be incremented. Therefore, the cell count represents the number of times that the process was executing code in the subdivision associated with that cell when the process was interrupted.

`scale` can be computed as $(RATIO * 0200000L)$, where `RATIO` is the desired ratio of `bufsiz` to profiled region size, and has a value between 0 and 1. The closer `RATIO` is to 1, the higher the resolution of the profile information.

`bufsiz` can be computed as $(size\_of\_region\_to\_be\_profiled * RATIO)$.

NOTES

Profiling is turned off by giving a `scale` of 0 or 1, and is rendered ineffective by giving a `bufsiz` of 0. Profiling is turned off when an `exec(2)` is executed, but remains on in both child and parent processes after a `fork(2)`. Profiling is turned off if a `buff` update would cause a memory fault.

SEE ALSO

`times(2)`, `monitor(3C)`, `unistd(4)`.

NAME

ptrace - process trace

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>
long ptrace(int request, pid_t pid, long addr,
long data);
```

DESCRIPTION

`ptrace()` 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\(5\)](#)], at which time it enters a stopped state and its parent is notified via the `wait(2)` system call. 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()`. The possible values of *request* are the following:

- 0 The child process requests that it is traced by its parent. This 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 *disp* [see [signal\(2\)](#)]. The *pid*, *addr*, and *data* arguments are ignored, and a result 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 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 a value of -1 is returned to the parent process and the parent's `errno` is set to `EIO`.
- 3 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 a value of -1 is returned to the parent process and the parent's `errno` is set to `EIO`.
- 4, 5 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 a value of -1 is returned to the parent process and the parent's `errno` is set to `EIO`.
- 6 With this request, a few entries in the child's user area can be written. *data* gives the value that is to be written and *addr* is the location of the entry. The entries can be the general registers and the condition codes of the Processor Status Word.
- 7 The child resumes 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 a value

of `-1` is returned to the parent process and the parent's `errno` is set to `EIO`.

- 8 The child is terminated, with the same consequences as `exit(2)`.
- 9 The trace bit in the Processor Status Word of the child is set and then the same steps are executed 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 `exec(2)` calls. If a traced process calls `exec(2)`, it stops before executing the first instruction of the new image showing signal `SIGTRAP`. `ptrace()` in general fails if one or more of the following are true:

`EIO` *request* is an illegal number.

`ESRCH`

pid identifies a child that does not exist or has not executed a `ptrace()` with request `0`.

`EPERM`

The invoking subject does not have the appropriate MAC privileges.

SEE ALSO

`exec(2)`, `signal(2)`, `wait(2)`, `unistd(4)`, `types(5)`.

putmsg(2)

NAME

putmsg, putpmsg - send a message on a stream

SYNOPSIS

```
#include <stropts.h>

int putmsg(int fildev, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flags);

int putpmsg(int fildev, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band, int flags);
```

DESCRIPTION

`putmsg()` creates a message from user-specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part, or both. The data and control parts to be sent are distinguished by placement in separate buffers. The semantics of each part is defined by the STREAMS module that receives the message.

The function `putpmsg()` does the same thing as `putmsg()`, but provides the user the ability to send messages in different priority bands. Except where noted, all information pertaining to `putmsg()` also pertains to `putpmsg()`.

fildev specifies a file descriptor referencing an open stream. *ctlptr* and *dataptr* each point to a `strbuf` structure, which contains the following members:

```
int maxlen;      /* not used */
int len;         /* length of data */
void *buf;       /* ptr to buffer */
```

ctlptr points to the structure describing the control part to be included in the message. The `buf` field in the `strbuf` structure points to the buffer where the control information resides, and the `len` field indicates the number of bytes to be sent. The `maxlen` field is not used by `putmsg` [see `getmsg(2)`]. In a similar manner, *dataptr* specifies the data to be included in the message. *flags* indicates what type of message should be sent.

To send the data part of a message, *dataptr* must not be `NULL` and the `len` field of *dataptr* must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part is sent if either *dataptr* (*ctlptr*) is `NULL` or the `len` field of *dataptr* (*ctlptr*) is set to -1.

For `putmsg()`, if a control part is specified, and *flags* is set to `RS_HIPRI`, a high priority message is sent. If no control part is specified, and *flags* is set to `RS_HIPRI`, `putmsg()` fails and sets `errno` to `EINVAL`. If *flags* is set to 0, a normal (non-priority) message is sent. If no control part and no data part are specified, and *flags* is set to 0, no message is sent, and 0 is returned.

For `putpmsg()`, the flags are different. *flags* is a bitmask with the mutually-exclusive flags `MSG_HIPRI` and `MSG_BAND`. If *flags* is set to 0, `putpmsg()` fails and sets `errno` to `EINVAL`. If a control part is specified and *flags* is set to `MSG_HIPRI` and *band* is set to 0, a high-priority message is sent. If *flags* is set to `MSG_HIPRI` and either no control part is specified or *band* is set to a non-zero value, `putpmsg()` fails and sets `errno` to `EINVAL`. If *flags* is set to `MSG_BAND`, then a message is sent in the priority band specified by *band*. If a control part and data part are not specified and *flags* is set to `MSG_BAND`, no message is sent and 0 is returned.

Normally, `putmsg()` will block if the stream write queue is full due to internal flow control conditions. For high-priority messages, `putmsg()` does not block on this condition. For other messages, `putmsg()` does not block when the write queue is full and `O_NDELAY` or `O_NONBLOCK` is set. Instead, it fails and sets `errno` to `EAGAIN`.

`putmsg()` or `putpmsg()` also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the stream, regardless of priority or whether `O_NDELAY` or `O_NONBLOCK` has been specified. No partial message is sent.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`putmsg()` and `putpmsg()` fail if one or more of the following apply:

EAGAIN

A non-priority message was specified, the `O_NDELAY` or `O_NONBLOCK` flag is set and the stream write queue is full due to internal flow control conditions.

EBADF

fildev is not a valid file descriptor open for writing.

EFAULT

ctlptr or *dataptr* points outside the allocated address space.

EINTR

A signal was caught during the `putmsg()` system call.

EINVAL

An undefined value is specified in *flags*, or *flags* is set to `RS_HIPRI` or `MSG_HIPRI` and no control part is supplied, or the stream or multiplexer referenced by *fildev* is linked (directly or indirectly) downstream from a multiplexer, or *flags* is set to `MSG_HIPRI` and *band* is non-zero (for `putpmsg()` only).

ENOSR

Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.

ENOSTR

A stream is not associated with *fildev*.

ENXIO

A hangup condition was generated downstream for the specified stream, or the other end of the pipe is closed.

EPIPE or EIO

The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

ERANGE

The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

In addition, `putmsg()` and `putpmsg()` will fail if the stream head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `putmsg()` or `putpmsg()` but reflects the prior error.

NOTES

If two processes open a FIFO file, in which one of them uses `putmsg()` to write a high priority message, and the other reads a high priority message using `getmsg()`, it is possible for messages to be lost. Such a loss can be avoided if the sending process is slowed down between successive invocations of `putmsg()`, by means of `sleep()`.

SEE ALSO

[getmsg\(2\)](#), [poll\(2\)](#), [putmsg\(2\)](#), [read\(2\)](#), [write\(2\)](#), [stropts\(5\)](#).

"Programmer's Guide: STREAMS".

NAME

`pwrite`, `pwrite64` - atomic position and write

SYNOPSIS

```
#include <unistd.h>

ssize_t pwrite(int fd, const void *buf, size_t nbyte,
off_t offset) ;

ssize_t pwrite64(int fd, const void *buf, size_t nbyte,
off64_t offset) ;
```

DESCRIPTION

The `pwrite()` system call does an atomic position-and-write, eliminating the necessity of using a locking mechanism when both operations are desired and file descriptors are shared. `pwrite()` is analogous to `write()` but takes a fourth argument, *offset*. The write is done as if an `lseek()` to *offset* (from the beginning of the file) were done first. Note that (though the semantics are analogous) an `lseek()` is not actually performed; the file pointer is not affected by `pwrite()`. The write of *nbytes* then starts at the specified offset.

The atomicity of `pwrite()` enables processes or threads that share file descriptors to write to the shared file at a particular offset without using a locking mechanism that would be necessary to achieve the same result in separate `lseek()` or `lseek64()` and `write()` system calls. Atomicity is required as the file pointer is shared and one thread might move the pointer using `lseek()` or `lseek64()` after another process completes an `lseek()` or `lseek64()` but prior to the `write()`.

There is no functional difference between `pwrite()` and `pwrite64()`, except for the interpretation of `off64_t` [see [lfs\(5\)](#)].

RETURN VALUE

Upon successful completion, `pwrite()` returns the number of bytes actually written from *buf*. Otherwise a `-1` and an error is returned.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `pwrite()` function will fail if:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor and the thread would be delayed in the `write()` operation.

EAGAIN

Total amount of system memory available when reading via raw I/O is temporarily insufficient.

EAGAIN

An attempt is made to write to a stream that can not accept data with the `O_NDELAY` or `O_NONBLOCK` flag set.

EAGAIN

If a `pwrite()` to a pipe or FIFO of `PIPE_BUF` bytes or less is requested and less than *nbytes* of free space is available.

EBADF

The *fd* argument is not a valid file descriptor open for writing.

EDEADLK

The `pwrite()` is sleeping and causes a deadlock situation to occur.

EFAULT

buf points outside the process' allocated address space.

EFBIG

An attempt was made to write a file that exceeds the implementation-dependent maximum file size or the process' file size limit.

EFBIG

The file is a regular file, *nbyte* is greater than 0 and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fil*.

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.

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. This error may also be returned under implementation-dependent conditions.

ENOLCK

The system record lock table was full, so the `pwrite()` could not go to sleep until the blocking record lock was removed.

ENOLCK

Enforced record locking was enabled, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.

ENOLINK

fil is on a remote machine and the link to that machine is no longer active.

ENOSPC

There was no free space remaining on the device containing the file.

ENOSR

An attempt is made to write to a stream with insufficient STREAMS memory resources available in the system.

EPIPE

An attempt is made to write to a pipe or FIFO that is not open for reading by any process, or that only has one end open. A *SIGPIPE* signal will also be sent to the thread.

ERANGE

The transfer request size was outside the range supported by the STREAMS file associated with *fil*.

The `pwrite()` function fails and the file pointer remains unchanged if:

EINVAL

The *offset* argument is invalid. The value is negative.

ESPIPE

fil is associated with a pipe or FIFO.

The `pwrite()` function may fail if:

EINVAL

The stream or multiplexer referenced by *fil* is linked (directly or indirectly) downstream from a multiplexer.

ENXIO

A request was made of a non-existent device, or the request was outside the capabilities of the device.

ENXIO

A hangup occurred on the stream being written to.

A write to a STREAMS file may fail if an error message has been received at the stream head. In this case, *errno* is set to the value included in the error message.

SEE ALSO

`creat(2)`, `dup(2)`, `fcntl(2)`, `lseek(2)`, `open(2)`, `pread(2)`, `write(2)`, `unistd(4)`, `lfs(5)`.

NAME

`read`, `readv` - read from file

SYNOPSIS

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

ssize_t read(int fd, void *buf, size_t nbyte);
ssize_t readv(int fd, const struct iovec *iov, int
iovcnt);
```

DESCRIPTION

The `read()` function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fd*, into the buffer pointed to by *buf*.

If *nbyte* is 0, `read()` will return 0 and have no other results.

On files that support seeking (for example, a regular file), the `read()` starts at a position in the file given by the file offset associated with *fd*. The file offset is incremented by the number of bytes actually read.

Files that do not support seeking, for example, terminals, always read from the current position. The value of a file offset associated with such a file is undefined.

No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 will be returned. If the file refers to a device special file, the result of subsequent `read()` requests is implementation-dependent.

If the value of *nbyte* is greater than `SSIZE_MAX`, the result is implementation-dependent.

When attempting to read from an empty pipe or FIFO:

- If no process has the pipe open for writing, `read()` will return 0 to indicate end-of-file.
- If some process has the pipe open for writing and `O_NONBLOCK` is set, `read()` will return -1 and set `errno` to `EAGAIN`.
- If some process has the pipe open for writing and `O_NONBLOCK` is clear, `read()` will block the calling thread until some data is written or the pipe is closed by all processes that had the pipe open for writing.

When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and has no data currently available:

- If `O_NONBLOCK` is set, `read()` will return a -1 and set `errno` to `EAGAIN`.
- If `O_NONBLOCK` is clear, `read()` will block the calling thread until some data becomes available.
- The use of the `O_NONBLOCK` flag has no effect if there is some data available.

The `read()` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read()` returns bytes with value 0. For example, `lseek()` allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap.

Upon successful completion, where *nbyte* is greater than 0, `read()` will mark for update the `st_atime` field of the file, and return the number of bytes read. This number will never be greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte*, if the `read()` request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a `read()` from a file associated with a terminal may return one typed line of data.

If a `read()` is interrupted by a signal before it reads any data, it will return `-1` with `errno` set to `EINTR`.

If a `read()` is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

A `read()` from a STREAMS file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT ioctl()` request, and can be tested with the `I_GRDOPT ioctl()`. In byte-stream mode, `read()` retrieves data from the stream until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, `read()` retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If `read()` does not retrieve all the data in a message, the remaining data is left on the stream, and can be retrieved by the next `read()` call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the `read()` returns is discarded, and is not available for a subsequent `read()`, `readv()` or `getmsg()` call.

How `read()` handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, `read()` accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The `read()` function then returns the number of bytes read, and places the zero-byte message back on the stream to be retrieved by the next `read()`, `readv()` or `getmsg()`. In message-nondiscard mode or message-discard mode, a zero-byte message returns 0 and the message is removed from the stream. When a zero-byte message is read as the first message on a stream, the message is removed from the stream and 0 is returned, regardless of the read mode.

A `read()` from a STREAMS file returns the data in the message at the front of the stream head read queue, regardless of the priority band of the message.

By default, streams are in control-normal mode, in which a `read()` from a STREAMS file can only process messages that contain a data part but do not contain a control part. The `read()` fails if a message containing a control part is encountered at the stream head. This default action can be changed by placing the stream in either control-data mode or control-discard mode with the `I_SRDOPT ioctl()` command. In control-data mode, `read()` converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, `read()` discards message control parts but returns to the process any data part in the message.

In addition, `read()` and `readv()` will fail if the stream head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `read()` or `readv()` but reflects the prior error. If a hangup occurs on the stream being read, `read()` continues to operate normally until the stream head read queue is empty. Thereafter, it returns 0.

The `readv()` function is equivalent to `read()`, but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*–1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to `IOV_MAX`.

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The `readv()` function always fills an area completely before proceeding to the next.

Upon successful completion, `readv()` marks for update the `st_atime` field of the file.

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

RETURN VALUE

Upon successful completion, `read()` and `readv()` return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return `-1` and set `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `read()`, and `readv()` functions will fail if:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor and the process would be delayed.

EBADF

The *fildev* argument is not a valid file descriptor open for reading.

EBADMSG

The file is a stream file that is set to control-normal mode and the message waiting to be read includes a control part.

EINTR

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

EINVAL

The stream or multiplexer referenced by *fildev* is linked (directly or indirectly) downstream from a multiplexer.

EIO A physical I/O error has occurred.

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. This error may also be generated for implementation-dependent reasons.

EISDIR

The *fildev* argument refers to a directory and the implementation does not allow the directory to be read using `read()` or `readv()`. The `readdir()` function should be used instead.

EOVERFLOW

The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildev*.

The `readv()` function will fail if:

EINVAL

The sum of the *iov_len* values in the *iov* array overflowed an `ssize_t`.

The `read()` and `readv()` functions may fail if:

ENXIO

A request was made of a non-existent device, or the request was outside the capabilities of the device.

The `readv()` function may fail if:

EINVAL

The *iovcnt* argument was less than or equal to 0, or greater than `IOV_MAX`.

SEE ALSO

`creat(2)`, `dup(2)`, `fcntl(2)`, `getmsg(2)`, `ioctl(2)`, `open(2)`, `pipe(2)`, `pread(2)`, `unistd(4)`, `lfs(5)`, `types(5)`, `uio(5)`, `streamio(7)`, `termio(7)`.

NAME

`readlink` - read the value of a symbolic link

SYNOPSIS

```
#include <unistd.h>

int readlink(const char *path, char *buf, size_t bufsiz);
```

DESCRIPTION

`readlink()` places the contents of the symbolic link referred to by *path* in the buffer *buf*, which has size *bufsiz*. The contents of the link are not null-terminated when returned.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`readlink()` fails and the buffer remains unchanged if:

EACCES

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

EFAULT

path or *buf* extends outside the allocated address space of the process.

EINVAL

The named file is not a symbolic link.

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

ELOOP

Too many symbolic links are encountered in translating *path*.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a path component exceeds `NAME_MAX`.

ENOENT

The named file does not exist.

ENOSYS

The file system does not support symbolic links.

ENOTDIR

A component of the path prefix is not a directory.

The `readlink()` function may fail if:

EACCES

Read permission is denied for the directory.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

RESULT

Upon successful completion `readlink()` returns the number of characters placed in the buffer; otherwise, it returns `-1` and places an error code in `errno`.

SEE ALSO

[lstat\(2\)](#), [stat\(2\)](#), [symlink\(2\)](#), [unistd\(4\)](#).

NAME

`rename` - change the name of a file

SYNOPSIS

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

DESCRIPTION

The `rename()` function changes the name of a file. The *old* argument points to the pathname of the file to be renamed. The *new* argument points to the new pathname of the file.

If the *old* argument and the *new* argument both refer to, and both link to the same existing file, `rename()` returns successfully and performs no other action.

If the *old* argument points to the pathname of a file that is not a directory, the *new* argument must not point to the pathname of a directory. If the link named by the *new* argument exists, it is removed and *old* renamed to *new*. In this case, a link named *new* will remain visible to other processes throughout the renaming operation and will refer either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory containing *old* and the directory containing *new*.

If the *old* argument points to the pathname of a directory, the *new* argument must not point to the pathname of a file that is not a directory. If the directory named by the *new* argument exists, it will be removed and *old* renamed to *new*. In this case, a link named *new* will exist throughout the renaming operation and will refer either to the file referred to by *new* or *old* before the operation began. Thus, if *new* names an existing directory, it must be an empty directory.

If *old* points to a pathname that names a symbolic link, the symbolic link is renamed. If *new* points to a pathname that names a symbolic link, the symbolic link is removed.

The *new* pathname must not contain a path prefix that names *old*. Write access permission is required for the directory containing *old* and the directory containing *new*. If the *old* argument points to the pathname of a directory, write access permission may be required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link named by the *new* argument exists and the file's link count becomes 0 when it is removed and no process has the file open, the space occupied by the file will be freed and the file will no longer be accessible. If one or more processes have the file open when the last link is removed, the link will be removed before `rename()` returns, but the removal of the file contents will be postponed until all references to the file are closed.

Upon successful completion, `rename()` will mark for update the `st_ctime` and `st_mtime` fields of the parent directory of each file.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, `errno` is set to indicate the error, and neither *old* nor *new* will be changed or created.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

EACCES

A component of either path prefix denies search permission; one of the directories containing *old* or *new* denies write permission; or one of the directories pointed to by *old* or *new* denies write permission.

EBUSY

The directory named by *old* or *new* is currently in use by the system or another process, and the

implementation considers this an error, or the file named by *old* or *new* is a named stream.

EDQUOT

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

EEXIST or ENOTEMPTY

The link named by *new* is a directory containing entries other than "." and "..".

EFAULT

old or *new* points outside the process' allocated address space.

EINVAL

old is a parent directory of *new*, or an attempt is made to rename "." or "..".

EINTR

A signal was caught during execution of the `rename()` system call.

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

EISDIR

new points to a directory but *old* points to a file that is not a directory.

EMLINK

The file named by *old* is a directory, and the link count of the parent directory of *new* would exceed `LINK_MAX`.

ELOOP

Too many symbolic links were encountered in translating *old* or *new*.

EMULTIHOP

Components of pathnames require hopping to multiple remote machines and the file system type does not allow it.

ENAMETOOLONG

The length of the *old* or *new* argument exceeds `PATH_MAX`, or the length of a component of *old* or *new* exceeds `NAME_MAX`.

ENOENT

A component of either *old* or *new* does not exist, or the file referred to by either *old* or *new* does not exist.

ENOLINK

Pathnames point to a remote machine and the link to that machine is no longer active.

ENOSPC

The directory that would contain *new* is out of space.

ENOTDIR

A component of a path prefix is not a directory; or the *old* parameter names a directory and the *new* parameter names a file.

EPERM or EACCES

The `S_ISVTX` flag is set on the directory containing the file referred to by *old* and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges; or *new* refers to an existing file, the `S_ISVTX` flag is set on the directory containing this file and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.

EROFS

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

EXDEV

The links named by *old* and *new* are on different file systems.

The `rename()` function may fail if:

EBUSY

The file named by the *old* or *new* arguments is a named stream.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds

PATH_MAX.

ETXTBSY

The file to be renamed is a pure procedure (shared text) file that is being executed.

NOTES

The system can deadlock if there is a loop in the file system graph. Such a loop takes the form of an entry in directory *a*, say *a/foo*, being a hard link to directory *b*, and an entry in directory *b*, say *b/bar*, being a hard link to directory *a*. When such a loop exists and two separate processes attempt to perform `rename a/foo b/bar` and `rename b/bar a/foo`, respectively, the system may deadlock attempting to lock both directories for modification. The superuser should replace hard links to directories by symbolic links.

SEE ALSO

`link(2)`, `rmdir(2)`, `symlink(2)`, `unlink(2)`, `stdio(5)`.

NAME

`rmdir` - remove a directory

SYNOPSIS

```
#include <unistd.h>
int rmdir(const char *path);
```

DESCRIPTION

`rmdir()` removes the directory named by the pathname pointed to by *path*. The directory must not have any entries other than "." and "..".

If *path* is the root directory or the directory of an active process, the behavior of `rmdir()` is unspecified.

If the directory's link count becomes zero and no process has the directory open, the space occupied by the directory is freed. The directory is no longer accessible. If one or more processes have the directory open when the last link is removed, the "." and ".." entries are removed before `rmdir()` returns and no new entries may be created in the directory, but the directory is not removed until all references to the directory have been closed.

If *path* is a symbolic link, it is not followed.

Upon successful completion `rmdir()` marks for update the `st_ctime` and `st_mtime` fields of the parent directory.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The named directory is removed unless one or more of the following apply:

EACCES

Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be removed.

EACCES

The parent directory has the sticky bit set and is not owned by the user; the directory is not owned by the user and is not writable by the user; the user is not a superuser.

EBUSY

The directory to be removed is currently in use by the system or by another process.

EEXIST or ENOTEMPTY

The directory contains entries in addition to "." and "..".

EFAULT

path points outside the process' allocated address space.

EINVAL

The directory to be removed is the current directory.

EINVAL

The directory to be removed is the "." entry of a directory.

EIO An I/O error occurred while accessing the file system.

ELOOP

Too many symbolic links were encountered in translating *path*.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and the file system does not allow it.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a path component exceeds

NAME_MAX.

ENOTDIR

A component of the path prefix is not a directory.

ENOENT

The named directory does not exist or is the null pathname.

EROFS

The directory entry to be removed is part of a read-only file system.

EPERM or EACCES

The `S_ISVTX` flag is set on the parent directory of the directory to be removed and the caller is not the owner of the directory to be removed, nor is the caller the owner of the parent directory, nor does the caller have the appropriate privileges.

ENOLINK

path points to a remote machine, and the link to that machine is no longer active.

The `rmdir()` function may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

RESULT

Upon successful completion, `rmdir()` returns 0. Otherwise `-1` is returned, and `errno` is set to indicate the error.

SEE ALSO

`mkdir(1)`, `rm(1)`, `rmdir(1)`, `mkdir(2)`, `remove(3C)`, `unistd(4)`.

semctl(2) - sysinfo(2)

NAME

`semctl` - semaphore control functions

SYNOPSIS

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

DESCRIPTION

`semctl()` provides a variety of semaphore control operations as specified by *cmd*. The fourth argument is optional and depends upon the operation requested. If required, it is of type `union semun`, which the application program must explicitly declare:

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} arg;
```

The following commands can be issued as the *cmd*. These commands relate to the semaphore specified by *semid* and *semnum*:

GETVAL

Return the value of `semval`. Requires read permission.

SETVAL

Set `semval` to the value of the fourth argument (*int*). When this command is successfully executed,

the `semadj` value corresponding to the specified semaphore in all processes is cleared. Requires read permission.

GETPID

Return the value of `(int) sempid`. Requires read permission.

GETNCNT

Return the value of `semncnt`. Requires read permission.

GETZCNT

Return the value of `semzcnt`. Requires read permission.

The following commands return and set, respectively, every `semval` in the set of semaphores.

GETALL

Return `semval` and place into the array of type `unsigned short` pointed to by the fourth argument to `semctl()`. Requires read permission.

SETALL

Set `semval` according to the array of type `unsigned short` pointed to by the fourth argument to `semctl()`. When this command is successfully executed, the `semadj` values corresponding to each specified semaphore in all processes are cleared. Requires write permission.

The following commands are also available:

IPC_STAT

Place the current value of each member of the `semid_ds` data structure associated with `semid` into the `semid_ds` structure pointed to by the fourth argument of `semctl()`. Requires read permission.

IPC_SET

Set the value of the following members of the `semid_ds` data structure associated with `semid` to the corresponding value found in the `semid_ds` structure pointed to by the fourth argument of `semctl()`.

`sem_perm.uid`

`sem_perm.gid`

`sem_perm.mode` /* only access permission bits */

This command can be executed only by a process that has an effective user ID equal to either that of superuser, or to the value of `sem_perm.cuid` or `sem_perm.uid` in the data structure associated with `semid`.

IPC_RMID

Remove the semaphore identifier specified by `semid` from the system and destroy the set of semaphores and data structure associated with it. This command can only be executed by a process that has an effective user ID equal to either that of superuser, or to the value of `sem_perm.cuid` or `sem_perm.uid` in the data structure associated with `semid`.

RETURN VALUE

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

GETVAL

The value of `semval`

GETPID

The value of `(int) sempid`

GETNCNT

The value of `semncnt`

GETZCNT

The value of `semzcnt`

All others

A value of 0

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`semctl()` fails if one or more of the following apply:

EACCES

Operation permission is denied to the calling process.

EINVAL

The value of *semid* is not a valid semaphore identifier, or the value of *semnum* is less than 0 or greater than or equal to `sem_nsems`, or the value of *cmd* is not a valid command.

EOVERFLOW

cmd is `IPC_STAT` and *uid* or *gid* is too large to be stored in the structure pointed to by the fourth argument of `semctl()`.

ERANGE

cmd is `SETVAL` or `SETALL` and the value to which *semval* is to be set is greater than the system imposed maximum.

EPERM

cmd is equal to `IPC_RMID` or `IPC_SET` and the effective user ID of the calling process is not equal to that of superuser, or to the value of `sem_perm.cuid` or `sem_perm.uid` in the data structure associated with *semid*.

EFAULT

The fourth argument points to an illegal address.

SEE ALSO

[semget\(2\)](#), [semop\(2\)](#), [sem\(5\)](#).

semget(2)

NAME

semget - get semaphore identifier

SYNOPSIS

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);□
```

DESCRIPTION

semget() returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores are created for *key* if one of the following is true:

- *key* is equal to `IPC_PRIVATE`.
- *key* does not already have a semaphore identifier associated with it, and $(semflg \& IPC\_CREAT)$ is true.

On creation, the data structure associated with the new semaphore identifier is initialized as follows:

- `sem_perm.cuid`, `sem_perm.uid`, `sem_perm.cgid`, and `sem_perm.gid` are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The access permission bits of `sem_perm.mode` are set equal to the access permission bits of *semflg*.
- `sem_nsems` is set equal to the value of *nsems*.
- `sem_otime` is set equal to 0 and `sem_ctime` is set equal to the current time.
- The data structure associated with each semaphore in the set is not initialised. The `semctl()` function with the command `SETVAL` or `SETALL` can be used to initialise each semaphore.

RETURN VALUE

Upon successful completion, a non-negative integer, namely a semaphore identifier, is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

semget() fails if one or more of the following apply:

EACCES

A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted.

EEXIST

A semaphore identifier exists for *key* but both $(semflg \& IPC\_CREAT)$ and $(semflg \& IPC\_EXCL)$ are non-zero.

EINVAL

nsems is either less than or equal to zero or greater than the system-imposed limit, or a semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to 0.

ENOENT

A semaphore identifier does not exist for *key* and $(semflg \& IPC\_CREAT)$ is equal to 0.

ENOSPC

A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores systemwide would be exceeded.

SEE ALSO

```
semctl(2), semop(2), stdipc(3C), sem(5).
```

NAME

semop - semaphore operations

SYNOPSIS

```
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, size_t nsops);
```

DESCRIPTION

semop() is used to perform atomically an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *sops* is a pointer to the array of semaphore-operation structures. *nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
ushort_t sem_num;          /* semaphore number */
short sem_op;              /* semaphore operation */
short sem_flg;             /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

sem_op specifies one of three semaphore operations as follows, depending on whether its value is negative, positive, or zero:

If *sem_op* is a negative integer, one of the following occurs:

- If the value of the semaphore *semval* is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg*&SEM_UNDO) is true, the absolute value of *sem_op* is added to the calling process' *semadj* value [see [exit\(2\)](#)] for the specified semaphore.
- If *semval* is less than the absolute value of *sem_op* and (*sem_flg*&IPC_NOWAIT) is true, **semop()** returns immediately.
- If *semval* is less than the absolute value of *sem_op* and (*sem_flg*&IPC_NOWAIT) is false, **semop()** increments the *semncnt* associated with the specified semaphore and suspends execution of the calling process until one of the following conditions occurs:
 - *semval* becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from *semval* and, if (*sem_flg*&SEM_UNDO) is true, the absolute value of *sem_op* is added to the calling process' *semadj* value for the specified semaphore.
 - The *semid* for which the calling process is awaiting action is removed from the system [see [semctl\(2\)](#)]. When this occurs, *errno* is set equal to EIDRM, and a value of -1 is returned.
 - The calling process receives a signal that is to be caught. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in [signal\(2\)](#).

If *sem_op* is a positive integer, the value of *sem_op* is added to *semval* and, if (*sem_flg*&SEM_UNDO) is true, the value of *sem_op* is subtracted from the calling process' *semadj* value for the specified semaphore.

If *sem_op* is zero, one of the following occurs:

- If *semval* is zero, **semop()** returns immediately.
- If *semval* is not equal to zero and (*sem_flg*&IPC_NOWAIT) is true, **semop()** returns immediately.
- If *semval* is not equal to zero and (*sem_flg*&IPC_NOWAIT) is false, **semop()** increments the *semzcnt* associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

- `semval` becomes zero, at which time the value of `semzcnt` associated with the specified semaphore is decremented.
- The `semid` for which the calling process is awaiting action is removed from the system. When this occurs, `errno` is set equal to `EIDRM`, and a value of `-1` is returned.
- The calling process receives a signal that is to be caught. When this occurs, the value of `semzcnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in `signal(2)`.

Upon successful completion, the value of `sempid` for each semaphore specified in the array pointed to by `sops` is set equal to the process ID of the calling process.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

The `semop()` function will fail if:

`E2BIG`

`nsops` is greater than the system-imposed maximum.

`EACCES`

Operation permission is denied to the calling process.

`EAGAIN`

The operation would result in suspension of the calling process but `(sem_flg & IPC_NOWAIT)` is non-zero.

`EFBIG`

`sem_num` is less than zero or greater than or equal to the number of semaphores in the set associated with `semid`.

`EIDRM`

The semaphore identifier `semid` is removed from the system.

`EINTR`

The `semop()` function was interrupted by a signal.

`EINVAL`

`semid` is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a `SEM_UNDO` will exceed the limit.

`ENOSPC`

The limit on the number of individual processes requesting an `SEM_UNDO` has been exceeded.

`ERANGE`

`semval` exceeds the system-imposed limit, or the `semadj` value exceeds the system-imposed limit.

SEE ALSO

`exec(2)`, `exit(2)`, `fork(2)`, `semctl(2)`, `semget(2)`, `sem(5)`.

setpgid(2)**NAME**

`setpgid` - set process group ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

DESCRIPTION

`setpgid()` sets the process group ID of the process with ID *pid* to *pgid*. If *pgid* is equal to *pid*, the process becomes a process group leader. If *pgid* is not equal to *pid*, the process becomes a member of an existing process group.

If *pid* is equal to 0, the process ID of the calling process is used. If *pgid* is equal to 0, the process specified by *pid* becomes a process group leader.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`setpgid()` returns an error if one or more of the following are true:

EACCES

pid matches the process ID of a child process of the calling process and the child process has successfully executed an [exec\(2\)](#) function.

EINVAL

pgid is less than (pid_t) 0, or greater than or equal to `PID_MAX`.

EINVAL

The calling process has a controlling terminal that does not support job control.

EPERM

The process indicated by the *pid* argument is a session leader.

EPERM

pid matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process.

EPERM

pgid does not match the process ID of the process indicated by the *pid* argument and there is no process with a process group ID that matches *pgid* in the same session as the calling process.

ESRCH

pid does not match the process ID of the calling process or of a child process of the calling process.

RETURN VALUE

Upon successful completion, `setpgid()` returns a value of 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

SEE ALSO

[exec\(2\)](#), [exit\(2\)](#), [fork\(2\)](#), [getpid\(2\)](#), [getpgid\(2\)](#), [setsid\(2\)](#), [unistd\(4\)](#), [types\(5\)](#).

setpgrp(2)**NAME**

`setpgrp` - set process group ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
pid_t setpgrp(void);
```

DESCRIPTION

If the calling process is not already a session leader, `setpgrp()` sets the process group ID and session ID of the calling process to the process ID of the calling process, and releases the calling process' controlling terminal.

RETURN VALUE

`setpgrp()` returns the value of the new process group ID.

NOTES

`setpgrp()` will be phased out in favor of the `setsid(2)` function.

SEE ALSO

`exec(2)`, `fork(2)`, `getpid(2)`, `kill(2)`, `setsid(2)`, `signal(2)`, `unistd(4)`, `types(5)`.

setsid(2)**NAME**

setsid - set session ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
pid_t setsid(void);
```

DESCRIPTION

If the calling process is not already a process group leader, `setsid()` sets the process group ID and session ID of the calling process to the process ID of the calling process, and releases the process' controlling terminal.

`setsid()` will fail and return an error if the following is true:

EPERM

The calling process is already a process group leader, or there are processes other than the calling process whose process group ID is equal to the process ID of the calling process.

RETURN VALUE

Upon successful completion, `setsid()` returns the calling process' session ID. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

NOTES

If the calling process is the last member of a pipeline started by a job control shell, the shell may make the calling process a process group leader. The other processes of the pipeline become members of that process group. In this case, the call to `setsid()` will fail. For this reason, a process that calls `setsid()` and expects to be part of a pipeline should always execute a `fork(2)` first; the parent should exit and the child should call `setsid()`, thereby insuring that the process will work reliably when started by both job control shells and non-job control shells.

SEE ALSO

`exec(2)`, `exit(2)`, `fork(2)`, `getpid(2)`, `getpgid(2)`, `getsid(2)`, `setpgid(2)`, `setpgrp(2)`, `signal(2)`, `sigsend(2)`, `unistd(4)`, `types(5)`.

NAME

`setuid`, `setgid` - set user and group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

DESCRIPTION

The `setuid()` system call sets the real user ID, effective user ID, and saved user ID of the calling process. The `setgid()` system call sets the real group ID, effective group ID, and saved group ID of the calling process.

At login time, the real user ID, effective user ID, and saved user ID of the login process are set to the login ID of the user responsible for the creation of the process. The same is true for the real, effective, and saved group IDs; they are set to the group ID of the user responsible for the creation of the process.

When a process calls `exec(2)` to execute a file, the user and/or group identifiers associated with the process can change. If the file executed is a "set-user-ID" file, the effective and saved user IDs of the process are set to the owner of the file executed. If the file executed is a "set-group-ID" file, the effective and saved group IDs of the process are set to the group of the file executed. If the file executed is not a "set-user-ID" or "setgroup-ID" file, the user ID, saved user ID, effective group ID, and saved group ID are not changed.

`setuid()`

If the effective user ID of the process calling `setuid()` is the superuser, the real, effective, and saved user IDs are set to the *uid*.

If the effective user ID of the calling process is not the superuser, but *uid* is either the real user ID or the saved user ID of the calling process, the effective user ID is set to *uid*.

`setgid()`

If the effective user ID of the process calling `setgid()` is the superuser, the real, effective, and saved group IDs are set to the *gid*.

If the effective user ID of the calling process is not the superuser, but *gid* is either the real group ID or the saved group ID of the calling process, the effective group ID is set to *gid*.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`setuid()` and `setgid()` fail if one or more of the following apply:

EPERM

In the case of `setuid()`, if the effective user ID is not the superuser, and the *uid* parameter does not match either the real or saved user IDs.

In the case of `setgid()`, if the effective user ID is not the superuser, and the *gid* parameter does not match either the real or saved group IDs.

EINVAL

uid or *gid* is out of the valid range.

RETURN VALUE

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

SEE ALSO

`exec(2)`, `getgroups(2)`, `getuid(2)`, `unistd(4)`, `stat(5)`, `types(5)`.

NAME

shmctl - shared memory control functions

SYNOPSIS

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shm_id_ds *buf);
```

DESCRIPTION

shmctl() provides control operations for shared memory. The following control operations can be specified by *cmd*.

IPC_STAT

Places the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of the structure are defined in *sys/shm.h*.

IPC_SET

Sets the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only access permission bits */
```

This function can be executed only by a process that has an effective user ID equal to that of superuser, or to the value of *shm_perm.cuid* or *shm_perm.uid* in the data structure associated with *shmid*.

IPC_RMID

Removes the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This function can be executed only by a process that has an effective user ID equal to that of superuser, or to the value of *shm_perm.cuid* or *shm_perm.uid* in the data structure associated with *shmid*.

SHM_LOCK

Locks the shared memory segment specified by *shmid* in memory. This function can be executed only by a process that has an effective user ID equal to superuser.

SHM_UNLOCK

Unlocks the shared memory segment specified by *shmid*. This function can be executed only by a process that has an effective user ID equal to superuser.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The shmctl() function will fail if:

EACCES

cmd is equal to *IPC_STAT* and operation permission is denied to the calling process.

EINVAL

shmid is not a valid shared memory identifier, or *cmd* is not a valid command.

ENOMEM

cmd is equal to *SHM_LOCK* and there is not enough memory.

EPERM

cmd is equal to `IPC_RMID` or `IPC_SET` and the effective user ID of the calling process is not equal to that of superuser, or to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with *shmid*.

EPERM

cmd is equal to `SHM_LOCK` or `SHM_UNLOCK` and the effective user ID of the calling process is not equal to that of superuser.

The `shmctl()` function may fail if:

EOVERFLOW

cmd is `IPC_STAT` and *uid* or *gid* is too large to be stored in the structure pointed to by *buf*.

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

SEE ALSO

`shmget(2)`, `shmop(2)`, `shm(5)`.

shmget(2)

NAME

shmget - get shared memory segment identifier

SYNOPSIS

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

DESCRIPTION

shmget() returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of at least *size* bytes are created for *key* if one of the following applies:

- *key* is equal to `IPC_PRIVATE`.
- *key* does not already have a shared memory identifier associated with it, and $(shmflg \ \& \ \text{IPC\_CREAT})$ is true.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- `shm_perm.cuid`, `shm_perm.uid`, `shm_perm.cgid`, and `shm_perm.gid` are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The access permission bits of `shm_perm.mode` are set equal to the access permission bits of *shmflg*. `shm_segsz` is set equal to the value of *size*.
- `shm_lpid`, `shm_nattch`, `shm_atime` and `shm_dtime` are set equal to 0.
- `shm_ctime` is set equal to the current time.

RETURN VALUE

Upon successful completion, a non-negative integer, namely a shared memory identifier is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

shmget() fails if one or more of the following apply:

EACCES

A shared memory identifier exists for *key* but operation permission specified by the low-order 9 bits of *shmflg* would not be granted.

EEXIST

A shared memory identifier exists for *key* but both $(shmflg \ \& \ \text{IPC\_CREAT})$ and $(shmflg \ \& \ \text{IPC\_EXCL})$ are true.

EINVAL

size is less than the system-imposed minimum or greater than the system-imposed maximum, or a shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.

ENOENT

A shared memory identifier does not exist for *key* and $(shmflg \ \& \ \text{IPC\_CREAT})$ is false.

ENOMEM

A shared memory identifier and associated shared memory segment are to be created but the amount of available memory is not sufficient to fill the request.

ENOSPC

A shared memory identifier is to be created but the system-imposed limit on the maximum number of

allowed shared memory identifiers systemwide would be exceeded.

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

SEE ALSO

`shmctl(2)`, `shmop(2)`, `stdipc(3C)`, `shm(5)`.

NAME

shmop:
shmat, shmdt - shared memory operations

SYNOPSIS

```
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

DESCRIPTION

shmat() attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

- If *shmaddr* is equal to (void *) 0, the segment is attached at the first available address as selected by the system.
- If *shmaddr* is not equal to (void *) 0 and (*shmflg*&SHM_RND) is true, the segment is attached at the address given by (*shmaddr* - ((ptrdiff_t)shmaddr%SHMLBA)).
- If *shmaddr* is not equal to (void *) 0 and (*shmflg*&SHM_RND) is false, the segment is attached at the address given by *shmaddr*.

shmdt() detaches from the calling process' data segment the shared memory segment located at the address specified by *shmaddr*.

The segment is attached for reading if (*shmflg*&SHM_RDONLY) is true otherwise it is attached for reading and writing.

RETURN VALUE

Upon successful completion, the result is as follows:

The shmat() function increments the value of *shm_nattch* in the data structure associated with the shared memory ID of the attached shared memory segment and returns the segment's start address.

The shmdt() function will decrement the value of *shm_nattch* in the data structure associated with the shared memory ID of the attached shared memory segment and return 0.

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The shmat() function will fail if:

EACCES

Operation permission is denied to the calling process.

EINVAL

shmid is not a valid shared memory identifier; *shmaddr* is not equal to zero, and the value of (*shmaddr* - ((ptrdiff_t)shmaddr%SHMLBA)) is an illegal address; or *shmaddr* is not equal to zero, (*shmflg*&SHM_RND) is false, and the value of *shmaddr* is an illegal address.

EMFILE

The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

ENOMEM

The available data space is not large enough to accommodate the shared memory segment.

The `shmdt()` function will fail if:

EINVAL

`shmdt()` fails and does not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.

EINVAL

This executable is a specially preloaded binary. The argument *shmaddr* lies illegally between the text and data regions.

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

SEE ALSO

`exec(2)`, `exit(2)`, `fork(2)`, `shmctl(2)`, `shmget(2)`, `shm(5)`, `preload(8)`.

NAME

sigaction - get/amend signal actions

SYNOPSIS

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
```

DESCRIPTION

The `sigaction()` function allows the calling process to examine and/or specify the action to be associated with a specific signal. The argument `sig` specifies the signal; acceptable values are defined in `<signal.h>`.
Only valid if generated with "c89":
The structure `sigaction`, used to describe an action to be taken, is defined in the header `<signal.h>` to include at least the following members:

Member Type	Member Name	Description
void(*) (int)	sa_handler	SIG_DFL, SIG_IGN or pointer to a function.
sigset_t	sa_mask	Additional set of signals to be blocked during signal-catching function.
int	sa_flags	Special flags to affect behavior of signal.
void(*) (int, siginfo_t *, void *)	sa_sigaction	Signal-catching function.

Only valid if generated with "cc":

```
void      (*sa_handler)();
sigset_t  sa_mask;
int       sa_flags;
```

If the argument `act` is not a null pointer, it points to a structure specifying the action to be associated with the specified signal. If the argument `oact` is not a null pointer, the action previously associated with the signal is stored in the location pointed to by the argument `oact`. If the argument `act` is a null pointer, signal handling is unchanged; thus, the call can be used to enquire about the current handling of a given signal. The `sa_handler` field of the `sigaction` structure identifies the action to be associated with the specified signal. If the `sa_handler` field specifies a signal-catching function, the `sa_mask` field identifies a set of signals that will be added to the process' signal mask before the signal-catching function is invoked. The `SIGKILL` and `SIGSTOP` signals will not be added to the signal mask using this mechanism; this restriction will be enforced by the system without causing an error to be indicated.

If the `SA_SIGINFO` flag (see below) is cleared in the `sa_flags` field of the `sigaction` structure, the `sa_handler` field identifies the action to be associated with the specified signal. If the `SA_SIGINFO` flag is set in the `sa_flags` field, the `sa_sigaction` field specifies a signal-catching function. If the `SA_SIGINFO` bit is cleared and the `sa_handler` field specifies a signal-catching function, or if the `SA_SIGINFO` bit is set, the `sa_mask` field identifies a set of signals that will be added to the signal mask of the thread before the signal-catching function is invoked.

The `sa_flags` field can be used to modify the behavior of the specified signal.

The following flags, defined in the header `<signal.h>`, can be set in `sa_flags`:

`SA_NOCLDSTOP`
Do not generate `SIGCHLD` when children stop.

`SA_ONSTACK`

If set and an alternate signal stack has been declared with `sigaltstack()` or `sigstack()`, the signal will be delivered to the calling process on that stack. Otherwise, the signal will be delivered on the current stack.

SA_RESETHAND

If set, the disposition of the signal will be reset to `SIG_DFL` and the `SA_SIGINFO` flag will be cleared on entry to the signal handler (*Note: SIGILL and SIGTRAP cannot be automatically reset when delivered; the system silently enforces this restriction*). Otherwise, the disposition of the signal will not be modified on entry to the signal handler.

In addition, if this flag is set, `sigaction()` behaves as if the `SA_NODEFER` flag were also set.

SA_RESTART

This flag affects the behavior of interruptible functions; that is, those specified to fail with `errno` set to `EINTR`. If set, and a function specified as interruptible is interrupted by this signal, the function will restart and will not fail with `EINTR` unless otherwise specified. If the flag is not set, interruptible functions interrupted by this signal will fail with `errno` set to `EINTR`.

SA_SIGINFO

Only valid if generated in XPG4 mode (with "c89"):

If cleared and the signal is caught, the signal-catching function will be entered as:

```
void func(int signo);
```

where *signo* is the only argument to the signal-catching function. In this case the `sa_handler` member must be used to describe the signal-catching function and the application must not modify the `sa_sigaction` member.

If `SA_SIGINFO` is set and the signal is caught, the signal-catching function will be entered as:

```
void func(int signo, siginfo_t *info, void *context);
```

where two additional arguments are passed to the signal-catching function. The second argument will point to an object of type `siginfo_t` explaining the reason why the signal was generated; the third argument can be cast to a pointer to an object of type `ucontext_t` to refer to the receiving process' context that was interrupted when the signal was delivered. In this case the `sa_sigaction` member must be used to describe the signal-catching function and the application must not modify the `sa_handler` member.

Only valid if generated with "cc":

If set and the signal is caught, pending signals of type *sig* are reliably queued to the calling process and two additional arguments are passed to the signal-catching function. If the second argument is not equal to `NULL`, it points to a `siginfo_t` structure containing the reason why the signal was generated [see [siginfo\(5\)](#)]; the third argument points to a `ucontext_t` structure containing the receiving process' context when the signal was delivered [see [ucontext\(5\)](#)].

Valid again below for both compilers:

The `si_signo` member contains the system-generated signal number.

The `si_errno` member may contain implementation-dependent additional error information; if non-zero, it contains an error number identifying the condition that caused the signal to be generated.

The `si_code` member contains a code identifying the cause of the signal. If the value of `si_code` is less than or equal to 0, then the signal was generated by a process and `si_pid` and `si_uid` respectively indicate the process ID and the real user ID of the sender. The `<signal.h>` header description contains information about the signal specific contents of the elements of the `siginfo_t` type.

SA_NOCLDWAIT

If set, and *sig* equals `SIGCHLD`, child processes of the calling processes will not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate, and `wait()`, `wait3()`, `waitid()` and `waitpid()` will fail and set `errno` to `ECHILD`. Otherwise, terminating child processes will be transformed into zombie processes, unless

SIGCHLD is set to SIG_IGN.

SA_NODEFER

If set and *sig* is caught, *sig* will not be added to the process' signal mask on entry to the signal handler unless it is included in *sa_mask*. Otherwise, *sig* will always be added to the process' signal mask on entry to the signal handler.

If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is not set in *sa_flags*, and the implementation supports the SIGCHLD signal, then a SIGCHLD signal will be generated for the calling process whenever any of its child processes stop. If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is set in *sa_flags*, then the implementation will not generate a SIGCHLD signal in this way.

When a signal is caught by a signal-catching function installed by `sigaction()`, a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either `sigprocmask()` or `sigsuspend()` is made). This mask is formed by taking the union of the current signal mask and the value of the *sa_mask* for the signal being delivered unless SA_NODEFER or SA_RESETHAND is set, and then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to `sigaction()`), until the SA_RESETHAND flag causes resetting of the handler, or until one of the `exec` functions is called.

If the previous action for *sig* had been established by `signal()`, the values of the fields returned in the structure pointed to by *oact* are unspecified, and in particular *oact*→*sa_handler* is not necessarily the same value passed to `signal()`. However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to `sigaction()` via the *act* argument, handling of the signal will be as if the original call to `signal()` were repeated.

If `sigaction()` fails, no new signal handler is installed.

It is unspecified whether an attempt to set the action for a signal that cannot be caught or ignored to SIG_DFL is ignored or causes an error to be returned with `errno` set to EINVAL.

Signal Generation and Delivery

A signal is said to be *generated* for (or sent to) a process or thread when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, and terminal activity, as well as invocations of `kill()`. In some circumstances, the same event generates signals for multiple processes.

At the time of generation, a determination is made whether the signal has been generated for the process or for a specific thread within the process. Signals which are generated by some action attributable to a particular thread, such as a hardware fault, are generated for the thread that caused the signal to be generated. Signals that are generated in association with a process ID or process group ID or an asynchronous event such as terminal activity are generated for the process.

Each process has an action to be taken in response to each signal defined by the system (see **Signal Actions**). A signal is said to be *delivered* to a process when the appropriate action for the process and signal is taken.

During the time between the generation of a signal and its delivery or acceptance, the signal is said to be *pending*. Ordinarily, this interval cannot be detected by an application. However, a signal can be *blocked* from delivery to a thread. If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the thread the signal will remain pending until it is unblocked, it is accepted when it is selected and returned by a call to the `sigwait()` function, or the action associated with it is set to ignore the signal. Signals generated for the process will be delivered to exactly one of those threads within the process which is in a call to a `sigwait()` function selecting that signal or has not blocked delivery of the signal. If there are no threads in a call to a `sigwait()` function selecting that signal, and if all threads within the process block delivery of the signal, the signal will remain pending on the process until a thread calls a `sigwait()` function selecting that signal, a thread unblocks delivery of the signal, or the action associated with the signal is set to ignore the signal. If the action associated with a blocked signal is to ignore the signal

and if that signal is generated for the process, it is unspecified whether the signal is discarded immediately upon generation or remains pending.

Each thread has a *signal mask* that defines the set of signals currently blocked from delivery to it. The signal mask for a thread is initialised from that of its parent or creating thread, or from the corresponding thread in the parent process if the thread was created as the result of a call to `fork()`. The `sigaction()`, `sigprocmask()` and `sigsuspend()` functions control the manipulation of the signal mask.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated. If a subsequent occurrence of a pending signal is generated, it is implementation-dependent as to whether the signal is delivered more than once. The order in which multiple, simultaneously pending signals are delivered to a process is unspecified.

When any stop signal (`SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`) is generated for a process, any pending `SIGCONT` signals for that process will be discarded. Conversely, when `SIGCONT` is generated for a process, all pending stop signals for that process will be discarded. When `SIGCONT` is generated for a process that is stopped, the process will be continued, even if the `SIGCONT` signal is blocked or ignored. If `SIGCONT` is blocked and not ignored, it will remain pending until it is either unblocked or a stop signal is generated for the process.

Signal Actions

There are three types of action that can be associated with a signal: `SIG_DFL`, `SIG_IGN` or a *pointer to a function*. Initially, all signals will be set to `SIG_DFL` or `SIG_IGN` prior to entry of the `main()` routine (see the `exec` functions). The actions prescribed by these values are as follows:

`SIG_DFL` – signal-specific default action

- The default actions for the signals defined in this description are specified in `<signal.h>`.
- If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a `SIGCHLD` signal will be generated for its parent process, unless the parent process has set the `SA_NOCLDSTOP` flag. While a process is stopped, any additional signals that are sent to the process will not be delivered until the process is continued, except `SIGKILL` which always terminates the receiving process. A process that is a member of an orphaned process group will not be allowed to stop in response to the `SIGTSTP`, `SIGTTIN` or `SIGTTOU` signals. In cases where delivery of one of these signals would stop such a process, the signal will be discarded.
- Setting a signal action to `SIG_DFL` for a signal that is pending, and whose default action is to ignore the signal (for example, `SIGCHLD`), will cause the pending signal to be discarded, whether or not it is blocked.

`SIG_IGN` – ignore signal

- Delivery of the signal will have no effect on the process. The behavior of a process is undefined after it ignores a `SIGFPE`, `SIGILL` or `SIGSEGV` signal that was not generated by `kill()` or `raise()`.
- The system will not allow the action for the signals `SIGKILL` or `SIGSTOP` to be set to `SIG_IGN`.
- Setting a signal action to `SIG_IGN` for a signal that is pending will cause the pending signal to be discarded, whether or not it is blocked.
- If a process sets the action for the `SIGCHLD` signal to `SIG_IGN`, the behavior is unspecified, except as specified below.

If the action for the `SIGCHLD` signal is set to `SIG_IGN`, child processes of the calling processes will not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate, and `wait()`, `wait3()`, `waitid()` and `waitpid()` will fail and set `errno` to `ECHILD`.

pointer to a function – catch signal

- On delivery of the signal, the receiving process is to execute the signal-catching function at the

specified address. After returning from the signal-catching function, the receiving process will resume execution at the point at which it was interrupted.

- If SA_SIGINFO is cleared, the signal-catching function will be entered as:

```
void func(int signo);
```

 where `func` is the specified signal-catching function and `signo` is the signal number of the signal being delivered.
 - If SA_SIGINFO is set, the signal-catching function will be entered as (*only valid in XPG4/5 mode*):

```
void func(int signo, siginfo_t *siginfo, void *ucontextptr);
```

 where `func` is the specified signal-catching function, `signo` is the signal number of the signal being delivered, `siginfo` points to an object of type `siginfo_t` associated with the signal being delivered, and `ucontextptr` points to a `ucontext_t`.
 - The behavior of a process is undefined after it returns normally from a signal-catching function for a SIGBUS, SIGFPE, SIGILL or SIGSEGV signal that was not generated by `kill()` or `raise()`.
 - The system will not allow a process to catch the signals SIGKILL and SIGSTOP.
 - If a process establishes a signal-catching function for the SIGCHLD signal while it has a terminated child process for which it has not waited, it is unspecified whether a SIGCHLD signal is generated to indicate that child process.
 - When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined is unspecified if they are called from a signal-catching function.
- The following table defines a set of functions that are either reentrant or not interruptible by signals. Therefore applications may invoke them, without restriction, from signal-catching functions: □

<code>access()</code>	<code>getegid()</code>	<code>rmdir()</code>	<code>tcdrain()</code>
<code>alarm()</code>	<code>geteuid()</code>	<code>setgid()</code>	<code>tcflow()</code>
<code>cfgetispeed()</code>	<code>getgid()</code>	<code>setpgid()</code>	<code>tcflush()</code>
<code>cfgetospeed()</code>	<code>getgroups()</code>	<code>setsid()</code>	<code>tcgetattr()</code>
<code>cfsetispeed()</code>	<code>getpgrp()</code>	<code>setuid()</code>	<code>tcgetpgrp()</code>
<code>cfsetospeed()</code>	<code>getpid()</code>	<code>sigaction()</code>	<code>tcsendbreak()</code>
<code>chdir()</code>	<code>getppid()</code>	<code>sigaddset()</code>	<code>tcsetattr()</code>
<code>chmod()</code>	<code>getuid()</code>	<code>sigdelset()</code>	<code>tcsetpgrp()</code>
<code>chown()</code>	<code>kill()</code>	<code>sigemptyset()</code>	<code>time()</code>
<code>close()</code>	<code>link()</code>	<code>sigfillset()</code>	<code>times()</code>
<code>creat()</code>	<code>lseek()</code>	<code>sigismember()</code>	<code>umask()</code>
<code>dup()</code>	<code>mkdir()</code>	<code>signal()</code>	<code>uname()</code>
<code>dup2()</code>	<code>mkfifo()</code>	<code>sigpause()</code>	<code>unlink()</code>
<code>execle()</code>	<code>open()</code>	<code>sigpending()</code>	<code>utime()</code>
<code>execve()</code>	<code>pathconf()</code>	<code>sigprocmask()</code>	<code>wait()</code>
<code>_exit()</code>	<code>pause()</code>	<code>sigset()</code>	<code>waitpid()</code>
<code>fcntl()</code>	<code>pipe()</code>	<code>sigsuspend()</code>	<code>write()</code>
<code>fork()</code>	<code>raise()</code>	<code>sleep()</code>	
<code>fpathconf()</code>	<code>read()</code>	<code>stat()</code>	
<code>fstat()</code>	<code>rename()</code>	<code>sysconf()</code>	

All functions not in the above table are considered to be unsafe with respect to signals. In the presence of signals, all functions will behave as defined when called from or interrupted by a signal-catching

function, with a single exception: when a signal interrupts an unsafe function and the signal-catching function calls an unsafe function, the behavior is undefined.

When a signal is delivered to a thread, if the action of that signal specifies termination, stop, or continue, the entire process will be terminated, stopped, or continued, respectively.

Signal Effects on Other Functions

Signals affect the behavior of certain functions defined by this description if delivered to a process while it is executing such a function. If the action of the signal is to terminate the process, the process will be terminated and the function will not return. If the action of the signal is to stop the process, the process will stop until continued or terminated. Generation of a `SIGCONT` signal for the process causes the process to be continued, and the original function will continue at the point the process was stopped. If the action of the signal is to invoke a signal-catching function, the signal-catching function will be invoked; in this case the original function is said to be *interrupted* by the signal. If the signal-catching function executes a `return` statement, the behavior of the interrupted function will be as described individually for that function. Signals that are ignored will not affect the behavior of any function; signals that are blocked will not affect the behavior of any function until they are unblocked and then delivered.

RETURN VALUE

Upon successful completion, `sigaction()` returns 0. Otherwise `-1` is returned, `errno` is set to indicate the error and no new signal-catching function will be installed.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `sigaction()` function will fail if:

`EINVAL`

The `sig` argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

The `sigaction()` function may fail if:

`EINVAL`

An attempt was made to set the action to `SIG_DFL` for a signal that cannot be caught or ignored (or both).

APPLICATION USAGE

The `sigaction()` function supersedes the `signal()` interface, and should be used in preference. In particular, `sigaction()` and `signal()` should not be used in the same process to control the same signal. The behavior of reentrant functions, as defined in the description, is as specified by this description, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that reentrant functions may be used in signal-catching functions without restrictions. Applications must still consider all effects of such functions on such things as data structures, files and process state. In particular, application writers need to consider the restrictions on interactions when interrupting `sleep()` and interactions among multiple handles for a file description. The fact that any specific function is listed as reentrant does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore. This description does not address the more general problem of synchronizing access to shared data structures. Note in particular that even the "safe" functions may modify the global variable `errno`; the signal-catching function may want to save and restore its value. Naturally, the same principles apply to the reentrancy of application routines and asynchronous data access. Note that `longjmp()` and `siglongjmp()` are not in the list of reentrant functions. This is because the code executing after `longjmp()` and `siglongjmp()` can call any unsafe functions with the same danger as calling those unsafe functions directly.

from the signal handler. Applications that use `longjmp()` and `siglongjmp()` from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either `malloc()` or `free()` functions or the standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Because any combination of different functions using a common data structure can cause reentrancy problems, it is not defined the behavior when any unsafe function is called in a signal handler that interrupts an unsafe function.

If the signal occurs other than as the result of calling `abort()`, `kill()` or `raise()`, the behavior is undefined if the signal handler calls any function in the standard library other than one of the functions listed in the table above or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type `volatile sig_atomic_t`. Furthermore, if such a call fails, the value of `errno` is indeterminate.

Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving process will resume execution at the point it was interrupted unless the signal handler makes other arrangements. If `longjmp()` or `_longjmp()` is used to leave the signal handler, then the signal mask must be explicitly restored by the process.

POSIX.4-1993 defines the third argument of a signal handling function when `SA_SIGINFO` is set as a `void*` instead of a `ucontext_t*`, but without requiring type checking. New applications should explicitly cast the third argument of the signal handling function to `ucontext_t*`.

The BSD optional four argument signal handling function is not supported by this specification. The BSD declaration would be

```
void handler(int sig, int code, struct sigcontext *scp,
             char *addr);
```

where `sig` is the signal number, `code` is additional information on certain signals, `scp` is a pointer to the `sigcontext` structure, and `addr` is additional address information. Much the same information is available in the objects pointed to by the second argument of the signal handler specified when `SA_SIGINFO` is set.

SEE ALSO

`kill(1)`, `exit(2)`, `intro_prm2(2)`, `kill(2)`, `pause(2)`, `sigaltstack(2)`, `signal(2)`, `sigprocmask(2)`, `sigsend(2)`, `sigsuspend(2)`, `wait(2)`, `waitid(2)`, `waitpid(2)`, `raise(3C)`, `setjmp(3C-ucb)`, `sigsetops(3C)`, `wait3(3C)`, `siginfo(5)`, `signal(5)`, `ucontext(5)`.

Manual "C-Compiler", chapter "The cc/c89 command".

sigaltstack(2)**NAME**

sigaltstack - set or get signal alternate stack context

SYNOPSIS

```
#include <signal.h>

int sigaltstack(const stack_t *ss, stack_t *oss);
```

DESCRIPTION

sigaltstack() is used to define an alternate stack area on which signals are to be processed. If *ss* is non-zero, it specifies a pointer to a stack area on which to deliver signals, and tells the system if the process is currently executing on that stack. When a signal's action indicates its handler should execute on the alternate signal stack (specified with a **sigaction(2)** call), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the alternate signal stack for the duration of the signal handler's execution.

The structure **sigaltstack** includes the following members.

```
char    *ss_sp
int      ss_size
int      ss_flags
```

If *ss* is not NULL, it points to a structure specifying the alternate signal stack that will take effect upon return from **sigaltstack()**. The *ss_sp* and *ss_size* fields specify the new base and size of the stack, which is automatically adjusted for direction of growth and alignment. The *ss_flags* field specifies the new stack state and may be set to the following:

SS_DISABLE

The stack is to be disabled and *ss_sp* and *ss_size* are ignored. If **SS_DISABLE** is not set, the stack will be enabled.

If *oss* is not NULL, it points to a structure specifying the alternate signal stack that was in effect prior to the call to **sigaltstack()**. The *ss_sp* and *ss_size* fields specify the base and size of that stack. The *ss_flags* field specifies the stack's state, and may contain the following values:

SS_ONSTACK

The process is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the process is executing on it will fail.

SS_DISABLE

The alternate signal stack is currently disabled.

The value **SIGSTKSZ** is a system default specifying the number of bytes that would be used to cover the usual case when manually allocating an alternate stack area. The value **MINSIGSTKSZ** is defined to be the minimum stack size for a signal handler. In computing an alternate stack size, a program should add that amount to its stack requirements to allow for the system implementation overhead. The constants **SS_ONSTACK**, **SS_DISABLE**, **SIGSTKSZ**, and **MINSIGSTKSZ** are defined in **signal(5)**.

After a successful call to one of the **exec** functions, there are no alternate signal stacks in the new process image.

In some implementations, a signal (whether or not indicated to execute on the alternate stack) will always execute on the alternate stack if it is delivered while another signal is being caught using the alternate stack.

Use of this function by library threads that are not bound to kernel-scheduled entities results in undefined behaviour.

RETURN VALUE

On success, **sigaltstack()** returns 0. On failure, it returns -1 and sets **errno** to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`sigaltstack()` fails if any of the following are true:

EFAULT

`ss` or `oss` points outside the process' allocated address space.

EINVAL

The `ss` argument is not a null pointer, and the `ss_flags` member pointed to by `ss` contains flags other than `SS_DISABLE`.

ENOMEM

The size of the alternate stack area is less than `MINSIGSTKSZ`.

EPERM

An attempt was made to modify an active stack.

NOTES

The following code fragment is typically used to allocate an alternate stack.

```
if ((sigstk.ss_sp = (char *)malloc(SIGSTKSZ)) == NULL)
    /* error return */;
sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk, (stack_t *)0) < 0)
    perror("sigaltstack");
```

SEE ALSO

[getcontext\(2\)](#), [sigaction\(2\)](#), [sigsetjmp\(3C\)](#), [signal\(5\)](#), [ucontext\(5\)](#).

NAME

signal, sigset, sighold, sigrelse, sigignore, sigpause - simplified signal management

SYNOPSIS

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
int sighold(int sig);
int sigignore(int sig);
int sigpause(int sig);
int sigrelse(int sig);
void (*sigset(int sig, void (*disp)(int)))(int);
```

DESCRIPTION

Use of any of these functions is unspecified in a multi-threaded process.

The `signal()` function chooses one of three ways in which receipt of the signal number `sig` is to be subsequently handled. If the value of `func` is `SIG_DFL`, default handling for that signal will occur. If the value of `func` is `SIG_IGN`, the signal will be ignored. Otherwise, `func` must point to a function to be called when that signal occurs. Such a function is called a *signal handler*.

When a signal occurs, if `func` points to a function, first the equivalent of a:

```
signal(sig, SIG_DFL);
```

is executed or an implementation-dependent blocking of the signal is performed. (If the value of `sig` is `SIGILL`, whether the reset to `SIG_DFL` occurs is implementation-dependent.) Next the equivalent of:

```
(*func)(sig);
```

is executed. The `func` function may terminate by executing a `return` statement or by calling `abort()`, `exit()`, or `longjmp()`. If `func()` executes a `return` statement and the value of `sig` was `SIGFPE` or any other implementation-dependent value corresponding to a computational exception, the behavior is undefined. Otherwise, the program will resume execution at the point it was interrupted.

If the signal occurs other than as the result of calling `abort()`, `kill()` or `raise()`, the behavior is undefined if the signal handler calls any function in the standard library other than one of the functions listed on the `sigaction()` page or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type `volatile sig_atomic_t`. Furthermore, if such a call fails, the value of `errno` is indeterminate.

At program startup, the equivalent of:

```
signal(sig, SIG_IGN);
```

is executed for some signals, and the equivalent of:

```
signal(sig, SIG_DFL);
```

is executed for all other signals [see [exec\(2\)](#)].

These functions provide simplified signal management for application processes. See [signal\(5\)](#) for an explanation of general signal concepts.

The `sigset()` function is used to modify signal dispositions. `sig` specifies the signal, which may be any signal except `SIGKILL` and `SIGSTOP`. `disp` specifies the signal's disposition, which may be `SIG_DFL`, `SIG_IGN`, or the address of a signal handler. If `sigset()` is used and `disp` is the address of a signal handler, the system adds `sig` to the calling process' signal mask before executing the signal handler; when the signal handler returns, the system restores the calling process' 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' signal mask

and the signal's disposition remains unchanged. If `sigset()` is used, and `disp` is not equal to `SIG_HOLD`, `sig` will be removed from the calling process' signal mask.

The `sighold()` function adds `sig` to the calling process' signal mask.

The `sigrelse()` function removes `sig` from the calling process' signal mask.

The `sigignore()` function sets the disposition of `sig` to `SIG_IGN`.

The `sigpause()` function removes `sig` from the calling process' signal mask and suspends the calling process until a signal is received. The `sigpause()` function restores the process' signal mask to its original state before returning.

If the action for the `SIGCHLD` signal is set to `SIG_IGN`, child processes of the calling processes will not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate, and `wait()`, `wait3()`, `waitid()` and `waitpid()` will fail and set `errno` to `ECHILD`.

RETURN VALUE

If the request can be honoured, `signal()` returns the value of `func` for the most recent call to `signal()` for the specified signal `sig`. Otherwise, `SIG_ERR` is returned and a positive value is stored in `errno`.

On success, `sigset()` returns `SIG_HOLD` if the signal had been blocked or the signal's previous disposition if it had not been blocked. On failure, it returns `SIG_ERR` and sets `errno` to indicate the error.

The `sigpause()` function suspends execution of the thread until a signal is received, whereupon it returns `-1` and sets `errno` to `EINTR`.

All other functions return zero on success. On failure, they return `-1` and set `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `signal()` function will fail if:

`EINVAL`

The value of the `sig` argument is not a valid signal or is equal to `SIGKILL` or `SIGSTOP`.

The `signal()` function may fail if:

`EINVAL`

An attempt was made to set the action to `SIG_DFL` for a signal that cannot be caught or ignored (or both).

The `sigset()`, `sighold()`, `sigrelse()`, `sigignore()` and `sigpause()` functions will fail if:

`EINVAL`

The `sig` argument is an illegal signal number.

The `sigset()`, and `sigignore()` functions will fail if:

`EINVAL`

An attempt is made to catch a signal that cannot be caught, or to ignore a signal that cannot be ignored.

NOTES

The `sigaction()` function provides a more comprehensive and reliable mechanism for controlling signals. New applications should use `sigaction()` instead of `signal()`.

`sighold()` 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' children are stopped or continued.

SEE ALSO

```
kill(2), pause(2), sigaction(2), sigsend(2), wait(2), waitid(2), signal(5).
```

sigpending(2)**NAME**

`sigpending` - examine signals that are blocked and pending

SYNOPSIS

```
#include <signal.h>
int sigpending(sigset_t *set);
```

DESCRIPTION

The `sigpending()` function stores, in the location referenced by the `set` argument, the set of signals that are blocked from delivery to the calling thread and that are pending on the process or the calling thread.

RETURN VALUE

On success, `sigpending()` returns 0. On failure, it returns `-1` and sets `errno` to indicate the error.

ERRORS

The following error code description is function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`sigpending()` fails if the following is true:

EFAULT

`set` points outside the process' allocated address space.

SEE ALSO

[sigaction\(2\)](#), [sigprocmask\(2\)](#), [sigsetops\(3C\)](#), [signal\(5\)](#).

sigprocmask(2)**NAME**

`sigprocmask`, `pthread_sigmask` - examine and change blocked signals

SYNOPSIS

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

DESCRIPTION

In a single-threaded process, the `sigprocmask()` function allows the calling process to examine or change (or both) the signal mask of the calling thread.

If the argument `set` is not a null pointer, it points to a set of signals to be used to change the currently blocked set.

The argument `how` indicates the way in which the set is changed, and consists of one of the following values:

`SIG_BLOCK`

The resulting set will be the union of the current set and the signal set pointed to by `set`.

`SIG_SETMASK`

The resulting set will be the signal set pointed to by `set`.

`SIG_UNBLOCK`

The resulting set will be the intersection of the current set and the complement of the signal set pointed to by `set`.

If the argument `oset` is not a null pointer, the previous mask is stored in the location pointed to by `oset`. If `set` is a null pointer, the value of the argument `how` is not significant and the process' signal mask is unchanged; thus the call can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to `sigprocmask()`, at least one of those signals will be delivered before the call to `sigprocmask()` returns.

It is not possible to block those signals which cannot be ignored. This is enforced by the system without causing an error to be indicated.

If any of the `SIGFPE`, `SIGILL`, `SIGSEGV` or `SIGBUS` signals are generated while they are blocked, the result is undefined, unless the signal was generated by a function capable of sending a signal to a specific process or thread.

If `sigprocmask()` fails, the thread's signal mask is not changed.

The use of the `sigprocmask()` function is unspecified in a multi-threaded process.

The `pthread_sigmask()` function is used to examine or change (or both) the calling thread's signal mask, regardless of the number of threads in the process. The effect is the same as described for `sigprocmask()`, without the restriction that the call be made in a single-threaded process.

RETURN VALUE

Upon successful completion, `sigprocmask()` returns 0. Otherwise `-1` is returned, `errno` is set to indicate the error and the process' signal mask will be unchanged.

Upon successful completion `pthread_sigmask()` returns 0; otherwise it returns the corresponding error number.

ERRORS

The following error code description is function-specific. You will find a general description in

`intro_prm2(2)` or in `errno(5)`.

The `sigprocmask()` and `pthread_sigmask()` functions will fail if:

`EINVAL`

The value of the *how* argument is not equal to one of the defined values.

The `pthread_sigmask()` function will not return an error code of `EINTR`.

SEE ALSO

`sigaction(2)`, `sigpending(2)`, `sigsuspend(2)`, `sigsetops(3C)`, `pthread_intro(3T)`, `signal(5)`.

NAME

`sigsend`, `sigsendset` - send a signal to a process or group of processes

SYNOPSIS

```
#include <sys/types.h>
#include <sys/signal.h>
#include <sys/procset.h>
int sigsend(idtype_t idtype, id_t id, int sig);
int sigsendset(procset_t *psp, int sig);
```

DESCRIPTION

`sigsend()` sends a signal to the process or group of processes specified by `id` and `idtype`. The signal to be sent is specified by `sig` and is either zero or one of the values listed in [signal\(5\)](#). If `sig` is zero (the null signal), error checking is performed but no signal is actually sent. This value can be used to check the validity of `id` and `idtype`.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is superuser, or `sig` is `SIGCONT` and the sending process has the same session ID as the receiving process.

The possible values of `idtype` have the following effects:

`P_PID`

`sig` is sent to the process with process ID `id`.

`P_PPID`

`sig` is sent to the parent process and `id` is ignored.

`P_PGID`

`sig` is sent to any process with process group ID `id`.

`P_SID`

`sig` is sent to any process with session ID `id`.

`P_UID`

`sig` is sent to any process with effective user ID `id`.

`P_GID`

`sig` is sent to any process with effective group ID `id`.

`P_CID`

`sig` is sent to any process with scheduler class ID `id` [see [prioctl\(2\)](#)].

`P_ALL`

`sig` is sent to all processes and `id` is ignored.

`P_MYID`

The value of `id` is taken from the calling process.

The process with a process ID of 0 is always excluded. The process with a process ID of 1 is excluded unless `idtype` is equal to `P_PID`.

`sigsendset()` provides an alternate interface for sending signals to sets of processes. This function sends signals to the set of processes specified by `psp`. `psp` is a pointer to a structure of type `procset_t`, defined in `sys/procset.h`, which includes the following members:

```
idop_t      p_op;
dtype_t     p_lidtype;
id_t        p_lid;
idtype_t    p_ridtype;
id_t        p_rid;
```

`p_lidtype` and `p_lid` specify the ID type and ID of one ("left") set of processes; `p_ridtype` and `p_rid` specify the ID type and ID of a second ("right") set of processes. ID types and IDs are specified just as for the `idtype` and `id` arguments to `sigsend()`. `p_op` specifies the operation to be performed on the two sets of processes to get the set of processes the system call is to apply to. The valid values for `p_op` and the processes they specify are:

`POP_DIFF`

Set difference: processes in left set and not in right set.

`POP_AND`

Set intersection: processes in both left and right sets.

`POP_OR`

Set union: processes in either left or right set or both.

`POP_XOR`

Set exclusive-OR: processes in left or right set but not in both.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`sigsend()` and `sigsendset()` fail if one or more of the following apply:

`EINVAL`

`sig` is not a valid signal number.

`EINVAL`

`idtype` is not a valid `idtype` field.

`EINVAL`

`sig` is `SIGKILL`, `idtype` is `P_PID` and `id` is 1 (`proc1`).

`ESRCH`

No process can be found corresponding to that specified by `id` and `idtype`.

`EPERM`

The user ID of the sending process is not superuser, and its real or effective user ID does not match the real or effective user ID of the receiving process, and the calling process is not sending `SIGCONT` to a process that shares the same session.

In addition, `sigsendset()` fails if:

`EFAULT`

`psp` points outside the process' allocated address space.

RETURN VALUE

On success, `sigsend()` returns 0. On failure, it returns `-1` and sets `errno` to indicate the error.

SEE ALSO

`kill(1)`, `getpgrp(2)`, `getpid(2)`, `kill(2)`, `prctl(2)`, `setpgid(2)`, `signal(2)`, `signal(5)`, `types(5)`.

sigsuspend(2)**NAME**

`sigsuspend` - install a signal mask and suspend process

SYNOPSIS

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

DESCRIPTION

The `sigsuspend()` function replaces the current signal mask of the calling thread with the set of signals pointed to by *sigmask* and then suspends the thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. This will not cause any other signals that may have been pending on the process to become pending on the thread.

If the action is to terminate the process, `sigsuspend()` does not return. If the action is to execute a signal catching function, `sigsuspend()` returns after the signal catching function returns. On return, the signal mask is restored to the set that existed before the `sigsuspend()` call.

It is not possible to block those signals that cannot be ignored [see [signal\(5\)](#)]. This is enforced by the system without causing an error to be indicated.

RETURN VALUE

Since `sigsuspend()` suspends process execution indefinitely, there is no successful completion result. On failure, it returns `-1` and sets `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`sigsuspend()` fails if either of the following is true:

EINTR

A signal is caught by the calling process and control is returned from the signal catching function.

EFAULT

set points outside the process' allocated address space.

SEE ALSO

[sigaction\(2\)](#), [sigpause\(2\)](#), [sigprocmask\(2\)](#), [sigsetops\(3C\)](#), [signal\(5\)](#).

NAME

stat, lstat, fstat, stat64, lstat64, fstat64 - get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
int fstat64(int fildes, struct stat64 *buf);
int lstat64(const char *, struct stat64 *buf);
int stat64(const char *, struct stat64 *buf); □
```

DESCRIPTION

The `stat()` function obtains information about the named file and writes it to the structure pointed to by `buf`. `path` points to a pathname naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

`lstat()` obtains file attributes similar to `stat()`, except when the named file is a symbolic link; in that case `lstat()` returns information about the link, while `stat()` returns information about the file the link references.

`fstat()` obtains information about an open file known by the file descriptor `fildes`, obtained from a successful `open()`, `creat()`, `dup()`, `fcntl()`, or `pipe()` system call.

`buf` is a pointer to a `stat()` structure into which information is placed concerning the file.

There is no functional difference between `stat()/lstat()/fstat()` and `stat64()/lstat64()/fstat64()`, except for the interpretation of `stat/stat64` [see [stat\(5\)](#)].

The contents of the `stat` structure pointed to by `buf` include the following members:

```
mode_t      st_mode;      /* File mode (see mknod(2)) */
ino_t       st_ino;       /* Inode number */
dev_t       st_dev;       /* ID of device containing */
               /* a directory entry for this file */
dev_t       st_rdev;      /* ID of device */
               /* This entry is defined only for */
               /* char special or block special files */
nlink_t     st_nlink;     /* Number of links */
uid_t       st_uid;       /* User ID of the file's owner */
gid_t       st_gid;       /* Group ID of the file's group */
off_t       st_size;      /* File size in bytes */
time_t      st_atime;     /* Time of last access */
time_t      st_mtime;     /* Time of last data modification */
time_t      st_ctime;     /* Time of last file status change */
               /* Times measured in seconds since */
               /* 00:00:00 UTC, Jan. 1, 1970 */
long        st_blksize;   /* Preferred I/O block size */
blkcnt_t    st_blocks;    /* Number 512 bytes blocks allocated */
```

The `stat64` structure is defined in the same way as the `stat` structure with the exception of the following members:

```
ino64_t     st_ino
    File serial number.
```

`off64_t st_size`
File size in bytes.

`blkcnt64_t st_blocks`
Number of blocks allocated for this object.

`st_mode`
The mode of the file as described in `mknod(2)`. In addition to the modes described in `mknod(2)`, the mode of a file may also be `S_IFLNK` if the file is a symbolic link. (Note that `S_IFLNK` may only be returned by `lstat()`.)

`st_ino`
Uniquely identifies the file in a given file system. The pair `st_ino` and `st_dev` uniquely identifies regular files.

`st_dev`
Uniquely identifies the file system that contains the file. Its value may be used as input to the `ustat(2)` system call to determine more information about this file system.

`st_rdev`
Should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.

`st_nlink`
May be used only by administrative commands.

`st_uid`
The user ID of the file's owner.

`st_gid`
The group ID of the file's group.

`st_size`
For regular files, this is the address of the end of the file. For block special or character special, this is not defined. See also `pipe(2)`.

`st_atime`
Time when file data was last accessed. Changed by the following system calls: `creat()`, `mknod()`, `pipe()`, `utime()`, and `read()`.

`st_mtime`
Time when data was last modified. Changed by the following system calls: `creat()`, `mknod()`, `pipe()`, `utime()`, and `write()`.

`st_ctime`
Time when file status was last changed. Changed by the following system calls: `chmod()`, `chown()`, `creat()`, `link()`, `mknod()`, `pipe()`, `unlink()`, `utime()`, and `write()`.

`st_blksize`
A hint as to the "best" unit size for I/O operations. This field is not defined for block special or character special files.

`st_blocks`
The total number of physical blocks of size 512 bytes actually allocated on disk. This field is not defined for block special or character special files.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`stat()` and `lstat()` fail if one or more of the following apply:

EACCES

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

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

ELOOP

Too many symbolic links were encountered in translating *path*.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and the file system does not allow it.

ENAMETOOLONG

The length of a *path* component exceeds `NAME_MAX` characters, or the length of the *path* exceeds `PATH_MAX` characters.

ENOENT

The named file does not exist or is the null pathname.

ENOTDIR

A component of the path prefix is not a directory.

ENOLINK

path points to a remote machine and the link to that machine is no longer active.

EOVERFLOW

The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

`stat()` and `lstat()` may fail if :

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

EOVERFLOW

A component is too large to store in the structure pointed to by *buf*.

`fstat()` fails if one or more of the following apply:

EBADF

fildev is not a valid open file descriptor.

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

ENOLINK

fildev points to a remote machine and the link to that machine is no longer active.

EOVERFLOW

The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

The `fstat()` function may fail if:

EOVERFLOW

A component is too large to store in the structure pointed to by *buf*.

SEE ALSO

`chmod(2)`, `chown(2)`, `creat(2)`, `link(2)`, `mknod(2)`, `pipe(2)`, `read(2)`, `time(2)`, `unlink(2)`, `utime(2)`, `write(2)`, `fattach(3C)`, `lfs(5)`, `stat(5)`, `types(5)`.

NAME

statvfs, fstatvfs, statvfs64, fstatvfs64 - get filesystem information

SYNOPSIS

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

int statvfs(const char *path, struct statvfs *buf);
int fstatvfs(int fildes, struct statvfs *buf);
int statvfs64(const char *path, struct statvfs64 *buf);
int fstatvfs64(int fildes, struct statvfs64 *buf);
```

DESCRIPTION

statvfs() returns a "generic superblock" describing a file system; it can be used to acquire information about mounted file systems. *buf* is a pointer to a structure (described below) that is filled by the system call.

path should name a file that resides on that file system. The file system type is known to the operating system. Read, write, or execute permission for the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

There is no functional difference between statvfs()/fstatvfs() and statvfs64()/fstatvfs64(), except for the interpretation of statvfs/statvfs64 [see [statvfs\(5\)](#)].

The statvfs structure pointed to by *buf* includes the following members:

```
ulong_t    f_bsize;           /* optimal I/O size for read and write */
ulong_t    f_frsize;          /* fundamental file system block size */
fsblkcnt_t f_blocks;          /* total # of blocks on file system
                               in units of f_frsize */
fsblkcnt_t f_bfree;           /* total # of free blocks
                               in units of f_frsize */
fsblkcnt_t f_bavail;          /* # of free blocks avail to non-superuser
                               in units of f_frsize */
fsfilcnt_t f_files;           /* total # of file nodes (inodes) */
fsfilcnt_t f_ffree;           /* total # of free file nodes */
fsfilcnt_t f_favail;          /* # of inodes avail to non-superuser */
ulong_t    f_fsid;            /* file system id (dev for now) */
char        f_basetype[FSTYPESZ]; /* target fs type name, null-terminated */
ulong_t    f_flag;            /* bit mask of flags */
ulong_t    f_namemax;         /* maximum filename length */
char        f_fstr[32];       /* file system specific string */
ulong_t    f_filler[16];      /* reserved for future expansion */
```

f_basetype contains a null-terminated FSType name of the mounted target (e.g. ufs mounted over nfs will contain ufs).

The following flags can be returned in the f_flag field:

```
ST_RDONLY    0x01    /* read-only file system */
ST_NOSUID    0x02    /* does not support setuid/setgid semantics */
ST_NOTRUNC   0x04    /* does not truncate filenames longer than
                       NAME_MAX */
```

fstatvfs() is similar to statvfs(), except that the file named by *path* in statvfs() is instead identified by an open file descriptor *fildes* obtained from a successful open(), creat(), dup(), fcntl(), or pipe() system call.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`statvfs()` and `fstatvfs()` fail if one or more of the following apply:

EINTR

A signal was caught during execution of the function.

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

EOVERFLOW

One of the values to be returned cannot be represented correctly in the structure pointed to by *buf*.

`statvfs()` fails if one or more of the following apply:

EACCES

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

EFAULT

path or *buf* points outside the process' allocated address space.

ELOOP

Too many symbolic links were encountered in translating *path*.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and the file system does not allow it.

ENAMETOOLONG

The length of a *path* component exceeds `NAME_MAX` characters, or the length of *path* exceeds `PATH_MAX` characters.

ENOENT

Either a component of the path prefix or the file referred to by *path* does not exist.

ENOLINK

path points to a remote machine and the link to that machine is no longer active.

ENOTDIR

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

`fstatvfs()` fails if one or more of the following apply:

EFAULT

buf points to an invalid address.

EBADF

fdes is not an open file descriptor.

`statvfs()` may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

RESULT

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

NOTES

`fstatvfs()` and `statvfs()` functions will fail if the total, or free, or available number of blocks or files cannot be represented correctly in the structure to be returned (`f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`).

SEE ALSO

[chmod\(2\)](#), [chown\(2\)](#), [creat\(2\)](#), [link\(2\)](#), [mknod\(2\)](#), [pipe\(2\)](#), [read\(2\)](#), [time\(2\)](#), [unlink\(2\)](#), [utime\(2\)](#), [write\(2\)](#), [lfs\(5\)](#), [statvfs\(5\)](#), [types\(5\)](#).

NAME

stime - set time

SYNOPSIS

```
#include <unistd.h>
int stime(const time_t *tp);
```

DESCRIPTION

`stime()` sets the system's idea of the time and date. *tp* points to the value of time as measured in seconds from 00:00:00 UTC January 1, 1970.

ERRORS

`stime()` will fail if:

`EPERM`

The effective user ID of the calling process is not superuser.

RETURN VALUE

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

SEE ALSO

`time(2)`, `unistd(4)`.

swapctl(2)**NAME**

swapctl - manage swap space

SYNOPSIS

```
#include <sys/stat.h>
#include <sys/swap.h>
int swapctl(int cmd, void *arg);
```

DESCRIPTION

swapctl() adds, deletes, or returns information about swap resources. *cmd* specifies one of the following options contained in *sys/swap.h*:

```
SC_ADD          /* add a resource for swapping */
SC_LIST         /* list the resources for swapping */
SC_REMOVE       /* remove a resource for swapping */
SC_GETNSWP      /* return number of swap resources */
```

When *SC_ADD* or *SC_REMOVE* is specified, *arg* is a pointer to a *swapres* structure containing the following members:

```
char    *sr_name;      /* pathname of resource */
off_t    sr_start;     /* offset to start of swap area */
off_t    sr_length;    /* length of swap area */
```

sr_start and *sr_length* are specified in 512-byte blocks.

When *SC_LIST* is specified, *arg* is a pointer to a *swaptable* structure containing the following members:

```
int      swt_n;         /* number of swapents following */
struct   swapent swt_ent[]; /* array of swt_n swapents */
```

A *swapent* structure contains the following members:

```
char    *ste_path;     /* name of the swap file */
off_t    ste_start;    /* starting block for swapping */
off_t    ste_length;   /* length of swap area */
long     ste_pages;    /* number of pages for swapping */
long     ste_free;     /* number of ste_pages free */
long     ste_flags;    /* ST_INDEL bit set if swap file */
                        /* is now being deleted */
```

SC_LIST causes swapctl() to return at most *swt_n* entries. The result of swapctl() is the number actually returned. The *ST_INDEL* bit is turned on in *ste_flags* if the swap file is in the process of being deleted.

When *SC_GETNSWP* is specified, swapctl() returns as its value the number of swap resources in use. *arg* is ignored for this operation.

The *SC_ADD* and *SC_REMOVE* functions will fail if the calling process does not have appropriate privileges.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Under the following conditions, the function swapctl() fails and sets *errno* to:

EEXIST

Part of the range specified by *sr_start* and *sr_length* is already being used for swapping on the specified resource (*SC_ADD*).

EFAULT

arg, *sr_name*, or *ste_path* points outside the allocated address space for the process.

EINVAL

The specified function value is not valid, the path specified is not a swap resource (`SC_REMOVE`), part of the range specified by `sr_start` and `sr_length` lies outside the resource specified (`SC_ADD`), or the specified swap area is less than one page (`SC_ADD`).

EISDIR

The path specified for `SC_ADD` is a directory.

ELOOP

Too many symbolic links were encountered in translating the pathname provided to `SC_ADD` or `SC_REMOVE`.

ENAMETOOLONG

The length of a component of the path specified for `SC_ADD` or `SC_REMOVE` exceeds `NAME_MAX` characters or the length of the path exceeds `PATH_MAX` characters.

ENOENT

The pathname specified for `SC_ADD` or `SC_REMOVE` does not exist.

ENOMEM

An insufficient number of `struct swapent` structures were provided to `SC_LIST`, or there were insufficient system storage resources available during an `SC_ADD` or `SC_REMOVE`, or the system would not have enough swap space after an `SC_REMOVE`.

ENOSYS

The pathname specified for `SC_ADD` or `SC_REMOVE` is not a file or block special file.

ENOTDIR

Pathname provided to `SC_ADD` or `SC_REMOVE` contained a component in the *path* prefix that was not a directory.

EPERM

The process does not have appropriate privileges.

EROFS

The pathname specified for `SC_ADD` is a read-only file system.

RETURN VALUE

Upon successful completion, the function `swapctl()` returns a value of 0 for `SC_ADD` or `SC_REMOVE`, the number of `struct swapent` entries actually returned for `SC_LIST`, or the number of swap resources in use for `SC_GETNSWP`. Upon failure, the function `swapctl()` returns a value of -1 and sets `errno` to indicate an error.

SEE ALSO

[stat\(5\)](#).

NAME

`symlink` - make a symbolic link to a file

SYNOPSIS

```
#include <unistd.h>

int symlink(const char *name1, const char *name2);
```

DESCRIPTION

`symlink()` creates a symbolic link *name2* to the file *name1*. Either name may be an arbitrary pathname, the files need not be on the same file system, and *name1* may be nonexistent.

The file to which the symbolic link points is used when an `open(2)` operation is performed on the link. A `stat(2)` on a symbolic link returns the linked-to file, while an `lstat(2)` returns information about the link itself. This can lead to surprising results when a symbolic link is made to a directory. To avoid confusion in programs, the `readlink(2)` call can be used to read the contents of a symbolic link.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

The symbolic link is made unless one or more of the following apply:

EACCES

Write permission is denied in the directory where the symbolic link is being created, or search permission is denied for a component of the path prefix of *name2*.

EDQUOT

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

EDQUOT

The new symbolic link cannot be created because the user's quota of disk blocks on the file system which will contain the link has been exhausted.

EDQUOT

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

EEXIST

The file referred to by *name2* already exists.

EFAULT

name1 or *name2* points outside the allocated address space for the process.

EIO

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

ELOOP

Too many symbolic links are encountered in translating *name2*.

ENAMETOOLONG

The length of the *name1* or *name2* argument exceeds `PATH_MAX`, or the length of a *name1* or *name2* component exceeds `NAME_MAX`.

ENOENT

A component of the path prefix of *name2* does not exist.

ENOSPC

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

ENOSPC

The new symbolic link cannot be created because no space is left on the file system which will contain

the link.

ENOSPC

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

ENOSYS

The file system does not support symbolic links.

ENOTDIR

A component of the path prefix of *name2* is not a directory.

EROFS

The file *name2* would reside on a read-only file system.

The `symlink()` function may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

RESULT

Upon successful completion `symlink()` returns a value of 0; otherwise, it returns `-1` and places an error code in `errno`.

APPLICATION USAGE

Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link provides no such assurance; in fact, the file named by the *name1* argument need not exist when the link is created. A symbolic link can cross file system boundaries.

Normal permission checks are made on each component of the symbolic link pathname during its resolution.

SEE ALSO

`cp(1)`, `lchown(2)`, `link(2)`, `lstat(2)`, `open(2)`, `readlink(2)`, `unlink(2)`, `unistd(4)`.

NAME

`sync` - update super block

SYNOPSIS

```
#include <unistd.h>
void sync(void);
```

DESCRIPTION

`sync()` causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

`sync()` should be used by programs that examine a file system, such as `fsck(1M)` or `df(1)`. `sync()` is mandatory before a re-boot.

The writing, although scheduled, is not necessarily completed before `sync()` returns. The `fsync()` system call completes the writing before it returns.

SEE ALSO

`fsync(2)`, `unistd(4)`.

NAME

`sysfs` - get file system type information

SYNOPSIS

```
#include <sys/fstyp.h>
#include <sys/fsid.h>
int sysfs(int opcode, const char *fsname);
int sysfs(int opcode, int fs_index, char *buf);
int sysfs(int opcode);
```

DESCRIPTION

`sysfs()` returns information about the file system types configured in the system. The number of arguments accepted by `sysfs()` varies and depends on the *opcode*. The currently recognized *opcodes* and their functions are:

GETFSIND

Translates *fsname*, a null-terminated file-system type identifier, into a file- system type index.

GETFSTYP

Translates *fs_index*, a file-system type index, into a null-terminated file-system type identifier and writes it into the buffer pointed to by *buf*; this buffer must be at least of size `FSTYPSZ` as defined in `sys/fstyp.h`.

GETNFSSTYP

Returns the total number of file system types configured in the system.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`sysfs()` fails if one or more of the following apply:

EINVAL

fsname points to an invalid file-system identifier; *fs_index* is zero, or invalid; *opcode* is invalid.

EFAULT

buf or *fsname* points outside the allocated address space for the process.

RETURN VALUE

Upon successful completion, `sysfs()` returns the file-system type index if the *opcode* is `GETFSIND`, a value of 0 if the *opcode* is `GETFSTYP`, or the number of file system types configured if the *opcode* is `GETNFSSTYP`. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

NAME

`sysinfo` - get and set system information

SYNOPSIS

```
#include <sys/systeminfo.h>
long sysinfo (int command, char *buf, long count);
```

DESCRIPTION

`sysinfo()` copies information relating to the Reliant UNIX system on which the process is executing into the buffer pointed to by *buf*; `sysinfo()` can also set certain information where appropriate *commands* are available. *count* is the size of the buffer.

The commands arguments which can be used are:

SI_SYSNAME

Copies into the array pointed to by *buf* the string that would be returned by `uname` [see [uname\(2\)](#)] in the `sysname` field. This is the name of the implementation of the operating system, e.g., System V or UTS.

SI_HOSTNAME

Copies into the array pointed to by *buf* a string that names the present host machine. This is the string that would be returned by `uname` [see [uname\(2\)](#)] in the `nodename` field. This host name or `nodename` is often the name the machine is known by locally.

The host name is the name of this machine as a node in some network; different networks may have different names for the node, but presenting the `nodename` to the appropriate network directory or name-to-address mapping service should produce a transport end point address. The name may not be fully qualified.

Internet host names may be up to 256 bytes in length (plus the terminating null).

SI_SET_HOSTNAME

Copies the null-terminated contents of the array pointed to by *buf* into the string maintained by the kernel whose value will be returned by succeeding calls to `sysinfo()` with the command `SI_HOSTNAME`. This command requires that the effective-user-id be superuser.

SI_RELEASE

Copies into the array pointed to by *buf* the string that would be returned by `uname` [see [uname\(2\)](#)] in the `release` field. Typical values might be 4.0 or 3.2.

SI_VERSION

Copies into the array pointed to by *buf* the string that would be returned by `uname` [see [uname\(2\)](#)] in the `version` field. The syntax and semantics of this string are defined by the system provider.

SI_MACHINE

Copies into the array pointed to by *buf* the string that would be returned by `uname` [see [uname\(2\)](#)] in the `machine` field (e.g. 3b2 or 580).

SI_ARCHITECTURE

Copies into the array pointed to by *buf* a string describing the instruction set architecture of the current system, e.g., mc68030, m32100, or i80486. These names may not match predefined names in the C language compilation system.

SI_HW_PROVIDER

Copies the name of the hardware manufacturer into the array pointed to by *buf*.

SI_HW_SERIAL

Copies into the array pointed to by *buf* a string which is the ASCII representation of the hardware-specific serial number of the physical machine on which the system call is executed. Note that this may be implemented in Read-Only Memory, via software constants set when building the

operating system, or by other means, and may contain non-numeric characters. It is anticipated that manufacturers will not issue the same "serial number" to more than one physical machine.

`SI_SRPC_DOMAIN`

Copies the Secure Remote Procedure Call domain name into the array pointed to by *buf*.

`SI_SET_SRPC_DOMAIN`

Sets the string to be returned by `sysinfo()` with the `SI_SRPC_DOMAIN` command to the *buf* array. This command requires that the effective user id be superuser.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`sysinfo()` will fail if one or both of the following apply:

`EPERM`

The process does not have appropriate privilege for a SET commands.

`EINVAL`

buf does not point to a valid address, or the data for a SET command exceeds the limits established by the implementation.

There is in many cases no corresponding programmatic interface to set these values; such strings are typically settable only by the superuser modifying entries in the `master.d` directory or the code provided by the particular OEM reading a serial number or code out of read-only memory, or hard-coded in the version of the operating system.

A good starting guess for *count* is 257, which is likely to cover all strings returned by this interface in typical installations.

RETURN VALUE

Upon successful completion, the value returned indicates the buffer size in bytes required to hold the complete value and the terminating null character. If this value is no greater than the value passed in *count*, the entire string was copied; if this value is greater than *count*, the string copied into *buf* has been truncated to *count*-1 bytes plus a terminating null character.

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

SEE ALSO

`uname(2)`, `sysconf(3C)`.

time(2) - write(2)

Programmer's Reference Manual

time(2)

NAME

`time` - get time

SYNOPSIS

```
#include <sys/types.h>
#include <time.h>
time_t time(time_t *tloc);
```

DESCRIPTION

`time()` returns the value of time in seconds since 00:00:00 UTC, January 1, 1970.

If *tloc* is non-zero, the result is also stored in the location to which *tloc* points.

RETURN VALUE

Upon successful completion, `time` returns the value of `time`. Otherwise, a value of `(time_t)-1` is returned and `errno` is set to indicate the error.

NOTES

`time()` fails and its actions are undefined if `tloc` points to an illegal address.

SEE ALSO

`stime(2)`, `ctime(3C)`, `time(5)`, `types(5)`.

NAME

`times` - get process and child process times

SYNOPSIS

```
#include <sys/types.h>
#include <sys/times.h>
clock_t times(struct tms *buffer);
```

DESCRIPTION

`times()` fills the `tms` structure pointed to by `buffer` with time-accounting information. The `tms` structure is defined in `sys/times.h` as follows:

```
struct    tms {
            clock_t      tms_utime;
            clock_t      tms_stime;
            clock_t      tms_cutime;
            clock_t      tms_cstime;
        };
```

This information comes from the calling process and each of its terminated child processes for which it has executed a `wait()` routine. All times are reported in clock ticks per second. Clock ticks are a system-dependent parameter.

`tms_utime` is the CPU time used while executing instructions in the user space of the calling process.

`tms_stime` is the CPU time used by the system on behalf of the calling process.

`tms_cutime` is the sum of the `tms_utime` and the `tms_cutime` of the child processes.

`tms_cstime` is the sum of the `tms_stime` and the `tms_cstime` of the child processes.

ERRORS

`times()` fails if:

`EFAULT`

`buffer` points to an illegal address.

RETURN VALUE

Upon successful completion, `times()` returns the elapsed real time, in clock ticks per second, from an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of `times()` to another. If `times()` fails, a `-1` is returned and `errno` is set to indicate the error.

SEE ALSO

`time(1)`, `timex(1)`, `exec(2)`, `fork(2)`, `time(2)`, `wait(2)`, `waitid(2)`, `times(5)`, `types(5)`.

NAME

uadmin - administrative functions

SYNOPSIS

```
#include <sys/uadmin.h>
int uadmin(int cmd, int fcn, void *mdep);
```

DESCRIPTION

`uadmin()` provides control for basic administrative functions. This system call is tightly coupled to the system administrative procedures and is not intended for general use. The argument *mdep* is provided for machine-dependent use and is not defined here.

The following commands are available, and can be specified for *cmd*:

A_SHUTDOWN

The system is shut down. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.

AD_HALT

Halt the processor and turn off the power.

AD_BOOT

Reboot the system, using `/stand/unix`.

AD_IBOOT

Interactive reboot; user is prompted for bootable program name.

A_REBOOT

The system stops immediately without any further processing. The action to be taken next is specified by *fcn* as above.

A_REMOUNT

The root file system is mounted again after having been fixed. This should be used only during the startup process.

ERRORS

The following error code description is function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`uadmin()` fails if any of the following apply:

EPERM

The effective user ID is not that of superuser.

RESULT

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

A_SHUTDOWN

Never returns

A_REBOOT

Never returns

A_REMOUNT

0

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

NAME

`ulimit` - get and set process limits

SYNOPSIS

```
#include <ulimit.h>
long int ulimit(int cmd, ... /* newlimit */ );
```

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

UL_GETFSIZE

Return the soft file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read. The return value is the integer part of the soft file size limit divided by 512. If the result cannot be represented as a `long int` the result is unspecified.

UL_SETFSIZE

Set the hard and soft file size limits for output operations of the process to the value of the second argument, taken as a `long int`. Any process may decrease its own hard limit, but only a process with appropriate privileges may increase the limit. The new file size limit is returned. The hard and soft file size limits are set to the specified value multiplied by 512. If the result would overflow an `rlim_t` the actual value set is unspecified.

UL_GMEMLIM

Gets the maximum possible break value [see `brk(2)`].

UL_GDESLIM

Gets the current value of the maximum number of open files per process configured in the system.

The `ulimit()` function will not change the setting of `errno` if successful.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

The `ulimit()` system call fails if either of the following applies:

EINVAL

The *cmd* argument is not valid.

EPERM

A process with an effective user ID other than superuser attempts to increase its file size limit.

NOTES

`ulimit()` is effective in limiting the growth of regular files. Pipes are currently limited to `PIPE_MAX`.

The `ulimit()` function will return "unlimited" if the current soft file size limit cannot be represented correctly in an object of type `long`. As this function is already obsolescent, the use of `getrlimit()` and `setrlimit()` is recommended for getting and setting process limits.

SEE ALSO

`brk(2)`, `getrlimit(2)`, `setrlimit(2)`, `write(2)`, `lfs(5)`, `ulimit(5)`.

NAME

`umask` - set and get file mode creation mask

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

DESCRIPTION

`umask()` sets the process' file mode creation mask to *cmask* and returns the previous value of the mask. Only the access permission bits of *cmask* and the file mode creation mask are used.

The process' file mode creation mask is used during `open()`, `creat()`, `mkdir()` and `mkfifo()` functions to turn off permission bits in the mode argument. Bit positions that are set in *cmask* are cleared in the mode of the created file.

RETURN VALUE

The file permission bits in the value returned by `umask()` will be the previous value of the file creation mask. The state of any other bits in that mask is unspecified. A subsequent call to `umask()` with the return value of the first call as argument will leave the state of the mask the same as its state before the first call, including any unspecified use of bits.

SEE ALSO

[mkdir\(1\)](#), [sh\(1\)](#), [chmod\(2\)](#), [creat\(2\)](#), [mknod\(2\)](#), [open\(2\)](#), [stat\(5\)](#), [types\(5\)](#).

NAME

umount - unmount a file system

SYNOPSIS

```
#include <sys/mount.h>
int umount(const char *file);
```

DESCRIPTION

umount() requests that a previously mounted file system contained on the block special file or directory identified by *file* be unmounted. *file* is a pointer to a *path* name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

umount() may be invoked only by the superuser.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

umount() will fail if one or more of the following apply:

EPERM

The process' effective user ID is not superuser.

EINVAL

file does not exist.

ELOOP

Too many symbolic links were encountered in translating the path pointed to by *file*.

ENAMETOOLONG

The length of the *file* argument exceeds `PATH_MAX` or the length of a *file* component exceeds `NAME_MAX`.

ENOTBLK

file is not a block special file.

EINVAL

file is not mounted.

EBUSY

A file on *file* is busy.

EFAULT

file points to an illegal address.

EREMOTE

file is a remote pathname.

ENOLINK

file is on a remote machine, and the link to that machine is no longer active.

EMULTIHOP

Components of the path pointed to by *file* require hopping to multiple remote machines.

RETURN VALUE

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

SEE ALSO

[mount\(2\)](#).

NAME

`uname` - get name of current Reliant UNIX system

SYNOPSIS

```
#include <sys/utsname.h>
int uname(struct utsname *name);
```

DESCRIPTION

`uname()` stores information identifying the current Reliant UNIX system in the structure pointed to by *name*.

`uname()` uses the structure `utsname` defined in `sys/utsname.h` whose members are:

```
char sysname[SYS_NMLN];
char nodename[SYS_NMLN];
char release[SYS_NMLN];
char version[SYS_NMLN];
char machine[SYS_NMLN];
```

`uname()` returns a null-terminated character string naming the current Reliant UNIX system in the character array `sysname`. Similarly, `nodename` contains the name that the system is known by on a communications network. `release` and `version` further identify the operating system. `machine` contains a standard name that identifies the hardware that the Reliant UNIX system is running on.

ERRORS

`EFAULT`

`uname()` fails if *name* points to an invalid address.

RETURN VALUE

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

NOTES

`uname` can now be used in conjunction with four environment variables, `OLD_SYSNAME`, `OLD_RELEASE`, `OLD_MACHINE`, `OLD_VERSION`, to change the default values returned for the system, the release, the machine and the version names, respectively.

The user who wishes to maintain backwards compatibility with the old releases, needs to set the desired environment variable to the old name, and `uname` will return the value set instead of the default.

SEE ALSO

[uname\(1\)](#), [utsname\(5\)](#).

NAME

unlink - remove directory entry

SYNOPSIS

```
#include <unistd.h>
int unlink(const char *path);
```

DESCRIPTION

The `unlink()` function removes a link to a file. If *path* names a symbolic link, `unlink()` removes the symbolic link named by *path* and does not affect any file or directory named by the contents of the symbolic link. Otherwise, `unlink()` removes the link named by the pathname pointed to by *path* and decrements the link count of the file referenced by the link.

When the file's link count becomes 0 and no process has the file open, the space occupied by the file will be freed and the file will no longer be accessible. If one or more processes have the file open when the last link is removed, the link will be removed before `unlink()` returns, but the removal of the file contents will be postponed until all references to the file are closed.

The *path* argument must not name a directory unless the process has appropriate privileges and the implementation supports using `unlink()` on directories.

Upon successful completion, `unlink()` will mark for update the `st_ctime` and `st_mtime` fields of the parent directory. Also, if the file's link count is not 0, the `st_ctime` field of the file will be marked for update.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The named file is unlinked unless one or more of the following apply:

EACCES

Search permission is denied for a component of the *path* prefix, or write permission is denied on the directory containing the link to be removed.

EBUSY

The file named by the *path* argument cannot be unlinked because it is being used by the system or another process and the implementation considers this an error.

EFAULT

path points outside the process' allocated address space.

EINTR

A signal was caught during the `unlink()` system call.

ELOOP

Too many symbolic links were encountered in translating *path*.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and the file system does not allow it.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a *path* component exceeds `NAME_MAX`.

ENOENT

The named file does not exist or is a null pathname. The user is not a superuser.

ENOLINK

path points to a remote machine and the link to that machine is no longer active.

ENOTDIR

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

EPERM

The file named by *path* is a directory, and either the calling process does not have appropriate privileges, or the implementation prohibits using `unlink()` on directories.

EPERM or EACCES

The `S_ISVTX` flag is set on the directory containing the file referred to by the *path* argument and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.

EROFS

The directory entry to be unlinked is part of a read-only file system.

The `unlink()` function may fail and not unlink the file if:

EBUSY

The file named by *path* is a named stream.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

ETXTBSY

The entry to be unlinked is the last link to a pure procedure file that is being executed.

APPLICATION USAGE

Applications should use `rmdir()` to remove a directory.

SEE ALSO

`rm(1)`, `close(2)`, `link(2)`, `open(2)`, `rmdir(2)`, `remove(3C)`, `unistd(4)`.

NAME

ustat - get file system statistics

SYNOPSIS

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

int ustat(dev_t dev, struct ustat *buf);
```

DESCRIPTION

ustat() returns information about a mounted file system. *dev* is a device number identifying a device containing a mounted file system. *buf* is a pointer to a ustat() structure that includes the following elements:

```
daddr_t    f_tfree;           /* Total free blocks */
ino_t      f_tinode;          /* Number of free inodes */
char       f_fname[6];        /* Filsys name */
char       f_fpack[6];        /* Filsys pack name */
```

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

ustat() fails if one or more of the following apply:

EINVAL

dev is not the device number of a device containing a mounted file system.

EFAULT

buf points outside the process' allocated address space.

EINTR

A signal was caught during a ustat() system call.

ENOLINK

dev is on a remote machine and the link to that machine is no longer active.

ECOMM

dev is on a remote machine and the link to that machine is no longer active.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and [errno](#) is set to indicate the error.

NOTES

ustat() will be phased out in favor of the [statvfs\(\)](#) function.

SEE ALSO

[stat\(2\)](#), [statvfs\(2\)](#), [fs\(4\)](#), [types\(5\)](#).

NAME

`utime` - set file access and modification times

SYNOPSIS

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

int utime(const char *path, const struct utimbuf *times);
```

DESCRIPTION

path points to a pathname naming a file. `utime()` sets the access and modification times of the named file.

If *times* is `NULL`, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use `utime()` in this manner.

If *times* is not `NULL`, *times* is interpreted as a pointer to a `utimbuf` structure (defined in `utime.h`) and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the superuser may use `utime` this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct utimbuf {
    time_t  actime;           /* access time */
    time_t  modtime;         /* modification time */
};
```

Upon successful completion, `utime()` will mark the time of the last file status change, `st_ctime`, to be updated, see `<sys/stat.h>`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`utime()` will fail if one or more of the following apply:

EACCES

Search permission is denied by a component of the path prefix or the *times* argument is a null pointer

EACCES

The effective user ID is not superuser and not the owner of the file and *times* is `NULL` and write access is denied.

EFAULT

times is not `NULL` and points outside the process' allocated address space.

EFAULT

path points outside the process' allocated address space.

EINTR

A signal was caught during the `utime()` system call.

ELOOP

Too many symbolic links were encountered in translating *path*.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and the file system does not allow it.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX` or the length of a path component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

ENOENT

The named file does not exist or is a null pathname.

ENOLINK

path points to a remote machine and the link to that machine is no longer active.

ENOTDIR

A component of the path prefix is not a directory.

EPERM

The *times* argument is not a null pointer and the calling process' effective user ID has write access to the file but does not match the owner of the file and the calling process does not have the appropriate privileges.

EROFS

The file system containing the file is mounted read-only.

The `utime()` function may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

RESULT

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

SEE ALSO

`stat(5)`, `types(5)`, `utime(5)`.

vfork(2)**NAME**

vfork - spawn new process in virtual memory

SYNOPSIS

```
#include <unistd.h>
pid_t vfork(void);
```

DESCRIPTION

vfork() can be used to create new processes without fully copying the address space of the old process. It is useful when the purpose of **fork()** would have been to create a new system context for an **execve()**. **vfork()** differs from **fork()** in that the child borrows the parent's memory and thread of control until a call to **execve()** or an exit (either by a call to **exit()** or abnormally.) The parent process is suspended while the child is using its resources.

vfork() returns 0 in the child's context and (later) the process ID (PID) of the child in the parent's context.

vfork() can normally be used just like **fork()**. It does not work, however, to return while running in the child's context from the procedure which called **vfork()** since the eventual return from **vfork()** would then return to a no longer existent stack frame. Be careful, also, to call **_exit()** rather than **exit()** if you cannot **execve()**, since **exit()** will flush and close standard I/O channels, and thereby mess up the parent processes standard I/O data structures. Even with **fork()** it is wrong to call **exit()** since buffered data would then be flushed twice.

RESULT

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

vfork() will fail and no child process will be created if one or more of the following apply:

EAGAIN

The system-imposed limit on the total number of processes under execution would be exceeded. This limit is determined when the system is generated.

EAGAIN

The system-imposed limit on the total number of processes under execution by a single user would be exceeded. This limit is determined when the system is generated.

ENOMEM

There is insufficient swap space for the new process.

APPLICATION USAGE

On some systems, **vfork()** is the same as **fork()**.

The **vfork()** function differs from **fork()** only in that the child process can share code and data with the calling process (parent process). This speeds cloning activity significantly at a risk to the integrity of the parent process if **vfork()** is misused.

The use of **vfork()** for any purpose except as a prelude to an immediate call to a function from the **exec** family, or to **_exit()**, is not advised.

The **vfork()** function can be used to create new processes without fully copying the address space of the

old process. If a forked process is simply going to call `exec` the data space copied from the parent to the child by `fork()` is not used. This is particularly inefficient in a paged environment, making `vmfork()` particularly useful. Depending upon the size of the parent's data space, `vmfork()` can give a significant performance improvement over `fork()`.

If signal handlers are invoked in the child process after `vmfork()`, they must follow the same rules as other code in the child process.

The `[vmfork, exec]` window begins at the `vmfork()` call and ends when the child completes its `exec` call.

SEE ALSO

`exec(2)`, `exit(2)`, `fork(2)`, `ioctl(2)`, `wait(2)`, `unistd(4)`.

NAME

`wait`, `waitpid` - wait for child process to stop or terminate

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

The `wait()` and `waitpid()` functions allow the calling process to obtain status information pertaining to one of its child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

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()` or `waitpid()` 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.

The `waitpid()` function will behave identically to `wait()`, if the `pid` argument is `(pid_t)-1` and the `options` argument is 0. Otherwise, its behavior will be modified by the values of the `pid` and `options` arguments.

The `pid` argument specifies a set of child processes for which status is requested. The `waitpid()` function will only return the status of a child process from this set:

- If `pid` is equal to `(pid_t)-1`, status is requested for any child process. In this respect, `waitpid()` is then equivalent to `wait()`.
- If `pid` is greater than 0, it specifies the process ID of a single child process for which status is requested.
- If `pid` is 0, status is requested for any child process whose process group ID is equal to that of the calling process.
- 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`.

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>` [see [wait\(5\)](#)].

WCONTINUED

The `waitpid()` function will report the status of any continued child process specified by `pid` whose status has not been reported since it continued from a job control stop.

WNOHANG

The `waitpid()` function will not suspend execution of the calling thread if status is not immediately available for one of the child processes specified by `pid`.

WUNTRACED

The status of any child processes specified by `pid` that are stopped, and whose status has not yet been reported since they stopped, will also be reported to the requesting process.

If the calling process has `SA_NOCLDWAIT` set or has `SIGCHLD` set to `SIG_IGN`, and the process has no unwaited for children that were transformed into zombie processes, the calling thread will block until all of the children of the process containing the calling thread terminate, and `wait()` and `waitpid()` will fail and set `errno` to `ECHILD`.

If `wait()` or `waitpid()` return because the status of a child process is available, these functions will return a value equal to the process ID of the child process. In this case, if the value of the argument `stat_loc` is not a null pointer, information will be stored in the location pointed to by `stat_loc`. If and only if the status returned is from a terminated child process that returned 0 from `main()` or passed 0 as the `status` argument to `_exit()` or `exit()`, the value stored at the location pointed to by `stat_loc` will be 0. Regardless of its value, this information may be interpreted using the following macros, which are defined in `<sys/wait.h>` and evaluate to integral expressions; the `stat_val` argument is the integer value pointed to by `stat_loc`.

`WIFEXITED(stat_val)`

Evaluates to a non-zero value if status was returned for a child process that terminated normally.

`WEXITSTATUS(stat_val)`

If the value of `WIFEXITED(stat_val)` is non-zero, this macro evaluates to the low-order 8 bits of the `status` argument that the child process passed to `_exit()` or `exit()`, or the value the child process returned from `main()`.

`WIFSIGNALED(stat_val)`

Evaluates to non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught [see [signal\(5\)](#)].

`WTERMSIG(stat_val)`

If the value of `WIFSIGNALED(stat_val)` is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

`WIFSTOPPED(stat_val)`

Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

`WSTOPSIG(stat_val)`

If the value of `WIFSTOPPED(stat_val)` is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

`WIFCONTINUED(stat_val)`

Evaluates to a non-zero value if status was returned for a child process that has continued from a job control stop.

If the information pointed to by `stat_loc` was stored by a call to `waitpid()` that specified the `WUNTRACED` flag and did not specify the `WCONTINUED` flag, exactly one of the macros `WIFEXITED(*stat_loc)`, `WIFSIGNALED(*stat_loc)`, and `WIFSTOPPED(*stat_loc)`, will evaluate to a non-zero value.

If the information pointed to by `stat_loc` was stored by a call to `waitpid()` that specified the `WUNTRACED` and `WCONTINUED` flags, exactly one of the macros `WIFEXITED(*stat_loc)`, `WIFSIGNALED(*stat_loc)`, `WIFSTOPPED(*stat_loc)`, and `WIFCONTINUED(*stat_loc)`, will evaluate to a non-zero value.

If the information pointed to by `stat_loc` was stored by a call to `waitpid()` that did not specify the `WUNTRACED` or `WCONTINUED` flags, or by a call to the `wait()` function, exactly one of the macros `WIFEXITED(*stat_loc)` and `WIFSIGNALED(*stat_loc)` will evaluate to a non-zero value.

If the information pointed to by `stat_loc` was stored by a call to `waitpid()` that did not specify the `WUNTRACED` flag and specified the `WCONTINUED` flag, or by a call to the `wait()` function, exactly one of the macros `WIFEXITED(*stat_loc)`, `WIFSIGNALED(*stat_loc)`, and `WIFCONTINUED(*stat_loc)`, will evaluate to a non-zero value.

There may be additional implementation-dependent circumstances under which `wait()` or `waitpid()` report status. This will not occur unless the calling process or one of its child processes explicitly makes use of a non-standard extension. In these cases the interpretation of the reported status is implementation-dependent.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes will be assigned a new parent process ID corresponding to an implementation-dependent system process.

RETURN VALUE

If `wait()` or `waitpid()` returns because the status of a child process is available, these functions will return

a value equal to the process ID of the child process for which status is reported. If `wait()` or `waitpid()` returns due to the delivery of a signal to the calling process, `-1` will be returned and `errno` will be 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*, `0` will be returned. Otherwise, `(pid_t)-1` will be returned, and `errno` will be set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

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. The value of the location pointed to by *stat_loc* is undefined.

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.

`EINTR`

The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

`EINVAL`

The *options* argument is not valid.

SEE ALSO

[exec\(2\)](#), [exit\(2\)](#), [fork\(2\)](#), [pause\(2\)](#), [ptrace\(2\)](#), [signal\(2\)](#), [waitid\(2\)](#), [wait3\(3C\)](#), [signal\(5\)](#), [types\(5\)](#), [wait\(5\)](#), [wstat\(5\)](#).

NAME

`waitid` - wait for child process to change state

SYNOPSIS

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop,
int options);
```

DESCRIPTION

The `waitid()` function suspends the calling thread until one child of the process containing the calling thread changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call to `waitid()`, `waitid()` returns immediately. If more than one thread is suspended in `wait()` or `waitpid()` waiting termination of the same process, exactly one thread will return the process status at the time of the target process termination.

The *idtype* and *id* arguments specify which children `waitid()` is to wait for.

- If *idtype* is `P_PID`, `waitid()` waits for the child with a process ID equal to `(pid_t) id`.
- If *idtype* is `P_PGID`, `waitid()` waits for any child with a process group ID equal to `(pid_t) id`.
- If *idtype* is `P_ALL`, `waitid()` waits for any children and *id* is ignored.

The *options* argument is used to specify which state changes `waitid()` is to wait for. It is formed by a bitwise OR of any of the following flags:

`WEXITED`

Wait for process(es) to exit.

`WTRAPPED`

Wait for traced process(es) to become trapped or reach a breakpoint [see [ptrace\(2\)](#)].

`WSTOPPED`

Wait for and return the process status of any child that has stopped upon receipt of a signal.

`WCONTINUED`

Return the status for any child that was stopped and has been continued.

`WNOHANG`

Return immediately if there are no children to wait for.

`WNOWAIT`

Keep the process whose status is returned in *infop* in a waitable state. This will not affect the state of the process; the process may be waited for again after this call completes.

The *infop* argument must point to a `siginfo_t` structure, as defined in [siginfo\(5\)](#). If `waitid()` returns because a child process was found that satisfied the conditions indicated by the arguments *idtype* and *options*, then the structure pointed to by *infop* will be filled in by the system with the status of the process. The `si_signo` member will always be equal to `SIGCHLD`.

RETURN VALUE

If `waitid()` returns due to a change of state of one of its children, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`waitid()` fails if one or more of the following are true.

ECHILD

The set of processes specified by *idtype* and *id* does not contain any unwaited-for processes.

EFAULT

infop points to an invalid address.

EINTR

`waitid()` was interrupted due to the receipt of a signal by the calling process.

EINVAL

An invalid value was specified for *options*, or *idtype* and *id* specify an invalid set of processes.

SEE ALSO

`exec(2)`, `exit(2)`, `fork(2)`, `pause(2)`, `ptrace(2)`, `signal(2)`, `sigaction(2)`, `wait(2)`, `siginfo(5)`, `types(5)`, `wait(5)`.

NAME

`write`, `writew` - write into a file

SYNOPSIS

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbyte);
#include <limits.h>
#include <sys/types.h>
#include <sys/uio.h>
int writew(int fd, const struct iovec *iov, int
iovcnt);
```

DESCRIPTION

The `write()` function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fd*.

If *nbyte* is 0, `write()` will return 0 and have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with *fd*. Before successful return from `write()`, the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation.

If a `write()` requests that more bytes be written than there is room for (for example, the process limit [ulimit] or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below) and the implementation will generate a `SIGXFSZ` signal for the thread.

If `write()` is interrupted by a signal before it writes any data, it will return `-1` with `errno` set to `EINTR`.

If `write()` is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If the value of *nbyte* is greater than `SSIZE_MAX`, the result is implementation-dependent.

After a `write()` to a regular file has successfully returned:

- Any successful `read()` from each byte position in the file that was modified by that write will return the data specified by the `write()` for that position until such byte positions are again modified.
- Any subsequent successful `write()` to the same byte position in the file will overwrite that file data.

Write requests to a pipe or FIFO will be handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request will append to the end of the pipe.
- Write requests of `PIPE_BUF` bytes or less will not be interleaved with data from other processes doing writes on the same pipe. Writes of greater than `PIPE_BUF` bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the `O_NONBLOCK` flag of the file status flags is set.
- If the `O_NONBLOCK` flag is clear, a write request may cause the thread to block, but on normal completion it

will return *nbyte*.

- If the `O_NONBLOCK` flag is set, `write()` requests will be handled differently, in the following ways:
 - The `write()` function will not block the thread.
 - A write request for `PIPE_BUF` or fewer bytes will have the following effect: If there is sufficient space available in the pipe, `write()` will transfer all the data and return the number of bytes requested. Otherwise, `write()` will transfer no data and return `-1` with `errno` set to `EAGAIN`.
 - A write request for more than `PIPE_BUF` bytes will case one of the following:
 - a) When at least one byte can be written, transfer what it can and return the number of bytes written. When all data previously written to the pipe is read, it will transfer at least `PIPE_BUF` bytes.
 - b) When no data can be written, transfer no data and return `-1` with `errno` set to `EAGAIN`.

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports non-blocking writes and cannot accept the data immediately:

- If the `O_NONBLOCK` flag is clear, `write()` will block the calling thread until the data can be accepted.
- If the `O_NONBLOCK` flag is set, `write()` will not block the process. If some data can be written without blocking the process, `write()` will write what it can and return the number of bytes written. Otherwise, it will return `-1` and `errno` will be set to `EAGAIN`.

Upon successful completion, where *nbyte* is greater than 0, `write()` 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 may be cleared.

If *fildev* refers to a stream, the operation of `write()` is determined by the values of the minimum and maximum *nbyte* range ("packet size") accepted by the stream. These values are determined by the topmost stream module. If *nbyte* falls within the packet size range, *nbyte* bytes will be written. If *nbyte* does not fall within the range and the minimum packet size value is 0, `write()` will break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, `write()` will fail with `errno` set to `ERANGE`. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends 0 bytes with 0 returned. However, writing a zero-length buffer to a STREAMS-based pipe or FIFO sends no message and 0 is returned. The process may issue `I_SWROPT` `ioctl()` to enable zero-length messages to be sent across the pipe or FIFO.

When writing to a stream, data messages are created with a priority band of 0. When writing to a stream that is not a pipe or FIFO:

- If `O_NONBLOCK` is clear, and the stream cannot accept data (the stream write queue is full due to internal flow control conditions), `write()` will block until data can be accepted.
- If `O_NONBLOCK` is set and the stream cannot accept data, `write()` will return `-1` and set `errno` to `EAGAIN`.
- If `O_NONBLOCK` is set and part of the buffer has been written while a condition in which the stream cannot accept additional data occurs, `write()` will terminate and return the number of bytes written.

In addition, `write()` and `writew()` will fail if the stream head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `write()` or `writew()` but reflects the prior error.

The `writew()` function is equivalent to `write()`, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*–1]. *iovcnt* is valid if greater than 0 and less than or equal to `IOV_MAX`, defined in `<limits.h>` [see [limits\(4\)](#)].

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. The `writew()` function will always write a complete area before proceeding to the next.

If *fildev* refers to a regular file and all of the *iov_len* members in the array pointed to by *iov* are 0, `writew()` will return 0 and have no other effect. For other file types, the behaviour is unspecified.

If the sum of the `iov_len` values is greater than `SSIZE_MAX`, the operation fails and no data is transferred.

If the `O_SYNC` bit has been set, write I/O operations on the file descriptor complete as defined by synchronised I/O file integrity completion.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *fil*des.

RETURN VALUE

Upon successful completion, `write()` will return the number of bytes actually written to the file associated with *fil*des. This number will never be greater than *n*byte. Otherwise, `-1` is returned and `errno` is set to indicate the error.

Upon successful completion, `writev()` returns the number of bytes actually written. Otherwise, it returns a value of `-1`, the file-pointer remains unchanged, and `errno` is set to indicate an error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `write()` and `writev()` functions will fail if:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor and the thread would be delayed in the `write()` operation.

EBADF

The *fil*des argument is not a valid file descriptor open for writing.

EFBIG

An attempt was made to write a file that exceeds the implementation-dependent maximum file size or the process' file size limit.

EFBIG

The file is a regular file, *n*byte is greater than 0 and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fil*des.

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.

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. 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, or that only has one end open. A `SIGPIPE` signal will also be sent to the thread.

ERANGE

The transfer request size was outside the range supported by the STREAMS file associated with *fil*des.

The `writev` function will fail if:

EINVAL

The sum of the `iov_len` values in the *iov* array would overflow an `ssize_t`.

The `write()` and `writev()` functions may fail if:

EINVAL

The stream or multiplexer referenced by *fil*des is linked (directly or indirectly) downstream from a multiplexer.

ENXIO

A request was made of a non-existent device, or the request was outside the capabilities of the device.

ENXIO

A hangup occurred on the stream being written to.

A write to a STREAMS file may fail if an error message has been received at the stream head. In this case, `errno` is set to the value included in the error message.

The `writew()` function may fail and set `errno` to:

EINVAL

The `iovcnt` argument was less than or equal to 0, or greater than `IOV_MAX`.

NOTES

`write()` and `writew()` may do a "partial read or write" due to the offset maximum. That is, the value returned may be less than `nbyte` if the number of bytes remaining which may be transferred is less than `nbyte`.

SEE ALSO

`chmod(2)`, `creat(2)`, `dup(2)`, `fcntl(2)`, `getrlimit(2)`, `lseek(2)`, `open(2)`, `pipe(2)`, `ulimit(2)`, `limits(4)`, `unistd(4)`, `lfs(5)`, `stropts(5)`, `types(5)`, `uio(5)`.

Miscellaneous functions from the library libucb(3)

alloca(3) - usleep(3)

Programmer's Reference Manual

alloca(3)
BSD Compatibility Package

NAME

`alloca` - memory allocator

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <alloca.h>
char *alloca(size)
unsigned size;
```

DESCRIPTION

`alloca` allocates *size* bytes of space off the run-time stack and returns a pointer to the allocated block. This temporary space is automatically freed when blocks are found deeper than current allocations or when subsequent calls to `alloca` are made in the form `alloca(0)`.

NOTES

It should work under any C implementation that uses an actual procedure stack (as opposed to a linked list of frames). There are some preprocessor constants that can be defined when compiling for your specific system, for improved efficiency.

SEE ALSO

`csh(1)`, `ld(1)`, `brk(2)`, `getrlimit(2)`, `sigstack(3)`, `sigvec(3)`, `calloc(3X)`.

getpagesize(3)
BSD Compatibility Package**NAME**

getpagesize - get system page size

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb  
#include <unistd.h>  
int getpagesize(void);
```

DESCRIPTION

getpagesize returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The getpagesize() function is equivalent to sysconf(_SC_PAGE_SIZE) and sysconf(_SC_PAGESIZE).

The page size is a system page size and need not be the same as the underlying hardware page size.

SEE ALSO

pagesize(1), brk(2), getrlimit(2), mmap(2), mprotect(2), munmap(2), msync(3C), sysconf(3C), unistd(4).

getusershell(3)
BSD Compatibility Package**NAME**

getusershell, setusershell, endusershell - get legal user shells

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
char *getusershell();
setusershell();
endusershell();
```

DESCRIPTION

getusershell 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 locations of the standard system shells, /usr/bin/csh, /usr/bin/sh, and /usr/bin/ksh are returned.

getusershell reads the next line (opening the file if necessary); setusershell rewinds the file; endusershell closes it.

RETURN VALUE

The routine getusershell returns a NULL pointer (0) on EOF or error.

NOTES

All information is contained in a static area so it must be copied if it is to be saved.

FILES

```
/etc/shells
/usr/bin/csh
/usr/bin/ksh
/usr/bin/sh
```

NAME

mctl - memory management control

SYNOPSIS

```
/usr/ucb/cc [flag ...] file ... -lucb
#include <sys/types.h>
#include <sys/mman.h>
int mctl(caddr_t addr, size_t len, int function, void *arg);
```

DESCRIPTION

`mctl()` applies a variety of control functions over pages identified by the mappings established for the address range `[addr, addr + len)`. The function to be performed is identified by the argument `function`. Valid functions are defined in `mman.h` as follows.

MC_LOCK

Lock the pages in the range in memory. This function is used to support `mlock()`. See [mlock\(3C\)](#) for semantics and usage. `arg` is ignored.

MC_LOCKAS

Lock the pages in the address space in memory. This function is used to support `mlockall()`. See [mlockall\(3C\)](#) for semantics and usage. `addr` and `len` are ignored. `arg` is an integer built from the flags:

MCL_CURRENT

Lock current mappings.

MCL_FUTURE

Lock future mappings.

MC_SYNC

Synchronize the pages in the range with their backing storage. Optionally invalidate cache copies. This function is used to support `msync()`. See [msync\(3C\)](#) for semantics and usage. `arg` is used to represent the `flags` argument to `msync`. It is constructed from an OR of the following values:

MS_SYNC

Synchronized write.

MS_ASYNC

Return immediately.

MS_INVALIDATE

Invalidate mappings.

`MS_ASYNC` returns after all I/O operations are scheduled. `MS_SYNC` does not return until all I/O operations are complete. Specify exactly one of `MS_ASYNC` or `MS_SYNC`. `MS_INVALIDATE` invalidates all cached copies of data from memory, requiring them to be re-obtained from the object's permanent storage location upon the next reference.

MC_UNLOCK

Unlock the pages in the range. This function is used to support `munlock()`. See [mlock\(3C\)](#) for semantics and usage. `arg` is ignored.

MC_UNLOCKAS

Remove address space memory lock, and locks on all current mappings. This function is used to support `munlockall()`. `addr` and `len` must have the value 0. `arg` is ignored.

RETURN VALUE

`mctl` returns 0 on success, -1 on failure.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`mctl()` fails if:

EAGAIN

Some or all of the memory identified by the operation could not be locked due to insufficient system resources.

EBUSY

`MS_INVALIDATE` was specified and one or more of the pages is locked in memory.

EINVAL

addr is not a multiple of the page size as returned by `getpagesize()`.

EINVAL

addr and/or *len* do not have the value 0 when `MC_LOCKAS` or `MC_UNLOCKAS` are specified.

EINVAL

arg is not valid for the function specified.

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

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.

EPERM

The process' effective user ID is not super-user and one of `MC_LOCK`, `MC_LOCKAS`, `MC_UNLOCK`, or `MC_UNLOCKAS` was specified.

SEE ALSO

[mmap\(2\)](#), [getpagesize\(3\)](#), [mlock\(3C\)](#), [mlockall\(3C\)](#), [msync\(3C\)](#), [mman\(5\)](#), [types\(5\)](#).

NAME

`reboot` - reboot system or halt processor

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <sys/reboot.h>
reboot(howto, [bootargs])
int howto;
char *bootargs;
```

DESCRIPTION

`reboot` reboots the system, and is invoked automatically in the event of unrecoverable system failures. *howto* is a mask of options passed to the bootstrap program. The system call interface permits only `RB_HALT` or `RB_AUTOBOOT` to be passed to the reboot program; the other flags are used in scripts stored on the console storage media, or used in manual bootstrap procedures. When none of these options (for instance `RB_AUTOBOOT`) is given, the system is rebooted from file `/stand/unix`. An automatic consistency check of the disks is then normally performed.

The bits of *howto* that are used are:

`RB_HALT`

The processor is simply halted; no reboot takes place. `RB_HALT` should be used with caution.

`RB_ASKNAME`

Interpreted by the bootstrap program itself, causing it to inquire as to what file should be booted. Normally, the system is booted from the file `/stand/unix` without asking.

RETURN VALUE

If successful, this call never returns. Otherwise, a `-1` is returned and an error is returned in the global variable `errno`.

DIAGNOSTICS

`EPERM`

The caller is not the super-user.

NOTES

Any other *howto* argument causes `/stand/unix` to boot.

Only the super-user may `reboot` a machine.

SEE ALSO

`crash(1M)`, `halt(1M)`, `init(1M)`, `reboot(1M)`.

NAME

scandir, alphasort - scan a directory

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <sys/types.h>
#include <sys/dir.h>
scandir(dirname, &namelist, select, compar)
char *dirname;
struct direct **namelist;
int (*select)();
int (*compar)();
alphasort(d1, d2)
struct direct **d1, **d2;
```

DESCRIPTION

`scandir` reads the directory *dirname* and builds an array of pointers to directory entries using `malloc`. The second parameter is a pointer to an array of structure pointers. The third parameter is a pointer to a routine which is called with a pointer to a directory entry and should return a non-zero value if the directory entry should be included in the array. If this pointer is `NULL`, then all the directory entries will be included. The last argument is a pointer to a routine which is passed to `qsort` to sort the completed array. If this pointer is `NULL`, the array is not sorted. `alphasort` is a routine which will sort the array alphabetically.

`scandir` returns the number of entries in the array and a pointer to the array through the parameter *namelist*.

RETURN VALUE

Returns `-1` if the directory cannot be opened for reading or if `malloc` cannot allocate enough memory to hold all the data structures.

SEE ALSO

[getdents\(2\)](#), [directory\(3C\)](#), [malloc\(3C\)](#), [qsort\(3C\)](#), [types\(5\)](#).

sigblock(3)
BSD Compatibility Package**NAME**

`sigblock`, `sigmask` - block signals

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <signal.h>
sigblock(mask)
int mask;
#define sigmask(signum)
```

DESCRIPTION

`sigblock` adds the signals specified in *mask* to the set of signals currently being blocked from delivery. Signals are blocked if the appropriate bit in *mask* is a 1; the macro `sigmask` is provided to construct the mask for a given *signum*. The previous mask is returned, and may be restored using `sigsetmask`. It is not possible to block `SIGKILL`, `SIGSTOP`, or `SIGCONT`; this restriction is silently imposed by the system.

RETURN VALUE

The previous set of masked signals is returned.

SEE ALSO

`kill(2)`, `sigaction(2)`, `signal(2)`, `sigsetmask(3)`, `sigvec(3)`, `signal(3-ucb)`, `signal(5)`.

NAME

sigfpe - signal handling for specific SIGFPE codes

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <signal.h>
#include <floatingpoint.h>
sigfpe_handler_type sigfpe(code, hdl)
sigfpe_code_type code;
sigfpe_handler_type hdl;
```

DESCRIPTION

This 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, specifying core dump using `abort`, or default handling respectively.

For these IEEE-related codes:

<code>FPE_FLTINEX_TRAP</code>	<code>fp_inexact</code>	floating inexact result
<code>FPE_FLTDIV_TRAP</code>	<code>fp_division</code>	floating division by zero
<code>FPE_FLTUND_TRAP</code>	<code>fp_underflow</code>	floating underflow
<code>FPE_FLTOVF_TRAP</code>	<code>fp_overflow</code>	floating overflow
<code>FPE_FLTBSUN_TRAP</code>	<code>fp_invalid</code>	branch or set on unordered
<code>FPE_FLTOPERR_TRAP</code>	<code>fp_invalid</code>	floating operand error
<code>FPE_FLTNAN_TRAP</code>	<code>fp_invalid</code>	floating Not-A-Number

default handling is defined to be to call the handler specified to `ieee_handler(3M)`.

For all other SIGFPE codes, default handling is to core dump using `abort(3C)`.

The compilation option `-ffpa` causes FPA recomputation to replace the default abort action for code `FPE_FPA_ERROR`. Note: `SIGFPE_DEFAULT` will restore abort rather than FPA recomputation for this code.

Three steps are required to intercept an IEEE-related SIGFPE code with `sigfpe`.

- 1) Set up a handler with `sigfpe`.
- 2) Enable the relevant IEEE trapping capability in the hardware, perhaps by using assembly-language instructions.
- 3) Perform a floating-point operation that generates the intended IEEE exception.

Unlike `ieee_handler(3M)`, `sigfpe` never changes floating-point hardware mode bits affecting IEEE trapping. No IEEE-related SIGFPE signals will be generated unless those hardware mode bits are enabled.

SIGFPE signals can be handled using `sigvec(3)`, `signal(3-ucb)`, `sigfpe(3)`, or `ieee_handler(3M)`. In a particular program, to avoid confusion, use only one of these interfaces to handle SIGFPE signals.

EXAMPLES

A user-specified signal handler might look like this:

```
void sample_handler( sig, code, scp, addr ) □
```

```

int sig ;                /* sig == SIGFPE always */
int code ;
struct sigcontext *scp ;
char *addr ;
{
    /*
     * Sample user-written sigfpe code handler.
     * Prints a message and continues.
     * struct sigcontext is defined in <signal.h>.
     */
    printf(" ieee exception code %x occurred at pc %X \n",
           code, scp->sc_pc);
}

```

It might be set up like this:

```

extern void sample_handler;
main
{
    sigfpe_handler_type hdl, old_handler1, old_handler2;
    /*
     * save current overflow and invalid handlers; set the new
     * overflow handler to sample_handler and set the new
     * invalid handler to SIGFPE_ABORT (abort on invalid)
     */
    hdl = (sigfpe_handler_type) sample_handler;
    old_handler1 = sigfpe(FPE_FLTOVF_TRAP, hdl);
    old_handler2 = sigfpe(FPE_FLTOPERR_TRAP, SIGFPE_ABORT);
    ...
    /*
     * restore old overflow and invalid handlers
     */
    sigfpe(FPE_FLTOVF_TRAP, old_handler1);
    sigfpe(FPE_FLTOPERR_TRAP, old_handler2);
}

```

RETURN VALUE

`sigfpe` returns `BADSIG` if `code` is not zero or a defined `SIGFPE` code.

FILES

```

/usr/include/floatingpoint.h
/usr/include/signal.h

```

SEE ALSO

[sigvec\(3\)](#), [signal\(3-ucb\)](#), [abort\(3C\)](#), [floatingpoint\(3M\)](#), [ieee_handler\(3M\)](#), [signal\(5\)](#).

NAME

siginterrupt - allow signals to interrupt system calls

SYNOPSIS

```
/usr/ucb/cc [flag ...] file ... -lucb
#include <signal.h>
int siginterrupt(int sig, int flag);
```

DESCRIPTION

siginterrupt() is used to change the system call restart behavior when a system call is interrupted by the specified signal. If the flag is false (0), then system calls will be restarted if they are interrupted by the specified signal and no data has been transferred yet. System call restart is the default behavior when the signal() routine is used.

The siginterrupt() function is used to change the restart behavior when a function is interrupted by the specified signal. The function siginterruptP(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;
}
```

If the flag is true (1), then restarting of system calls is disabled. If a system call is interrupted by the specified signal and no data has been transferred, the system call will return -1 with errno set to EINTR. Interrupted system calls that have started transferring data will return the amount of data actually transferred.

Issuing a siginterrupt() call during the execution of a signal handler will cause the new action to take place on the next signal to be caught.

ERRORS

The siginterrupt() function will fail if:

EINVAL

The sig argument is not a valid signal number.

RETURN VALUE

A 0 value indicates that the call succeeded. A -1 value indicates that an invalid signal number has been supplied.

NOTES

The siginterrupt() function supports programs written to historical system interfaces. A portable application, when being written or rewritten, should use sigaction() with the SA_RESTART flag instead of siginterrupt().

This library routine uses an extension of the sigvec() system call that is not available in BSD 4.2, hence it should not be used if backward compatibility is needed.

SEE ALSO

```
sigaction(2), signal(2), sigblock(3), sigpause(3), sigsetmask(3), sigvec(3),  
signal(3-ucb), signal(5).
```

NAME

signal - simplified software signal facilities

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <signal.h>
void (*signal(sig, func))()
void (*func)();
```

DESCRIPTION

signal() is a simplified interface to the more general **sigvec(3)** facility. Programs that use **signal()** in preference to **sigvec()** are more likely to be portable to all systems.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (for example, bus error), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background [see **termio(7)**]. Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the **SIGKILL** and **SIGSTOP** signals, the **signal()** call allows signals either to be ignored or to interrupt to a specified location. The following is a list of all signals with names as in the include file **<signal.h>**:

SIGHUP		Hangup
SIGINT		Interrupt
SIGQUIT	*	Quit
SIGILL	*	Illegal instruction
SIGTRAP	*	Trace trap
SIGABRT	*	Abort (generated by abort(3C) routine)
SIGEMT	*	Emulator trap
SIGFPE	*	Arithmetic exception
SIGKILL		Kill (cannot be caught, blocked, or ignored)
SIGBUS	*	Bus error
SIGSEGV	*	Segmentation violation
SIGSYS	*	Bad argument to system call
SIGPIPE		Write on a pipe or other socket with no one to read it
SIGALRM		Alarm clock
SIGTERM		Software termination signal

SIGURG	*b	Urgent condition present on socket
SIGSTOP	*d	Stop (cannot be caught, blocked, or ignored)
SIGTSTP	*d	Stop signal generated from keyboard
SIGCONT	*b	Continue after stop (cannot be blocked)
SIGCHLD	*b	Child status has changed
SIGTTIN	*d	Background read attempted from control terminal
SIGTTOU	*d	Background write attempted to control terminal
SIGIO	*b	I/O is possible on a descriptor [see fcntl(2)]
SIGXCPU		CPU time limit exceeded [see setrlimit(2)]
SIGXFSZ		File size limit exceeded [see setrlimit(2)]
SIGVTALRM		Virtual time alarm [see getitimer(3C)]
SIGPROF		Profiling timer alarm [see getitimer(3C)]
SIGWINCH	*b	Window changed [see termio(7)]
SIGUSR1		User-defined signal 1
SIGUSR2		User-defined signal 2

The starred signals ("*") in the list above cause a core image if not caught or ignored.

If *func* is `SIG_DFL`, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with "*b" or "*d". Signals marked with "*b" are discarded if the action is `SIG_DFL`; signals marked with "*d" cause the process to stop. If *func* is `SIG_IGN` the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted.

If a caught signal occurs during certain system calls, terminating the call prematurely, the call is automatically restarted. In particular this can occur during a `read()` or `write()` on a slow device (such as a terminal; but not a file) and during a `wait()`.

The value of `signal()` is the previous (or initial) value of *func* for the particular signal.

After a `fork()` or `vfork()` the child inherits all signals. An `execve()` resets all caught signals to the default action; ignored signals remain ignored.

NOTES

The handler routine can be declared like this:

```
void handler(sig, code, scp, addr)
int sig, code;
struct sigcontext *scp;
char *addr;
```

In this example:

sig Signal number.

code Parameter of certain signals that provides additional detail.

scp Pointer to the `sigcontext` structure (defined in `<signal.h>`), used to restore the context from before the signal.

addr Additional address information.

RETURN VALUE

The previous action is returned on a successful call. Otherwise, `-1` is returned and `errno` is set to indicate the error.

ERRORS

`signal` fails and no action takes place.

`EINVAL`

sig is not a valid signal number, or is `SIGKILL` or `SIGSTOP`.

SEE ALSO

`kill(1)`, `execve(2)`, `fcntl(2)`, `fork(2)`, `getrlimit(2)`, `kill(2)`, `ptrace(2)`, `read(2)`, `sigaction(2)`, `wait(2)`, `write(2)`, `sigblock(3)`, `sigpause(3)`, `sigsetmask(3)`, `sigstack(3)`, `sigvec(3)`, `wait3(3)`, `getitimer(3C)`, `setjmp(3C)`, `setjmp(3C-ucb)`, `signal(5)`, `termio(7)`.

NAME

`sigpause` - automatically release blocked signals and wait for interrupt

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <signal.h>
int sigpause(int sigmask) ; □
```

DESCRIPTION

`sigpause()` assigns *sigmask* to the set of masked signals and then waits for a signal to arrive; on return the set of masked signals is restored. *sigmask* is usually 0 to indicate that no signals are now to be blocked. `sigpause()` always terminates by being interrupted, returning `EINTR`.

In normal usage, a signal is blocked using `sigblock()`, to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using `sigpause()` with the mask returned by `sigblock()`.

SEE ALSO

`signal(2)`, `sigaction(2)`, `sigblock(3)`, `sigvec(3)`, `signal(3-ucb)`, `signal(5)`.

sigsetmask(3)
BSD Compatibility Package**NAME**

sigsetmask - set current signal mask

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <signal.h>
sigsetmask(mask)
int mask;
#define sigmask(signum)
```

DESCRIPTION

sigsetmask sets the current signal mask (those signals that are blocked from delivery). Signals are blocked if the corresponding bit in *mask* is a 1. The macro **sigmask** is provided to construct the mask for a given *signum*.

The system quietly disallows SIGKILL, SIGSTOP, or SIGCONT from being blocked.

RETURN VALUE

The previous set of masked signals is returned.

SEE ALSO

[kill\(2\)](#), [signal\(2\)](#), [sigblock\(3\)](#), [sigpause\(3\)](#), [sigvec\(3\)](#), [signal\(3-ucb\)](#), [signal\(5\)](#).

sigstack(3)
BSD Compatibility Package**NAME**

sigstack - set and/or get signal stack context

SYNOPSIS

```

/usr/ucb/cc [ flag ... ] file ... -lucb
#include <signal.h>

int sigstack(struct sigstack *ss, struct sigstack *oss);

```

DESCRIPTION

sigstack allows users to define an alternate stack, called the signal stack, on which signals are to be processed. When a signal's action indicates its handler should execute on the signal stack (specified with a [sigvec\(3\)](#) call), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution.

A signal stack is specified by a **sigstack** structure, which includes the following members:

```

char *ss_sp;    /* signal stack pointer */
int ss_onstack; /* current status */

```

ss_sp is the initial value to be assigned to the stack pointer when the system switches the process to the signal stack. On machines where the stack grows downwards in memory, this is *not* the address of the beginning of the signal stack area. **ss_onstack** field is zero or non-zero depending on whether the process is currently executing on the signal stack or not.

If **ss** is not a NULL pointer, **sigstack** sets the signal stack state to the value in the **sigstack** structure pointed to by **ss**. If **ss_onstack** is non-zero, the system will think that the process is executing on the signal stack. If **ss** is a NULL pointer, the signal stack state will be unchanged. If **oss** is not a NULL pointer, the current signal stack state is stored in the **sigstack** structure pointed to by **oss**.

After a successful call to one of the **exec** functions, there are no alternate signal stacks in the new process image.

RETURN VALUE

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

DIAGNOSTICS

sigstack fails and the signal stack context remains unchanged.

EFAULT

Either **ss** or **oss** points to memory that is not a valid part of the process address space.

EPERM

An attempt was made to modify an active stack.

NOTES

Signal stacks are not grown automatically, as is done for the normal stack. If the stack overflows unpredictable results may occur.

A portable application, when being written or rewritten, should use **sigaltstack()** instead of **sigstack()**.

On some implementations, stack space is automatically extended as needed. On those implementations, automatic extension is typically not available for an alternate stack. If a signal stack overflows, the resulting behavior of the process is undefined.

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. The implementation may assume that the size of the signal stack is `SIGSTKSZ` as found in `<signal.h>`. An implementation that would like to specify a signal stack size other than `SIGSTKSZ` should use `sigaltstack()`.

Programs should not use `longjmp()` 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()`, `setcontext()`, or `swapcontext()` 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()` is recommended over this function.

SEE ALSO

`exec(2)`, `fork(2)`, `sigaltstack(2)`, `signal(2)`, `sigvec(3)`, `signal(3-ucb)`, `setjmp(3C)`, `sigsetjmp(3C)`, `signal(5)`.

sigvec(3) BSD Compatibility Package

NAME

sigvec - software signal facilities

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <signal.h>
int sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;
```

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *blocked* or *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special *signal stack*.

All signals have the same *priority*. Signal routines execute with the signal that caused their invocation to be *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a `sigblock()` or `sigsetmask()` call, or when a signal is delivered to the process.

A process may also specify a set of *flags* for a signal that affect the delivery of that signal.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a `sigblock()` or `sigsetmask()` call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and using the logical OR in the signal mask associated with the handler to be invoked.

The action to be taken when the signal is delivered is specified by a `sigvec` structure, which includes the following members:

```
void    (*sv_handler)();          /* signal handler */
int     sv_mask;                  /* signal mask to apply */
int     sv_flags;                 /* see signal options */
#define SV_ONSTACK    /* take signal on signal stack */
#define SV_INTERRUPT  /* do not restart system on signal return */
#define SV_RESETHAND  /* reset handler to SIG_DFL when signal taken */
```

If the `SV_ONSTACK` bit is set in the flags for that signal, the system will deliver the signal to the process on the signal stack specified with `sigstack`, rather than delivering the signal on the current stack.

If `vec` is not a `NULL` pointer, `sigvec()` assigns the handler specified by `sv_handler`, the mask specified by `sv_mask`, and the flags specified by `sv_flags` to the specified signal. If `vec` is a `NULL` pointer, `sigvec()` does not change the handler, mask, or flags for the specified signal.

The mask specified in `vec` is not allowed to block `SIGKILL`, `SIGSTOP`, or `SIGCONT`. The system enforces this restriction silently.

If *ovec* is not a NULL pointer, the handler, mask, and flags in effect for the signal before the call to `sigvec()` are returned to the user. A call to `sigvec()` with *vec* a NULL pointer and *ovec* not a NULL pointer can be used to determine the handling information currently in effect for a signal without changing that information.

The following is a list of all signals with names as in the include file `/usr/include/signal.h`:

SIGHUP		Hangup
SIGINT		Interrupt
SIGQUIT	*	Quit
SIGILL	*	Illegal instruction
SIGTRAP	*	Trace trap
SIGABRT	*	Abort (generated by <code>abort(3C)</code> routine)
SIGEMT	*	Emulator trap
SIGFPE	*	Arithmetic exception
SIGKILL		Kill (cannot be caught, blocked, or ignored)
SIGBUS	*	Bus error
SIGSEGV	*	Segmentation violation
SIGSYS	*	Bad argument to system call
SIGPIPE		Write on a pipe or other socket with no one to read it
SIGALRM		Alarm clock
SIGTERM		Software termination signal
SIGURG	*b	Urgent condition present on socket
SIGSTOP	*d	Stop (cannot be caught, blocked, or ignored)
SIGTSTP	*d	Stop signal generated from keyboard
SIGCONT	*b	Continue after stop (cannot be blocked)
SIGCHLD	*b	Child status has changed
SIGTTIN	*d	Background read attempted from control terminal
SIGTTOU	*d	Background write attempted to control terminal
SIGIO	*b	I/O is possible on a

		descriptor [see fcntl(2)]
SIGXCPU		CPU time limit exceeded [see setrlimit(2)]
SIGXFSZ		File size limit exceeded [see setrlimit(2)]
SIGVTALRM		Virtual time alarm [see setitimer(3C)]
SIGPROF		Profiling timer alarm [see setitimer(3C)]
SIGWINCH	*b	Window changed [see termio(7)]
SIGUSR1		User-defined signal 1
SIGUSR2		User-defined signal 2

The starred signals ("*") in the list above cause a core image if not caught or ignored.

Once a signal handler is installed, it remains installed until another `sigvec()` call is made, or an `execve()` is performed, unless the `SV_RESETHAND` bit is set in the flags for that signal. In that case, the value of the handler for the caught signal will be set to `SIG_DFL` before entering the signal-catching function, unless the signal is `SIGILL`, `SIGPWR`, or `SIGTRAP`. Also, if this bit is set, the bit for that signal in the signal mask will not be set; unless the signal mask associated with that signal blocks that signal, further occurrences of that signal will not be blocked. The `SV_RESETHAND` flag is not available in BSD 4.2, hence it should not be used if backward compatibility is needed.

The default action for a signal may be reinstated by setting the signal's handler to `SIG_DFL`; this default is termination except for signals marked with "*b" or "*d". Signals marked with "*b" are discarded if the action is `SIG_DFL`; signals marked with "*d" cause the process to stop. If the process is terminated, a core image will be made in the current working directory of the receiving process if the signal is one for which an asterisk appears in the above list.

If the handler for that signal is `SIG_IGN`, the signal is subsequently ignored, and pending instances of the signal are discarded.

If a caught signal occurs during certain system calls, the call is normally restarted. The call can be forced to terminate prematurely with an `EINTR` error return by setting the `SV_INTERRUPT` bit in the flags for that signal. The `SV_INTERRUPT` flag is not available in BSD 4.2, hence it should not be used if backward compatibility is needed. The affected system calls are `read()` or `write()` on a slow device (such as a terminal or pipe or other socket, but not a file) and during a `wait()`.

After a `fork()` or `vfork()` the child inherits all signals, the signal mask, the signal stack, and the restart/interrupt and reset-signal-handler flags.

The `execve()` call resets all caught signals to default action and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that interrupt system calls continue to do so.

The accuracy of *addr* is machine dependent. For example, certain machines may supply an address that is on the same page as the address that caused the fault. If an appropriate *addr* cannot be computed it will be set to `SIG_NOADDR`.

RETURN VALUE

A 0 value indicates that the call succeeded. A -1 return value indicates that an error occurred and `errno` is set to indicate the reason.

ERRORS

`sigvec()` will fail and no new signal handler will be installed if one of the following occurs:

EFAULT

Either `vec` or `ovec` is not a `NULL` pointer and points to memory that is not a valid part of the process address space.

EINVAL

`sig` is not a valid signal number, or, `SIGKILL`, or `SIGSTOP`.

NOTES

`SIGPOLL` is a synonym for `SIGIO`. A `SIGIO` will be issued when a file descriptor corresponding to a `STREAMS` [see `intro_prm2(2)`] file has a selectable event pending. Unless that descriptor has been put into asynchronous mode [see `fcntl(2)`], a process must specifically request that this signal be sent using the `I_SETSIG` `ioctl` call [see `streamio(7)`]. Otherwise, the process will never receive `SIGPOLL`.

The handler routine can be declared like this:

```
void handler(sig, code, scp, addr)
int sig, code;
struct sigcontext *scp;
char *addr;
```

In this example:

sig Signal number.

code Parameter of certain signals that provides additional detail.

scp Pointer to the `sigcontext` structure (defined in `signal.h`), used to restore the context from before the signal and *addr* is additional address information.

The signals `SIGKILL`, `SIGSTOP`, and `SIGCONT` cannot be ignored.

SEE ALSO

`exec(2)`, `fcntl(2)`, `fork(2)`, `getrlimit(2)`, `ioctl(2)`, `kill(2)`, `ptrace(2)`, `read(2)`, `signal(2)`, `umask(2)`, `wait(2)`, `write(2)`, `sigblock(3)`, `sigpause(3)`, `sigsetmask(3)`, `sigstack(3)`, `wait3(3)`, `getitimer(3C)`, `setjmp(3C)`, `setjmp(3C-ucb)`, `signal(3-ucb)`, `signal(5)`, `streamio(7)`, `termio(7)`.

syscall(3)
BSD Compatibility Package**NAME**

`syscall` - indirect system call

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb  
include <sys/syscall.h>  
int syscall(number, arg, ...);
```

DESCRIPTION

`syscall()` performs the system call whose assembly language interface has the specified *number*, and arguments *arg* ... Symbolic constants for system calls can be found in the header file `/usr/include/sys/syscall.h`.

RETURN VALUE

When the C bit is set, `syscall()` returns -1 and sets the external variable `errno` [see [intro_prm2\(2\)](#)].

SEE ALSO

[intro_prm2\(2\)](#), [pipe\(2\)](#).

NAME

ualarm - schedule signal after interval in microseconds

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <unistd.h>
useconds_t ualarm(useconds_t useconds, useconds_t interval);
```

DESCRIPTION

ualarm() sends signal `SIGALRM` to the invoking process in a number of microseconds given by the *useconds* argument. Unless caught or ignored, the signal terminates the process.

If the *interval* argument is non-zero, the `SIGALRM` signal will be sent to the process every *interval* microseconds after the timer expires (for instance, after *useconds* microseconds have passed).

Because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 2.147.483.647 microseconds.

Interactions between **ualarm()** and either **alarm()** or **sleep()** are unspecified.

RETURN VALUE

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.

NOTES

ualarm() is a simplified interface to **setitimer(3C)**, and uses the `ITIMER_REAL` interval timer.

SEE ALSO

alarm(2), **sigpause(3)**, **sigvec(3)**, **usleep(3)**, **signal(3-ucb)**, **getitimer(3C)**, **sleep(3C-ucb)**, **unistd(4)**.

usleep(3)
BSD Compatibility Package**NAME**

usleep - suspend execution for interval in microseconds

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <unistd.h>
int usleep(useconds_t useconds);
```

DESCRIPTION

The `usleep()` function will cause the calling thread to be suspended from execution until either the number of real-time microseconds specified by the argument *useconds* 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 due to the scheduling of other activity by the system.

The *useconds* argument must be less than 1,000,000. If the value of *useconds* is 0, then the call has no effect.

If a `SIGALARM` signal is generated for the calling process during execution of `usleep` and if the `SIGALARM` signal is being ignored or blocked from delivery, it is unspecified whether `usleep()` returns when the `SIGALARM` signal is scheduled. If the signal is being blocked, it is also unspecified whether it remains pending after `usleep()` returns or it is discarded.

If a `SIGALARM` signal is generated for the calling process during execution of `usleep()`, except as a result of a prior call to `alarm()`, and if the `SIGALARM` signal is not being ignored or blocked from delivery, it is unspecified whether that signal has any effect other than causing `usleep()` to return.

If a signal-catching function interrupts `usleep()` and examines or changes either the time a `SIGALARM` is scheduled to be generated, the action associated with the `SIGALARM` signal, or whether the `SIGALARM` signal is blocked from delivery, the results are unspecified.

If a signal-catching function interrupts `usleep` and calls `siglongjmp()` or `longjmp()` to restore an environment saved prior to the `usleep()` call, the action associated with the `SIGALARM` signal and the time at which a `SIGALARM` signal is scheduled to be generated are unspecified. It is also unspecified whether the `SIGALARM` signal is blocked, unless the process' signal mask is restored as part of the environment.

Implementations may place limitations on the granularity of timer values. For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer value will be rounded up to the next supported value.

Interactions between `usleep()` and `setitimer()`, `alarm()` or `sleep()` are unspecified.

RETURN VALUE

On successful completion, `usleep()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

The `usleep()` function may fail if:

`EINVAL`

The time interval specified 1,000,000 or more microseconds.

NOTES

Applications are recommended to use `setitimer()` instead of `usleep()`.

SEE ALSO

`alarm(2)`, `sigaction(2)`, `sigpause(3)`, `ualarm(3)`, `getitimer(3C)`, `sleep(3C-ucb)`,

`unistd(4)`.

C programming functions from the standard library libc(3C)

a64l(3C) - ctype(3C)

Programmer's Reference Manual

a64l(3C)

NAME

`a64l`, `l64a` - convert between 32-bit integer and radix-64 ASCII string

SYNOPSIS

```
#include <stdlib.h>
long a64l(const char *s);
char *l64a(long value);
```

DESCRIPTION

These functions are used to maintain numbers stored in radix-64 ASCII characters. This is a notation by which 32-bit integers can be represented by up to six characters; each character represents a digit in radix-64 notation. If the type `long` contains more than 32 bits, only the low-order 32 bits are used for these operations.

The characters used to represent "digits" are "." for 0, "/" for 1, "0" through "9" for 2–11, "A" through "Z" for 12–37, and "a" through "z" for 38–63.

The `a64l()` function takes a pointer to a radix-64 representation, in which the first digit is the least significant, and returns a corresponding `long` value. If the string pointed to by `s` contains more than six characters, `a64l()` uses the first six. If the first six characters of the string contain a null terminator, `a64l()` uses only characters preceding the null terminator. 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. If the type `long` contains more than 32 bits, the resulting value is sign-extended. The behavior of `a64l()` is unspecified if `s` is a null pointer or the string pointed to by `s` was not generated by a previous call to `l64a()`.

The `l64a()` function takes a `long` argument and returns a pointer to the corresponding radix-64 representation. The behavior of `l64a()` is unspecified if `value` is negative.

The value returned by `l64a()` may be a pointer into a static buffer. Subsequent calls to `l64a()` may overwrite the buffer.

This interface need not be thread-safe.

RETURN VALUE

On successful completion, `a64l()` returns the `long` value resulting from conversion of the input string. If a string pointed to by `s` is an empty string, `a64l()` returns 0L.

The `l64a()` function returns a pointer to the radix-64 representation. If `value` is 0L, `l64a()` returns a pointer to an empty string.

ERRORS

No errors are defined.

APPLICATION USAGE

If the type `long` contains more than 32 bits, the result of `a64l(l64a(x))` is `x` in the low-order 32 bits.

SEE ALSO

`strtoul(3C)`, `stdlib(5)`.

abort(3C)**NAME**

`abort` - generate an abnormal termination signal

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()` on all open streams and message catalog descriptors, and the default actions defined for `SIGABRT`. The `SIGABRT` signal is sent to the calling process as if by means of the `raise()` function with the argument `SIGABRT`.

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

RETURN VALUE

If `SIGABRT` is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message `abort() - core dumped` is written by the shell [see [sh\(1\)](#)].

SEE ALSO

[sh\(1\)](#), [exit\(2\)](#), [kill\(2\)](#), [signal\(2\)](#), [stdio\(3S\)](#), [stdlib\(5\)](#).

NAME

`abs`, `labs` - return integer absolute value

SYNOPSIS

```
#include <stdlib.h>
int abs(int val);
long int labs(long int lval);
```

DESCRIPTION

`abs()` returns the absolute value of its `int` operand. `labs()` returns the absolute value of its `long` operand.

NOTES

In two's-complement representation, the absolute value of the negative integer with largest magnitude `INT_MIN` might not be representable.

SEE ALSO

`fabs(3M)`, `floor(3M)`, `stdlib(5)`.

addseverity(3C)**NAME**

`addseverity` - build a list of severity levels

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()`. *severity* is an integer value indicating the seriousness of the condition, and *string* is a pointer to a string describing the condition. This string is not limited to any particular 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\(3C\)](#)].

EXAMPLES

When the function `addseverity()` is used as follows:

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

the following call to `fmtmsg()`:

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

produces:

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

RETURN VALUE

`addseverity()` returns `MM_OK` on success or `MM_NOTOK` on failure.

SEE ALSO

[fmtmsg\(1\)](#), [fmtmsg\(3C\)](#), [gettxt\(3C\)](#), [printf\(3S\)](#), [fmtmsg\(5\)](#).

NAME

`atexit` - add program termination routine

SYNOPSIS

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

DESCRIPTION

`atexit()` adds the function *func* to a list of functions to be called without arguments on normal termination of the program. A program is terminated normally either by a call to `exit()` or by a return from `main`. At most 32 functions may be registered by `atexit()`; the functions will be called in the reverse order of their registration.

Any functions previously registered by `atexit()` are no longer registered after a successful call to one of the `exec` functions.

`atexit()` returns 0 if the registration succeeds, non-zero if it fails.

APPLICATION USAGE

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

The application should call `sysconf()` to obtain the value of `ATEXIT_MAX`, the number of functions that can be registered. There is no way for an application to tell how many functions have already been registered with `atexit()`.

SEE ALSO

`exit(2)`, `sysconf(3C)`, `stdlib(5)`.

basename(3C)**NAME**

basename - return the last element of a path name

SYNOPSIS

```
#include <libgen.h>
char *basename(char *path);
```

DESCRIPTION

The `basename()` function takes the path name pointed to by *path* and returns a pointer to the final component of the path name, 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 `."`.

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()`.

This interface need not be thread-safe.

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

Input string	Output string
<code>"/usr/lib"</code>	<code>"lib"</code>
<code>"/usr/"</code>	<code>"usr"</code>
<code>"/"</code>	<code>"/"</code>

SEE ALSO

`basename(1)`, `dirname(3G)`, `libgen(5)`.

bsd_signal(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 **APPLICATION 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 VALUE

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)`.

APPLICATION 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()`. 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()`.

SEE ALSO

`sigaction(2)`, `signal(3-ucb)`, `sigaddset(3C)`, `sigemptyset(3C)`, `signal(5)`.

bsearch(3C)**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

bsearch() is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table (an array) indicating where data may be found. It returns a null pointer if the data cannot be found. *key* points to a data instance to be sought in the table. *base* points to the element at the base of the table. *nel* is the number of elements in the table. *size* is the number of bytes in each element. The function pointed to by *compar* is called with two arguments that point to *key* and an element of the table (in this order). The function must return an integer less than, equal to, or greater than 0 as accordingly the first argument is to be considered less than, equal to, or greater than the second. The table must contain the following elements: first all elements that are less than *key*, then all elements that are equal to *key*, and finally all elements that are greater than *key*.

EXAMPLES

This example 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 {
    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));
}

```

RETURN VALUE

A null pointer is returned if the key cannot be found in the table; otherwise, a pointer is returned, pointing to the required place in the table.

NOTES

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.

SEE ALSO

[bsearch\(3C\)](#), [lsearch\(3C\)](#), [qsort\(3C\)](#), [tsearch\(3C\)](#), [stdlib\(5\)](#).

NAME

bstring:
bcopy, bcmp, bzero - bit and byte string 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 functions `bcopy()`, `bcmp()`, and `bzero()` operate on variable length strings of bytes. They do not check for null bytes as the routines in `string(3C)` do.

`bcopy()` copies *n* bytes from string *s1* to the string *s2*. Overlapping strings are handled correctly.

The `bcmp()` function compares the first *n* bytes of the area pointed to by *s1* with the area pointed to by *s2*.

`bzero()` places *n* 0 bytes in the string *s*.

RETURN VALUE

The `bcopy()` function returns no value.

The `bcmp()` function returns 0 if *s1* and *s2* are identical, non-zero otherwise. Both areas are assumed to be *n* bytes long. If the value of *n* is 0, `bcmp()` returns 0.

The `bzero()` function returns no value.

SEE ALSO

`memory(3C)`, `string(3C)`, `strings(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 (single-byte) character in the initial shift state.

The behavior of this function is affected by the `LC_CTYPE` category of the current locale.

RETURN VALUE

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

ERRORS

No errors are defined.

SEE ALSO

`wctob(3C)`, `stdio(5)`, `wchar(5)`.

NAME

`__clone`, `clone` - create a child process

SYNOPSIS

```
int __clone(int (* fn) (void *arg), void *  
child_stack, int flags, void * arg);
```

DESCRIPTION

`__clone` creates a new process like `fork(2)` does. Unlike `fork(2)`, `__clone` allows the child process to share parts of its execution context with its parent process, such as the memory space, the table of file descriptors, and the table of signal handlers. The main use of `__clone` is to implement threads: multiple threads of control in a program that run concurrently in a shared memory space. *Note:* This function should not be used directly. Normal usage is only from the Pthread library.

When the child process is created, it executes the function application `fn(arg)`. The `fn` argument is a pointer to the function that is called by the child process at the beginning of its execution. The `arg` argument is passed back to the `fn` function.

When the `fn(arg)` function application returns, the child process terminates. The integer returned by `fn` is the exit code for the child process. The child process may also terminate explicitly by calling `exit(2)` or after receiving a fatal signal.

The `child_stack` argument specifies the location of the stack used by the child process. Since the child and parent processes share memory, it is not possible in general for the child process to execute in the same stack as the parent process. The parent process must therefore set up memory space for the child stack and pass a pointer to this space to `__clone`. Since stacks grow downwards `child_stack` must point to the topmost address of the memory space set up for the child stack.

The low byte of `flags` contains the number of the signal sent to the parent when the child dies.

Amount of Sharing

- Memory space □
The parent and the child processes run in the same memory space. In particular, memory writes performed by the parent process or by the child process are also visible in the other process. Moreover, any memory mapping or unmapping performed with `mmap(2)` or `munmap(2)` by the child or parent process also affects the other process.
- File descriptors □
The parent and the child processes share the same file descriptor table. File descriptors always refer to the same files in the parent and in the child process. Any file descriptor created by the parent process or by the child process is also valid in the other process. Similarly, if one of the processes closes a file descriptor, or changes its associated flags, the other process is also affected.
- Signal handler □
The parent and the child processes share the same table of signal handlers. If the parent or child process calls `sigaction(2)` to change the behavior associated with a signal, the behavior is also changed in the other process as well. However, the parent and child processes still have distinct signal masks and sets of pending signals. So, one of them may block or unblock some signals using `sigprocmask(2)` without affecting the other process.
- File system □
The parent and the child processes share the same file system information. This includes the root of the file system, the current working directory, and the umask. Any call to `chroot(2)`, `chdir(2)`, or `umask(2)` performed by the parent or child process also takes effect in the other process.

RETURN VALUES

On success, the PID of the child process is returned in the parent's thread of execution. On failure, a `-1` will be returned in the parent's context, no child process will be created, and `errno` will be set appropriately.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

EAGAIN

The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user `CHILD_MAX` would be exceeded.

EAGAIN

Total amount of system memory available when reading via raw I/O is temporarily insufficient.

EINVAL

The *child_stack* argument is not a valid address.

ENOMEM

The swap space is too small.

SEE ALSO

[fork\(2\)](#), [pthread_create\(3T\)](#).

catgets(3C)**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

`catgets()` attempts to read message *msg_num*, in set *set_num*, from the message catalog identified by *catd*. *catd* is a catalog descriptor returned from an earlier call to `catopen()`. *s* points to a default message string which will be returned by `catgets()` if the function fails.

This interface need not be thread-safe.

RETURN VALUE

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 because the message catalog identified by *catd* is not currently available, a pointer to *s* is returned and `errno` may be set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `catgets()` function may fail if:

EBADF

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

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_num* and *msg_num* is not in the message catalog.

SEE ALSO

[catclose\(3C\)](#), [catopen\(3C\)](#), [nl_types\(5\)](#).

catopen(3C)**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

`catopen()` opens a message catalog and returns a catalog descriptor. *name* specifies the name to be used in opening the message catalog. *name* contains a / then *name* specifies a pathname for the message catalog. Otherwise, the environment variable `NLSPATH` is used. 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 is used [see [nl_types\(5\)](#)]. This default may be affected by the setting of `LC_MESSAGES` if the value of *oflag* is `NL_CAT_LOCALE`, or the `LANG` environment variable if *oflag* is 0.

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.

The names of message catalogs, and their location in the filestore, differ 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 for specifying exactly 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 the `LANG` environment variable (see following section), and %N substitutes the value of the *name* parameter passed to `catopen()`. Thus, in the above example, `catopen()` will search in `/nlslib/$LANG/name.cat`, then in `/nlslib/name/$LANG`, for the required message catalog.

`NLSPATH` will normally be set up on a system wide basis (e.g., 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.
%l    The value of the language element of LANG.
%t    The value of the territory element of LANG.
%c    The value of the codeset element of LANG.
%%    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 message catalogs, should they exist.

Should the `LANG` variable not be set then the value of `LC_MESSAGES` as returned by `setlocale` is used. If

this is `NULL` then the default path as defined in `nl_types` is used.

If the value of the `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.

`catclose()` closes the message catalog identified by `catd`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

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

`OPEN_MAX` file descriptors are 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.

RESULT

If successful, `catopen()` returns a message catalog descriptor for use on subsequent calls to `catgets()` and `catclose()`. Otherwise `catopen()` returns `(nl_catd)-1`.

`catclose()` returns 0 if successful, otherwise `-1`.

NOTES

Some implementations of `catopen()` use `malloc()` to allocate space for internal buffer areas. The `catopen()` function may fail if there is insufficient storage space available to accommodate these buffers.

Portable applications must assume that message catalogue descriptors are not valid after a call to one of the `exec` functions.

Application writers should be aware that guidelines for the location of message catalogues have not yet been developed. Therefore they should take care to avoid conflicting with catalogues used by other applications and the standard utilities.

SEE ALSO

```
catgets(3C), setlocale(3C), environ(5), fcntl(5), nl_types(5).
```

NAME

`clock` - report CPU time used

SYNOPSIS

```
#include <time.h>
□
clock_t clock(void);
```

DESCRIPTION

`clock()` returns the 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()` system call, the `pclose()` function, or the `system()` function.

Dividing the value returned by `clock()` by the constant `CLOCKS_PER_SEC`, defined in the `time.h` header file, will give the time in seconds.

The resolution of the clock unit is hardware dependent.

NOTES

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). If the process time used is not available or cannot be represented, `clock()` returns the value `(clock_t)-1`.

SEE ALSO

`times(2)`, `wait(2)`, `popen(3S)`, `system(3S)`, `time(5)`.

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()` function, but it is used where string values rather than numeric values are returned.

The *name* argument represents the system variable to be queried. The implementation supports the *name* value of `_CS_PATH`, defined in `<unistd.h>`.

```
_CS_PATH
_CS_XBS5_ILP32_OFF32_CFLAGS
_CS_XBS5_ILP32_OFF32_LDFLAGS
_CS_XBS5_ILP32_OFF32_LIBS
_CS_XBS5_ILP32_OFF32_LINTFLAGS
_CS_XBS5_ILP32_OFFBIG_CFLAGS
_CS_XBS5_ILP32_OFFBIG_LDFLAGS
_CS_XBS5_ILP32_OFFBIG_LIBS
_CS_XBS5_ILP32_OFFBIG_LINTFLAGS
_CS_XBS5_LP64_OFF64_CFLAGS
_CS_XBS5_LP64_OFF64_LDFLAGS
_CS_XBS5_LP64_OFF64_LIBS
_CS_XBS5_LP64_OFF64_LINTFLAGS
_CS_XBS5_LPBIG_OFFBIG_CFLAGS
_CS_XBS5_LPBIG_OFFBIG_LDFLAGS
_CS_XBS5_LPBIG_OFFBIG_LIBS
_CS_XBS5_LPBIG_OFFBIG_LINTFLAGS
```

If *len* is not zero, 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 zero and *buf* is a null pointer, then `confstr()` still returns the integer value as defined below, but does not return a string. If *len* is zero but *buf* is not a null pointer, the result is unspecified.

RETURN VALUE

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 zero and sets `errno` to indicate the error.

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

ERRORS

The `confstr()` function will fail if :

`EINVAL`

The value of the *name* argument is invalid.

APPLICATION USAGE

An application can distinguish between an invalid name parameter value and one that corresponds to a configurable variable that has no configuration-defined value by checking if `errno` is modified. This mirrors the behavior of `sysconf()`.

The original need for this function was to provide a way of finding the configuration-defined default value for the environment variable `PATH`. Since `PATH` can be modified by the user to include directories that could contain utilities replacing XCU specification standard utilities, applications need a way to determine the system-supplied `PATH` environment variable value that contains the correct search path for the standard utilities.

An application could use:

```
confstr (name, (char *)NULL, (size_t) 0)
```

to find out how big a buffer is needed for the string value; use `malloc()` to allocate a buffer to hold the string; and call `confstr()` again to get the string. Alternately, it could allocate a fixed, static buffer that is big enough to hold most answers (perhaps 512 or 1024 bytes), but then use `malloc()` to allocate a larger buffer if it finds that this is too small.

SEE ALSO

`pathconf(2)`, `sysconf(3C)`, `unistd(4)`.

NAME

conv:
 toupper, tolower, _toupper, _tolower, toascii, towlower, towupper - convert characters

SYNOPSIS

```
#include <ctype.h>
int toupper(int c);
int tolower(int c);
int _toupper(int c);
int _tolower(int c);
int toascii(int c);
#include <wctype.h>
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
```

DESCRIPTION

`toupper()` and `tolower()` have as their domain the range of the function `getc(3S)`: all the values represented in an unsigned `char` and the value of the macro `EOF` as defined in `stdio.h`. If the argument of `toupper()` represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of `tolower()` represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

The macros `_toupper()` and `_tolower()` accomplish the same things as `toupper()` and `tolower()`, respectively, but have restricted domains and are faster. `_toupper()` requires a lower-case letter as its argument; its result is the corresponding upper-case letter. `_tolower()` requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

`toascii()` yields its argument with all bits turned off that are not part of a standard 7-bit ASCII character; it is intended for compatibility with other systems.

For an upper-case wide character, the `towlower()` function returns the corresponding lower-case wide character if the corresponding wide character exists in lower-case in the locale. The `wc` argument that is passed to the function must be representable with the data type `wchar_t`, and must match the current language setting or `WEOF`. If `wc` does not satisfy one of these conditions, the result of the function is undefined. If the argument passed is not an upper-case wide character, but corresponds to the character set of `wchar_t`, it will not be converted.

The same applies for the `towupper()` function, but `towupper()` returns, for a lower-case wide character, the corresponding upper-case wide character as defined in the category `LC_CTYPE`.

RETURN VALUE

Upon successful completion, `toupper()` and `towupper()` return the upper-case letter corresponding to the argument passed. Otherwise they return the argument unchanged.

Upon successful completion, `toupper()` and `_toupper()` return the upper-case letter corresponding to the argument passed.

On successful completion, `tolower()` and `towlower()` return the lower-case letter corresponding to the argument passed; otherwise they return the argument unchanged.

On successful completion, `_tolower()` returns the lower-case letter corresponding to the argument passed.

The `toascii()` function returns the value `(c & 0x7f)`.

ERRORS

No errors are defined.

SEE ALSO

`ctype(3C)`, `setlocale(3C)`, `getc(3S)`, `ctype(5)`, `wctype(5)`.

NAME

`crypt`, `setkey`, `encrypt` - generate encryption of strings

SYNOPSIS

```
#include <unistd.h>
char *crypt(const char *key, const char *salt);
void encrypt(char block[64], int edflag);
#include <stdlib.h>
void setkey(const char *key); □
```

DESCRIPTION

`crypt()` is the password encryption function. It is based on a one-way encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

key is the input string to encrypt, for instance, a user's typed password. Only the first eight characters are used; the rest are ignored. *salt* is a two-character string chosen from the set `a-zA-Z0-9./`; this string is used to perturb the hashing algorithm in one of 4096 different ways, after which the input string is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted input string. The first two characters of the return value are the *salt* itself.

The `setkey()` and `encrypt()` functions provide (rather simple) access to the implementation-dependent hashing algorithm. The argument of `setkey()` is a character array of length 64 containing only the characters with numerical value 0 and 1. This string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key that is set into the machine. This is the key that will be used with the hashing algorithm to encrypt the string *block* with the `encrypt()` function.

The `setkey()` interface need not be thread-safe.

The *block* argument of `encrypt()` is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by `setkey()`. If *edflag* is 0, the argument is encoded. If *edflag* is 1, the argument may be decoded; if it is not, `errno` will be set to `ENOSYS`.

RETURN VALUE

The return value for `crypt()` points to static data that is overwritten by each call.

ERRORS

The functions will fail if:

`ENOSYS`

The functionality is not supported by the system.

NOTES

As `setkey()` and `encrypt()` do not return a value, applications wishing to check for errors should set `errno` to 0 and test `errno`. If it is non-zero, an error must have occurred.

SEE ALSO

`login(1)`, `passwd(1)`, `getpass(3C)`, `crypt(3X)`, `passwd(4)`, `unistd(4)`, `stdlib(5)`.

NAME

`ctime`, `ctime_r`, `localtime`, `localtime_r`, `gmtime`, `gmtime_r`, `asctime`, `asctime_r`, `timezone`, `altzone`, `daylight`, `tzname`, `tzset` - convert date and time to string

SYNOPSIS

```
#include <time.h>

char *ctime(const time_t *clock);

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

struct tm *localtime(const time_t *clock);

struct tm *localtime_r(const time_t *clock, struct tm *result);

struct tm *gmtime(const time_t *clock);

struct tm *gmtime_r(const time_t *clock, struct tm *result);

char *asctime(const struct tm *timeptr);

char *asctime_r(const struct tm *tm, char *buf);

extern long int timezone, altzone;
extern int daylight;
extern char *tzname[2];
void tzset(void);
```

DESCRIPTION

`ctime()`, `localtime()`, and `gmtime()` accept arguments of type `time_t`, pointed to by `clock`, representing the time in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970. `ctime()` returns a pointer to a 26-character string in the following form (time zone and daylight savings corrections are made before the string is generated; the fields are constant in width):

```
Fri Sep 13 00:00:00 1986\n\0
```

`ctime()` is equivalent to:

```
asctime(localtime(clock))
```

The `asctime()`, `ctime()`, `gmtime()` and `localtime()` functions return values in one of two static objects: a broken-down time structure and an array of `char`. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

The `ctime()` interface need not be thread-safe.

The `ctime_r()` function converts the calendar time pointed to by `clock` to local time in exactly the same form as `ctime()` and puts the string into the array pointed to by `buf` (which contains at least 26 bytes) and returns `buf`.

Unlike `ctime()`, the thread-safe version `ctime_r()` is not required to set `tzname`.

`localtime()` and `gmtime()` return pointers to `tm` structures, described below. `localtime()` corrects for the main time zone and possible alternate ("daylight savings") time zone; `gmtime()` converts directly to UTC, which is the time the Reliant UNIX system uses internally.

The `localtime()` interface need not be thread-safe.

The `localtime_r()` function converts the calendar time pointed to by `clock` into a broken-down time stored in the structure to which `result` points. The `localtime_r()` function also returns a pointer to that same structure.

Unlike `localtime()`, the thread-safe version is not required to set `tzname`.

The `gmtime()` interface need not be thread-safe.

The `gmtime_r()` function converts the calendar time pointed to by `clock` into a broken-down time expressed as Coordinated Universal Time (UTC). The broken-down time is stored in the structure referred to by `result`. The `gmtime_r()` function also returns the address of the same structure.

`asctime()` converts a `tm` structure to a 26-character string, as shown in the above example, and returns a pointer to the string using the equivalent of the following algorithm:

```
char *asctime(const struct tm *timeptr)
{
    static char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];
    sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_mday, timeptr->tm_hour,
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year);
    return result;
}
```

Declarations of all the functions and externals, and the `tm` structure, are in the `time.h` header file. The structure declaration is:

```
struct    tm {
    int    tm_sec;        /* seconds after the minute - [0, 61] */
                        /* 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.

The `asctime()` interface need not be thread-safe.

The `asctime_r()` function converts the broken-down time in the structure pointed to by `tm` into a string that is placed in the user-supplied buffer pointed to by `buf` (which contains at least 26 bytes) and then returns `buf`.

The external variable `altzone` contains the difference, in seconds, between UTC 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 will handle the new daylight savings time starting with the first Sunday in April, 1987.

`tzset()` uses the contents of the environment variable `TZ` to override the value of the different external variables. The function `tzset()` is called by `asctime()` or may also be called by the user. Further details

about the TZ environment variable will be found in the description of `environ(5)`.

`tzset()` 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
```

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` will be set to 5*60*60, `tzname[1]` is EDT, `altzone` will be 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` will be set positive. Starting and ending times are relative to the alternate 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`. `ctime()`, `localtime()`, `mktime()`, and `strftime()` will 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.

Note that in most installations, TZ is set to the correct value by default when the user logs on, via the local `/etc/profile` file [see `profile(4)` and `timezone(4)`].

RETURN VALUE

The `ctime()` and `localtime()` function return the pointer returned by `asctime()` with that broken-down time as an argument.

Upon successful completion, `localtime_r()` returns a pointer to the structure pointed to by the argument *result*.

Upon successful completion, `asctime()` returns a pointer to the string.

Upon successful completion, `asctime_r()` returns a pointer to a character string containing the date and time. This string is pointed to by the argument *buf*. If the function is unsuccessful, it returns NULL.

The `gmtime()` function returns a pointer to a `struct tm`.

Upon successful completion, `gmtime_r()` returns the address of the structure pointed to by the argument *result*. If an error is detected, or UTC is not available, `gmtime_r()` returns a NULL pointer.

ERRORS

No errors are defined.

NOTES

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 start and end days annually if the form of calendar used is the Julian.

`gmtime()` and `localtime()` do not support localized date and time formats. `strftime(3C)` should be used to achieve maximum portability.

FILES

```
/usr/lib/locale/locale/LC_TIME
```

contains specific date and time information for the local environment

SEE ALSO

`time(2)`, `cftime(3C)`, `clock(3C)`, `getenv(3C)`, `mktime(3C)`, `putenv(3C)`, `setlocale(3C)`,

```
strftime(3C), printf(3S), pthread_intro(3T), profile(4), timezone(4), environ(5),  
time(5).
```

NAME

ctype:
 isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint,
 isgraph, iscntrl, isascii - character handling

SYNOPSIS

```
#include <ctype.h>
int isalpha(int c);
int isupper(int c);
int islower(int c);
int isdigit(int c);
int isxdigit(int c);
int isalnum(int c);
int isspace(int c);
int ispunct(int c);
int isprint(int c);
int isgraph(int c);
int iscntrl(int c);
int isascii(int c); □
```

DESCRIPTION

These macros classify character-coded integer values. Each is a predicate returning non-zero for true, zero 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()`, e.g. `setlocale(LC_CTYPE, newlocale)`.

The macro `isascii()` is defined on all integer values; the other macros are only defined 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 file and represents end-of-file.

`isalpha()`

Tests for any character for which `isupper()` or `islower()` is true, or any character that is one of an implementation-defined set of characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, or `isspace()` is true. In the C locale, `isalpha()` returns true only for the characters for which `isalpha()` or `islower()` is true.

`isupper()`

Tests for any character that is an upper-case letter or is one of an implementation-defined set of characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, `isspace()`, or `islower()` is true. In the C locale, `isalpha()` returns true only for the characters defined as upper-case ASCII characters.

`islower()`

Tests for any character that is a lower-case letter or is one of an implementation-defined set of characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, `isspace()`, or `isalpha()` is true. In the C locale, `islower()` returns true only for the characters defined as lower-case ASCII characters.

`isdigit()`

Tests whether the character is a decimal digit.

`isxdigit()`

Tests whether the character is a hexadecimal digit ([0-9], [A-F] or [a-f]).

`isalnum()`

Tests for any character for which `isalpha()` or `isdigit()` is true (letter or digit).

`isspace()`

Tests for any space, tab, carriage-return, newline, vertical-tab or form-feed (standard white-space characters) or for one of an implementation-defined set of characters for which `isalnum()` is false. In the C locale, `isspace()` returns true only for the standard white-space characters.

`ispunct()`

Tests for any printing character which is neither a space nor a character for which `isalnum()` is true.

`isprint()`

Tests for any printing character, including space.

`isgraph()`

Tests for any printing character, except space.

`iscntrl()`

Tests for any "control character" as defined by the character set.

`isascii()`

Tests for any ASCII character, code between 0 and 0177 inclusive.

All the character classification macros and the conversion functions and macros use a table lookup.

Functions exist for all the above-defined macros. To get the function form, the macro name must be undefined [e.g., `#undef isdigit()`].

RETURN VALUE

If the argument to any of the character handling macros is not in the domain of the function, the result is undefined.

FILES

`/usr/lib/locale/locale/LC_CTYPE`

SEE ALSO

[chrtbl\(1M\)](#), [setlocale\(3C\)](#), [stdio\(3S\)](#), [ascii\(5\)](#), [ctype\(5\)](#), [environ\(5\)](#).

dbm(3C) - fwide(3C)

NAME

dbm:

`dbmopen`, `dbmclose`, `fetch`, `store`, `delete`, `firstkey`, `nextkey` - data base subroutines

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -ldb
```

```
#include <dbm.h>
```

```
typedef struct {
```

```
char *dptr;
```

```
int dsize;
```

```
} datum;
```

```
int dbmopen(char *file);
```

```
int dbmclose(void);
```

```
datum fetch(datum key);
```



```
int store(datum key, datum content);
int delete(datum key);
datum firstkey(void);
datum nextkey(datum key);
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option `-l dbm`.

keys and *contents* are described by the `datum` typedef. A `datum` specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has `.dir` as its suffix. The second file contains all data and has `.pag` as its suffix.

Before a database can be accessed, it must be opened by `dbmopen`. At the time of this call, the files `file.dir` and `file.pag` must exist. An empty database is created by creating zero-length `.dir` and `.pag` files.

A database may be closed by calling `dbmclose`. You must close a database before opening a new one.

Once open, the data stored under a key is accessed by `fetch` and data is placed under a key by `store`. A key (and its associated contents) is deleted by `delete`. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of `firstkey` and `nextkey`. `firstkey` will return the first key in the database. With any key `nextkey` will return the next key in the database. This code will traverse the data base:

```
for (key = firstkey; key.dptr != NULL; key = nextkey(key))
```

RETURN VALUE

All functions that return an `int` indicate errors with negative values. A zero return indicates no error. Routines that return a `datum` indicate errors with a `NULL` (0) *dptr*.

NOTES

The `.pag` file will contain holes so that its apparent size is about four times its actual content.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. `store` will return an error in the event that a disk block fills with inseparable data.

The order of keys presented by `firstkey` and `nextkey` depends on a hashing function, not on anything interesting.

Updating and reading a database concurrently can result in inconsistencies.

The same functionality can be found in the `libns1` library.

SEE ALSO

[ar\(1\)](#), [cat\(1\)](#), [cp\(1\)](#), [tar\(1\)](#), [ndbm\(3C\)](#).

NAME

`difftime` - compute the difference between two calendar times

SYNOPSIS

```
#include <time.h>
□
□
□
double difftime(time_t time1, time_t time0);
```

DESCRIPTION

`difftime()` computes the difference between two calendar times. `difftime()` returns the difference (*time1*−*time0*) expressed in seconds as a `double`. This function is provided because there are no general arithmetic properties defined for type `time_t`.

SEE ALSO

`ctime(3C)`, `time(5)`.

directory(3C)**NAME**

directory:
 opendir, readdir, readdir_r, telldir, seekdir, rewinddir, closedir, readdir64 -
 directory operations

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *filename);
struct dirent *readdir(DIR *dirp);
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
long int telldir(DIR *dirp);
void seekdir(DIR *dirp, long int loc);
void rewinddir(DIR *dirp);
int closedir(DIR *dirp);
struct dirent64 *readdir64(DIR *dirp);
```

DESCRIPTION

`opendir()` opens the directory named by *filename* and associates a directory stream with it. `opendir()` returns a pointer to be used to identify the directory stream in subsequent operations. The directory stream is positioned at the first entry. A null pointer is returned if *filename* cannot be accessed or is not a directory, or if it cannot `malloc(3C)` enough memory to hold a `DIR` structure or a buffer for the directory entries. A successful call to any of the `exec` functions will close any directory streams that are open in the calling process.

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 may be removed from a directory or added to a directory asynchronously to the operation of `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.

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 which may be overwritten by another call to `readdir()` on the same directory stream. This data is 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()` or `rewinddir()`, whether a subsequent call to `readdir()` returns an entry for that file is unspecified.

The `readdir()` function may buffer several directory entries per actual read operation; `readdir()` marks for update the `st_atime` field of the directory each time the directory is actually read.

After a call to `fork()`, either the parent or child (but not both) may continue processing the directory stream using `readdir()`, `rewinddir()` or `seekdir()`. 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.

The `readdir()` interface need not be thread-safe.

The `readdir_r()` function initialises the `dirent` structure referenced by *entry* to represent the directory entry at the current position in the directory stream referred to by *dirp*, store a pointer to this structure at the location referenced by *result*, and positions the directory stream at the next entry.

The storage pointed to by *entry* will be large enough for a `dirent` with an array of `char d_name` member containing at least `NAME_MAX` plus one elements.

On successful return, the pointer returned at **result* will the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer will have the value `NULL`.

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 may buffer several directory entries per actual read operation; the `readdir_r()` function marks for update the `st_atime` field of the directory each time the directory is actually read.

Applications wishing to check for error situations should set `errno` to 0 before calling `readdir_r()`. If `errno` is set to non-zero on return, an error occurred.

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()`, the directory position returned from the `telldir()` is the same as that supplied as a *loc* argument for `seekdir()`.

`seekdir()` sets the position of the next `readdir_r()` operation on the directory stream. The new position reverts to the position associated with the directory stream at the time the `telldir()` operation that provides *loc* was performed. If the value of *loc* was not obtained from an earlier call to `telldir()` or if a call to `rewinddir()` occurred between the call to `telldir()` and the call to `seekdir()`, the results of subsequent calls to `readdir_r()` are unspecified.

`rewinddir()` resets the position of the named directory stream to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to `opendir()` would.

In the `readdir_r()` and `rewinddir()` functions, either the parent or child process (but not both) can continue the directory stream using `readdir_r()`, `rewinddir()` or `seekdir()` after a `fork()` call. If both processes use these functions, behavior is undefined.

`closedir()` closes the named directory stream and frees the `DIR` structure.

There is no functional difference between `readdir_r()` and `readdir64_r()`, except for the interpretation of `struct dirent64` [see [dirent\(4\)](#)].

RETURN VALUE

Upon successful completion, `opendir()` returns a pointer to an object of type `DIR`. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

Upon successful completion, `readdir_r()` returns 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.

If successful, the `readdir_r()` function returns zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The following errors can occur as a result of these operations:

`ENOTDIR`

A component of *filename* is not a directory.

`EACCES`

A component of *filename* denies search permission.

EACCES

Read permission is denied on the specified directory.

EMFILE

The maximum number of file descriptors are currently open.

ENFILE

The system file table is full.

EFAULT

filename points outside the allocated address space.

ELOOP

Too many symbolic links were encountered in translating *filename*

ENAMETOOLONG

The length of the *filename* argument exceeds `PATH_MAX`, or the length of a *filename* component exceeds `NAME_MAX`.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

ENOENT

filename points to the name of a file that does not exist or to an empty string.

The `readdir()` function will fail if:

EOVERFLOW

One of the values in the structure to be returned cannot be represented correctly.

The `readdir()` function 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.

The `readdir_r()` function may fail if:

EBADF

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

`telldir()`, `seekdir()`, and `closedir()` return `-1` on failure and set `errno` to the following value:

EBADF

The file descriptor determined by the directory stream is no longer valid. This results if the directory stream has been closed.

EXAMPLES

Here is a sample program that prints the names of all the files in the current directory:

```
#include <stdio.h>
#include <dirent.h>
main()
{
    DIR *dirp;
    struct dirent *direntp;
    dirp = opendir( "." );
    while ( (direntp = readdir( dirp )) != NULL )
        (void)printf( "%s\n", direntp->d_name );
    closedir( dirp );
    return (0);
}
```

NOTES

`rewinddir()` is implemented as a macro, so its function address cannot be taken.

SEE ALSO

`getdents(2)`, `lstat(2)`, `symlink(2)`, `pthread_intro(3T)`, `dirent(4)`, `limits(4)`, `lfs(5)`, `types(5)`.

NAME

`div`, `ldiv` - 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);
```

DESCRIPTION

`div()` computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. This function 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*.

`div()` returns a structure of type `div_t`, comprising both the quotient and remainder:

```
typedef struct div_t {
    int    quot;    /*quotient*/
    int    rem;     /*remainder*/
} div_t;
```

`ldiv()` is similar to `div()`, except that the arguments and the members of the returned structure (which has type `ldiv_t`) all have type `long int`.

SEE ALSO

[stdlib\(5\)](#).

NAME

`dlclose` - close a `dlopen()` object

SYNOPSIS

```
#include <dlfcn.h>
int dlclose(void *handle);
```

DESCRIPTION

`dlclose()` is used to inform the system that the object referenced by a *handle* returned from a previous `dlopen()` invocation is no longer needed by the application.

The use of `dlclose()` reflects a statement of intent on the part of the process, but does not create any requirement upon the implementation, such as removal of the code or symbols referenced by *handle*. Once an object has been closed using `dlclose()` an application should assume that its symbols are no longer available to `dlsym()`. All objects loaded automatically as a result of invoking `dlopen()` on the referenced object are also closed.

Although a `dlclose()` operation is not required to remove structures from an address space, neither is an implementation prohibited from doing so. The only restriction on such a removal is that no object will be removed to which references have been relocated, until or unless all such references are removed. For instance, an object that had been loaded with a `dlopen()` operation specifying the `RTLD_GLOBAL` flag might provide a target for dynamic relocations performed in the processing of other objects – in such environments, an application may assume that no relocation, once made, will be undone or remade unless the object requiring the relocation has itself been removed.

RETURN VALUE

If the referenced object was successfully closed, `dlclose()` returns 0. If the object could not be closed, or if *handle* does not refer to an open object, `dlclose()` returns a non-zero value. More detailed diagnostic information will be available through `dlerror()`.

ERRORS

No errors are defined.

APPLICATION USAGE

A portable application will employ a *handle* returned from a `dlopen()` invocation only within a given scope bracketed by the `dlopen()` and `dlclose()` operations. Implementations are free to use reference counting or other techniques such that multiple calls to `dlopen()` referencing the same object may return the same object for *handle*. Implementations are also free to re-use a *handle*. For these reasons, the value of a *handle* must be treated as an opaque object by the application, used only in calls to `dlsym()` and `dlclose()`.

SEE ALSO

[dlerror\(3C\)](#), [dlopen\(3C\)](#), [dlsym\(3C\)](#), [dladdr\(3X\)](#), [dlfcn\(5\)](#).

dlerror(3C)**NAME**

`dlerror` - get diagnostic information

SYNOPSIS

```
#include <dlfcn.h>
char *dlerror(void);
```

DESCRIPTION

`dlerror()` returns a null-terminated character string (with no trailing newline) that describes the last error that occurred during dynamic linking processing. 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, will result in `NULL` being returned.

RETURN VALUE

If successful, `dlerror()` returns a null-terminated character string. Otherwise, `NULL` is returned.

ERRORS

No errors are defined.

APPLICATION USAGE

The messages returned by `dlerror()` may reside in a static buffer that is overwritten on each call to `dlerror()`. Application code should not write to this buffer. Programs wishing to preserve an error message should make their own copies of that message. Depending on the application environment with respect to asynchronous execution events, such as signals or other asynchronous computation sharing the address space, portable applications should use a critical section to retrieve the error pointer and buffer.

SEE ALSO

[dlclose\(3C\)](#), [dlopen\(3C\)](#), [dlsym\(3C\)](#), [dladdr\(3X\)](#), [dlfcn\(5\)](#).

dlopen(3C)**NAME**

`dlopen` - gain access to an executable object file

SYNOPSIS

```
#include <dlfcn.h>

void *dlopen(const char *file, int mode);
```

DESCRIPTION

`dlopen()` provides callers at runtime with a shared object specified by *file*. All objects required for the specified shared object are also made available with this object, i.e. all objects specified as a `DT_NEEDED` entry in the `.dynamic` section of the original object. Shared objects, which in turn are required for these objects, are likewise made available, etc.

A successful `dlopen()` returns a *handle* which the caller may use on subsequent calls to `dlsym()` and `dlclose()`. The value of this *handle* should not be interpreted in any way by the caller.

file is the name of the object to be opened: if this name contains a slash, it will be used as a pathname for the object either in absolute form or relative to the current directory; if this is the name of a shared object, it will be expanded to a pathname in accordance with the guidelines specified under **NOTES** No. 1. If the value of *file* is 0, `dlopen()` provides a *handle* on a global symbol object.

This object provides access to the symbols from an ordered set of objects consisting of the original program image file, together with any objects loaded at program startup as specified by that process image file, and the set of objects loaded using a `dlopen()` operation together with the `RTLD_GLOBAL` flag. As the latter set of objects can change during execution, the set identified by *handle* can also change dynamically.

Only a single copy of an object file is brought into the address space, even if `dlopen()` is invoked multiple times in reference to the file, and even if different pathnames are used to reference the file.

The *mode* parameter describes how `dlopen()` will operate upon *file* with respect to the processing of relocations and the scope of visibility of the symbols provided within *file*. When an object is brought into the address space of a process, it may 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. The *mode* parameter governs when these relocations take place and may have the following values:

RTLD_LAZY

Under this mode, only references to data symbols are relocated when the object is loaded. References to functions are not relocated until a given function is invoked for the first time. This mode should result in better performance, since a process may not reference all of the functions in any given shared object.

RTLD_NOW

All necessary relocations are performed when the object is first loaded. This may waste some processing if relocations are performed for functions that are never referenced. This behavior may be useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution will be available.

Any object loaded by `dlopen()` that requires relocations against global symbols can reference the symbols in the original process image file, any objects loaded at program startup, from the object itself as well as any other object included in the same `dlopen()` invocation, and any objects that were loaded in any `dlopen()` invocation and which specified the `RTLD_GLOBAL` flag. To determine the scope of visibility for the symbols loaded with a `dlopen()` invocation, the *mode* parameter should be bitwise or'ed with one of the following values:

RTLD_GLOBAL

The object's 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 loaded with this

mode to be searched.

RTLD_LOCAL

The object's symbols are not made available for the relocation processing of any other object.

The program mapping file and all objects loaded at program startup have the mode `RTLD_GLOBAL`. If neither `RTLD_LOCAL` nor `RTLD_GLOBAL` is specified as the mode, `RTLD_LOCAL` is taken as the default.

If a *file* is specified in multiple `dlopen()` invocations, *mode* is interpreted at each invocation. Note, however, that once `RTLD_NOW` has been specified all relocations will have been completed rendering further `RTLD_NOW` operations redundant and any further `RTLD_LAZY` operations irrelevant. Similarly note that once `RTLD_GLOBAL` has been specified the object will maintain the `RTLD_GLOBAL` status regardless of any previous or future specification of `RTLD_LOCAL`, so long as the object remains in the address space [see [dlclose\(3C\)](#)].

Symbols introduced into a program through calls to `dlopen()` may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous `dlopen()` operations. To resolve the ambiguities such a situation might present, the resolution of a symbol reference to symbol definition is based on a symbol resolution order. Two such resolution orders are defined: *load* or *dependency* ordering. *Load* order establishes an ordering among symbol definitions, such that the definition first loaded (including definitions from the image file and any dependent objects loaded with it) has priority over objects added later [via `dlopen()`]. *Load* ordering is used in relocation processing. *Dependency* ordering uses a breadth-first order starting with a given object, then all of its dependencies, then any dependents of those, iterating until all dependencies are satisfied. With the exception of the global symbol object obtained via a `dlopen()` operation on a *file* of 0, *dependency* ordering is used by the `dlsym()` function. *Load* ordering is used in `dlsym()` operations upon the global symbol object.

When an object is first made accessible via `dlopen()` it and its dependent objects are added in *dependency* order. Once all the objects are added, relocations are performed using *load* order. Note that if an object or its dependencies had been previously loaded, the *load* and *dependency* orders may yield different resolutions.

The symbols introduced by `dlopen()` operations, and available through `dlsym()` are at a minimum those which are exported as symbols of global scope by the object. Typically such symbols will be those that were specified in (for example) C source code as having *extern* linkage.

RETURN VALUE

If *file* cannot be found, cannot be opened for reading, is not of an appropriate object format for processing by `dlopen()`, or if an error occurs during the process of loading *file* or relocating its symbolic references, `dlopen()` will return `NULL`. More detailed diagnostic information will be available through `dlerror()`.

ERRORS

No errors are defined.

NOTES

1. If other shared objects were link edited with *file* when *file* was built, those objects will automatically be loaded by `dlopen()`. The directory search path that will be used to find both *file* and the other needed objects is the same as that used by the runtime loader. In particular, *file* is searched for in
 - a) the directory specified by *file* if it is not a simple file name (i.e. it contains a "/" character). In this case, the exact file is the only placed searched; steps b) through d) below are ignored.
 - b) any path specified via the `LD_RUN_PATH` environment variable set when the executable was link edited [see [ld\(1\)](#)].
 - c) any directory specified by the environment variable `LD_LIBRARY_PATH` or `LD_LIBRARY64S_PATH`.
 These environment variables should contain a colon-separated list of directories, in the same format as the `PATH` variable [see [sh\(1\)](#)]. 32-bit programs will only examine `LD_LIBRARY_PATH`, `LD_LIBRARY64S_PATH` will be ignored. 64-bit programs will examine `LD_LIBRARY64S_PATH` instead, if it is set; `LD_LIBRARY_PATH` will be ignored then. If

`LD_LIBRARY64_PATH` is not set, 64-bit programs will examine the variable `LD_LIBRARY_PATH`.

Both variables will be ignored if the process is running `setuid()` or `setgid()` [see [exec\(2\)](#)].

- d) the default search paths will be used. These are `/lib:/usr/lib:/usr/lib/cmplrs/cc` for 32-bit programs, `/lib64:/usr/lib64:/usr/lib64/cmplrs/cc` for 64-bit programs.

2. Objects loaded by a single invocation of `dlopen()` may import symbols from one another or from any object loaded automatically during program startup, but objects loaded by one `dlopen()` invocation may not directly reference symbols from objects loaded by a different `dlopen()` invocation (unless loaded using the `RTLD_GLOBAL` mode). Those symbols may, however, be referenced indirectly using `dlsym()`.

Users who wish to gain access to the symbol table of the `a.out` itself using `dlopen(0, mode)` should be aware that some symbols defined in the `a.out` may not be available to the dynamic linker. The symbol table created by `ld` for use by the dynamic linker might contain only a subset of the symbols defined in the `a.out`: specifically those referenced by the shared objects with which the `a.out` is linked.

3. The dynamic linker is only available with the `libc` dynamic library (do not use `-dn`).

SEE ALSO

[dlclose\(3C\)](#), [dlerror\(3C\)](#), [dlsym\(3C\)](#), [dladdr\(3X\)](#), [dlfcn\(5\)](#).

Chapter on "The C compilation system" in "Guide to Tools for Programming in C".

NAME

`dlsym` - obtain the address of a symbol from a `dlopen()` object

SYNOPSIS

```
#include <dlfcn.h>

void *dlsym(void *handle, const char *name);
```

DESCRIPTION

`dlsym()` allows a process to obtain the address of a symbol defined within an object made accessible through a `dlopen()` call. *handle* is the value returned from a call to `dlopen()` [and which has not since been released via a call to `dlclose()`], *name* is the symbol's name as a character string.

`dlsym()` will search for the named symbol in all objects loaded automatically as a result of loading the object referenced by *handle* [see [dlopen\(3C\)](#)]. Load ordering is used in `dlsym()` operations upon the global symbol object. The symbol resolution algorithm used will be *dependency* order as described in `dlopen()`.

RETURN VALUE

If *handle* does not refer to a valid object opened by `dlopen()`, or if the named symbol cannot be found within any of the objects associated with *handle*, `dlsym()` will return `NULL`. More detailed diagnostic information will be available through `dlerror()`.

ERRORS

No errors are defined.

EXAMPLES

The following example shows how one can 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);
```

APPLICATION USAGE

Special purpose values for *handle* are reserved for future use. These values and their meanings are:

RTLD_NEXT

Specifies the next object after this one that defines *name*. *This one* refers to the object containing the invocation of `dlsym()`. The *next* object is the one found upon the application of a *load* order symbol resolution algorithm [see [dlopen\(3C\)](#)]. The next object is either one of global scope (because it was introduced as part of the original process image or because it was added with a `dlopen()` operation including the `RTLD_GLOBAL` flag), or is an object that was included in the same `dlopen()` operation that loaded this one.

The `RTLD_NEXT` flag is useful to navigate an intentionally created hierarchy of multiply defined symbols created through *interposition*. For example, if a program wished to create an implementation of `malloc()` that embedded some statistics gathering about memory allocations, such an implementation could use the real `malloc()` definition to perform the memory allocation – and itself only embed the necessary logic to implement the statistics gathering function.

SEE ALSO

`dlclose(3C)`, `dlerror(3C)`, `dlopen(3C)`, `dladdr(3X)`, `dlfcn(5)`.

dm_settime(3C)**NAME**

dm_settime, dm_fsettime - set all file times

SYNOPSIS

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

int dm_settime(char *file, struct timeval *atp, struct timeval *mtp, □
□□□□□□□□□□□□ struct timeval *ctp); □

int dm_fsettime(int fd, struct timeval *atp, struct timeval *mtp, □
□□□□□□□□□□□□ struct timeval *ctp); □
```

DESCRIPTION

dm_settime() and dm_fsettime() allow the access, modification, and change times of a file [as returned by `stat(2)`] to be set independently. These calls are intended for use by data management applications performing services such as file backup, recovery, and migration. Use of these calls is restricted to the super-user.

In the case where the named file is a symbolic link, dm_settime() operates on the symbolic link itself. dm_fsettime() is like dm_settime() but works with an open file referenced by the argument descriptor, such as would be obtained by an `open(2)` call.

If any of *atp*, *mtp*, or *ctp* is NULL, then the file's access, modified, or change time respectively will not be changed. If *atp* is not NULL, the file's access time is set to the specified value. If *mtp* is not NULL, the file's modification time is set to the specified value. If *ctp* is not NULL, the file's change time is set to the specified value. Unlike `utime(2)`, the file access or modification time can be set without altering the file's change time.

RETURN VALUES

dm_settime() and dm_fsettime() returns:

- 0 on success.
- 1 on failure and sets `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

EPERM

The effective user ID of the process is not super-user.

EINVAL

The file system containing the file does not support independent setting of file times.

EROFS

The file system containing the file is mounted read-only.

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

In addition, dm_settime() will fail if one or more of the following are true:

ELOOP

Too many symbolic links were encountered in translating *file*.

ENOENT

The file referred to by *file* does not exist.

ENOTDIR

A component of the path prefix of *file* is not a directory.

EACCES

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

EFAULT

file, *atp*, *mtp*, or *ctp* points outside the process' allocated address space.

In addition, `dm_fsettime()` will fail if one or more of the following are true:

EBADF

fd is not a valid open file descriptor.

EFAULT

atp, *mtp*, or *ctp* points outside the process' allocated address space.

SEE ALSO

`stat(2)`, `utime(2)`, `types(5)`.

NAME

`doconfig` - execute a configuration script

SYNOPSIS

```
#include <sac.h>

int doconfig(int fd, char *script, long rflag);
```

DESCRIPTION

`doconfig` is a Service Access Facility library function that interprets the configuration scripts contained in the files `/etc/saf/pmtag/_config`, `/etc/saf/_sysconfig`, and `/etc/saf/pmtag/svctag`.

`script` is the name of the configuration script; `fd` is a file descriptor that designates the stream to which stream manipulation operations are to be applied; `rflag` is a bitmask that indicates the mode in which `script` is to be interpreted. `rflag` may take two values, `NORUN` and `NOASSIGN`, which may be or'd. If `rflag` is zero, all commands in the configuration script are eligible to be interpreted. If `rflag` has the `NOASSIGN` bit set, the `assign` command is considered illegal and will generate an error return. If `rflag` has the `NORUN` bit set, the `run` and `runwait` commands are considered illegal and will generate error returns.

The configuration language in which `script` is written consists of a sequence of commands, each of which is interpreted separately. The following reserved keywords are defined: `assign`, `push`, `pop`, `runwait`, and `run`. The comment character is `#`; when a `#` occurs on a line, everything from that point to the end of the line is ignored. Blank lines are not significant. No line in a command script may exceed 1024 characters.

`assign` *variable*=*value*

Used to define environment variables. *variable* is the name of the environment variable and *value* is the value to be assigned to it. The value assigned must be a string constant; no form of parameter substitution is available. *value* may be quoted. The quoting rules are those used by the shell for defining environment variables. `assign` will fail if space cannot be allocated for the new variable or if any part of the specification is invalid.

`push` *module1* [, *module2*, *module3*, ...]

Used to push STREAMS modules onto the stream designated by *fd*. *module1* is the name of the first module to be pushed, *module2* is the name of the second module to be pushed, etc. The command will fail if any of the named modules cannot be pushed. If a module cannot be pushed, the subsequent modules on the same command line will be ignored and modules that have already been pushed will be popped.

`pop` [*module*]

Used to pop STREAMS modules off the designated stream. If `pop` is invoked with no arguments, the top module on the stream is popped. If an argument is given, modules will be popped one at a time until the named module is at the top of the stream. If the named module is not on the designated stream, the stream is left as it was and the command fails. If *module* is the special keyword `ALL`, then all modules on the stream will be popped. Note that only modules above the topmost driver are affected.

`runwait` *command*

The `runwait` command runs a command and waits for it to complete. *command* is the pathname of the command to be run. The command is run with `/usr/bin/sh -c` prepended to it; shell scripts may thus be executed from configuration scripts. The `runwait` command will fail if *command* cannot be found or cannot be executed, or if *command* exits with a non-zero status.

`run` *command*

The `run` command is identical to `runwait` except that it does not wait for *command* to complete. *command* is the pathname of the command to be run. `run` will not fail unless it is unable to create a child process to execute the command.

Although they are syntactically indistinguishable, some of the commands available to `run` and `runwait` are

interpreter built-in commands. Interpreter built-ins are used when it is necessary to alter the state of a process within the context of that process. The `doconfig` interpreter built-in commands are similar to the shell special commands and, like these, they do not spawn another process for execution. See `sh(1)`. The initial set of built-in commands is:

```
cd
ulimit
umask
```

RETURN VALUE

`doconfig` returns 0 if the script was interpreted successfully. If a command in the script fails, the interpretation of the script ceases at that point and a positive number is returned; this number indicates which line in the script failed. If a system error occurs, a value of `-1` is returned. When a script fails, the process whose environment was being established should *not* be started.

SEE ALSO

`sh(1)`, `pmadm(1M)`, `sacadm(1M)`.

drand48(3C)**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 int xsubi[3]);
long int lrand48(void);
long int nrand48(unsigned short int xsubi[3]);
long int mrand48(void);
long int jrand48(unsigned short int xsubi[3]);
void srand48(long int seedval);
unsigned short int *seed48(unsigned short int seed16v[3]);
void lcong48(unsigned short int param[7]);□
```

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruent algorithm and 48-bit integer arithmetic.

Functions `drand48()` and `erand48()` return non-negative double-precision floating-point values uniformly distributed over the interval $[0.0, 1.0)$.

`lrand48()` and `nrand48()` return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

Functions `mrnd48()` 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 `mrnd48()` is called. Although it is not recommended practice, constant default initializer values will be supplied automatically if `drand48()`, `lrand48()`, or `mrnd48()` 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 congruent formula:

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \quad n \geq 0$$

The parameter m is 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

$$a = 5DEECE66D_{16} = 273673163155_8$$

$$c = B_{16} = 13_8$$

The value returned by any of the functions `drand48()`, `erand48()`, `lrand48()`, `nrand48()`, `mrnd48()`, 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 `mrnd48()` store the last X_i generated in an internal buffer. X_i must be initialized prior to being invoked. For the functions `erand48()`, `nrand48()`, and `jrand48()`, the calling program must 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, i.e., 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 via `seed48()` when the program is restarted.

The initialization function `lcong48()` allows the user to specify the initial X_i , the multiplier value a and the addend value c . Argument array elements `param[0-2]` specify X_i , `param[3-5]` 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.

The `drand48()`, `lrand48()` and `mrnd48()` interfaces need not be thread-safe.

RETURN VALUE

As described in the **DESCRIPTION** above.

ERRORS

No errors are defined.

SEE ALSO

`rand(3C)`, `stdlib(5)`.

NAME

dup2 - duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>
□
int dup2(int fildes, int fildes2);
```

DESCRIPTION

fildes is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than `OPEN_MAX` (the maximum number of open files). `dup2()` causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred 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.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`dup2()` will fail if one or more of the following are true:

`EBADF`

fildes is not a valid open file descriptor.

`EBADF`

fildes2 is negative or greater than or equal to `OPEN_MAX`.

`EINTR`

A signal was caught during the `dup2()` call.

`EMFILE`

`OPEN_MAX` file descriptors are currently open.

RETURN VALUE

Upon successful completion a non-negative integer, namely, the file descriptor, is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

SEE ALSO

`creat(2)`, `close(2)`, `exec(2)`, `fcntl(2)`, `open(2)`, `pipe(2)`, `lockf(3C)`, `limits(4)`, `unistd(4)`.

NAME

`ecvt`, `fcvt`, `gcvt` - convert floating-point number to string

SYNOPSIS

```
#include <stdlib.h>
char *ecvt(double value, int ndigit, int *decpt, int
*sign);
char *fcvt(double value, int ndigit, int *decpt, int
*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()` 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()` 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()` converts *value* to a null-terminated string (similar to that of the `%g` format of `printf()`) 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()` 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.

These interfaces need not be thread-safe.

RETURN VALUE

The `ecvt()` and `fcvt()` functions return a pointer to a null-terminated string of digits.

The `gcvt()` function returns *buf*.

The values returned by `ecvt()` and `fcvt()` point to a single static data array whose content is overwritten by each call.

ERRORS

No errors are defined.

SEE ALSO

`setlocale(3C)`, `printf(3S)`, `stdlib(5)`.

NAME

`end`, `etext`, `edata` - last locations in program

SYNOPSIS

```
extern etext;
extern edata;
extern 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 address above the program text.

`edata`

The address of `edata` is the first address above the initialized data region.

`end` The address of `end` is the first address above the uninitialized data region.

NOTES

When execution begins, the program break (the first location beyond the data) coincides with `end`, but the program break may be reset by the routines `brk`, `malloc`, the standard input/output library [see [stdio\(3S\)](#)], 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)` [see [brk\(2\)](#)].

SEE ALSO

`cc(1)`, [brk\(2\)](#), [malloc\(3C\)](#), [stdio\(3S\)](#).

NAME

fattach - attach a STREAMS-based file descriptor to an object in the file system name space

SYNOPSIS

```
#include <stropts.h>
int fattach(int fildev, const char *path);
```

DESCRIPTION

The **fattach** routine attaches a STREAMS-based file descriptor to an object in the file system name space, effectively associating a name with *fildev*. *fildev* must be a valid open file descriptor representing a STREAMS file. *path* 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 file until the STREAMS file is detached from the node. *fildev* can be attached to more than one *path*, i.e., a stream can have several names associated with it.

The attributes of the named stream [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 device associated with *fildev*. If any attributes of the named stream are subsequently changed [e.g., **chmod(2)**], the attributes of the underlying object are not affected.

File descriptors referring to the underlying file, opened prior to an **fattach()** call, continue to refer to the underlying file.

RETURN VALUE

If successful, **fattach()** returns 0; otherwise it returns -1 and sets **errno** to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in **intro_prm2(2)** or in **errno(5)**.

Under the following conditions, the function **fattach** fails and sets **errno** to:

EACCES

Search permission is denied for a component of the path prefix, or the process is the owner of *path* but does not have write permissions on the file named by *path*.

EBADF

fildev is not a valid open file descriptor.

ENOENT

A component of *path* does not name an existing file or *path* is an empty string.

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.

EBUSY

path is currently a mount point or has a STREAMS file descriptor attached it.

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.

ELOOP

Too many symbolic links were encountered in translating *path*.

EREMOTE

path is a file in a remotely mounted directory.

The `fattach()` function may fail if:

`EINVAL`

The *fildev* argument does not refer to a STREAMS file.

`ENAMETOOLONG`

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

`EXDEV`

A link to a file on another file system was attempted.

APPLICATION USAGE

The `fattach()` function behaves similarly to the traditional `mount(2)` function in the way a file is temporarily replaced by the root directory of the mounted file system. In the case of `fattach()`, the replaced file need not be a directory and the replacing file is a STREAMS file.

SEE ALSO

`fdetach(1M)`, `fdetach(3C)`, `isastream(3C)`, `stropts(5)`, `streamio(7)`.

fdetach(3C)**NAME**

`fdetach` - detach a name from a STREAMS-based file descriptor

SYNOPSIS

```
#include <stropts.h>
int fdetach(const char *path);
```

DESCRIPTION

The `fdetach()` routine detaches a STREAMS-based file descriptor from a name in the file system. *path* is the path name of the object in the file system name space, which was previously attached [see [fattach\(3C\)](#)]. The user must be the owner of the file or a user with the appropriate privileges. All subsequent operations on *path* will operate on the file system node and not on the STREAMS file. The permissions and status of the node are restored to the state the node was in before the STREAMS file was attached to it.

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()` on the attached file.

RETURN VALUE

If successful, `fdetach()` returns 0; otherwise it returns -1 and sets `errno` to indicate an error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Under the following conditions, the function `fdetach` fails and sets `errno` to:

`EACCES`

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

`EPERM`

The effective user ID is not the owner of *path* or is not a user with appropriate permissions.

`ENOTDIR`

A component of the path prefix is not a directory.

`ENOENT`

path does not exist.

`EINVAL`

path is not attached to a STREAMS file.

`ENAMETOOLONG`

The size of *path* exceeds `PATH_MAX`, or a path name component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

`ELOOP`

Too many symbolic links were encountered in translating *path*.

The `fdetach()` function may fail if:

`ENAMETOOLONG`

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

SEE ALSO

[fdetach\(1M\)](#), [fattach\(3C\)](#), [stropts\(5\)](#), [streamio\(7\)](#).

NAME

`ffs` - find first set bit

SYNOPSIS

```
#include <strings.h>
int ffs(int i);
```

DESCRIPTION

`ffs()` finds first bit set in the argument passed and returns the index of that bit. Bits are numbered starting at 1 from the low order bit.

RETURN VALUE

The `ffs()` function returns the index of the first bit set. If *i* is 0, then `ffs()` returns 0.

ERRORS

No errors are defined.

SEE ALSO

`strings(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

Based on a message's classification component, `fmtmsg()` writes a formatted message to `stderr`, to the console, or to both.

`fmtmsg()` can be used instead of the usual `printf()` interface to display messages to `stderr`. `fmtmsg()`, in conjunction with `gettext()`, provides a simple interface for producing language-independent applications.

A formatted message consists of up to five standard components as defined below. The component, *classification*, 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 detected. 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` (nonrecoverable).

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` application.

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 reason for 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: "TO"

FIX:". 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

There are two environment variables that control the behavior of `fmtmsg()`: `MSGVERB` and `SEV_LEVEL`.

`MSGVERB` tells `fmtmsg()` which message components it is to select when writing messages to `stderr`. The value of `MSGVERB` is a colon-separated list of optional keywords. `MSGVERB` 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 the `MSGVERB` environment variable to see which message components it is to select when generating a message to write to the standard error stream, `stderr`. The values accepted on the initial call are saved for future calls.

`MSGVERB` affects only which components are selected for display to the standard error stream. All message components are included in console messages.

`SEV_LEVEL` defines severity levels and associates print strings with them for use by `fmtmsg()`. The standard severity levels shown 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()` is controlling.

```
0    (no severity is used)
1    HALT
2    ERROR
3    WARNING
4    INFO
```

`SEV_LEVEL` can be set as follows:

```
SEV_LEVEL=[description[:description[:...]]]
export SEV_LEVEL
```

description is a comma-separated list containing three fields:

description=*severity_keyword*, *level*, *printstring*

severity_keyword is a character string that is used as the keyword on the `-s severity` option to the `fmtmsg()`

command. This field is not used by the `fmtmsg()` function.

level 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.

printstring 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 see 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).

EXAMPLES

Example 1:

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

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 the following output:

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

RETURN VALUE

The exit codes for `fmtmsg()` are the following:

`MM_OK` The function succeeded.

`MM_NOTOK`

The function was not successfully executed.

`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.

ERRORS

No errors are defined.

SEE ALSO

`fmtmsg(1)`, `addseverity(3C)`, `gettext(3C)`, `printf(3S)`, `fmtmsg(5)`.

fnmatch(3C)**NAME**

`fnmatch` - match filename or pathname

SYNOPSIS

```
#include <fnmatch.h>

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

DESCRIPTION

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

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` utility is an example of this.

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 pathnames, since it gives no special significance to the slash character. With the `FNM_PATHNAME` flag, `fnmatch()` does match pathnames, but without tilde expansion, parameter expansion, or special treatment for period at the beginning of a filename.

The *flags* argument modifies the interpretation of *pattern* and *string*. It is the bitwise inclusive OR of zero or more of the flags defined in the header `fnmatch.h`. If the `FNM_PATHNAME` flag is set in *flags*, then 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 the `FNM_PATHNAME` flag is not set, the slash character is treated as an ordinary character.

If `FNM_NOESCAPE` is not set in *flags*, 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 `FNM_NOESCAPE` is set, a backslash character will be treated as an ordinary character.

If `FNM_PERIOD` is set in *flags*, then a leading period in *string* will match a period in *pattern*. 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 `FNM_PERIOD` is not set, then no special restrictions are placed on matching a period.

The interpretation of range specifications in *pattern* is influenced by the following flags:

- If `FNM_RANGEOK` is set, a range specification in the format `[m-n-r]` is interpreted as `[m-r]`.
- If `FNM_RANGE_ERR` is set, the range specification `[m-n-r]` returns an error value.
- `FNM_CLRANGE_ERR` initiates the return of an error value for a range specification in the format `[ [=m=] -r ]`. If `FNM_CLRANGE_ERR` is not set, the minus sign `-` is treated as a normal character.

RETURN VALUE

If *string* matches the pattern specified by *pattern*, then `fnmatch()` returns zero. If there is no match, `fnmatch()` returns `FNM_NOMATCH`. If an unknown pattern or flag is specified, `fnmatch()` returns `FNM_BADPAT`. Both error values are defined in the include file `fnmatch.h`. If another error occurs, `fnmatch()` returns another non-zero value.

ERRORS

No errors are defined.

SEE ALSO

[glob\(3C\)](#), [wordexp\(3C\)](#), [fnmatch\(5\)](#).

fpgetround(3C)**NAME**

fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky - IEEE floating-point environment control

SYNOPSIS

```
#include <ieeefp.h>
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 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.

```
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) */
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) */
```

fpgetround() returns the current rounding mode.

fpsetround() sets the rounding mode and returns the previous rounding mode.

fpgetmask() returns the current exception masks.

fpsetmask() sets the exception masks and returns the previous setting.

fpgetsticky() returns the current exception sticky flags.

fpsetsticky() sets (clears) the exception sticky flags and returns the previous setting.

The default environment is set to round to nearest (FP_RN), and all traps are disabled.

Individual bits may be examined using the constants defined in ieeefp.h.

NOTES

fpsetsticky() modifies all sticky flags. fpsetmask() changes all mask bits. fpsetmask() clears the sticky bit corresponding to any exception being enabled.

C requires truncation ("round to zero") for floating point to integral conversions. The current rounding mode has no effect on these conversions.

To recover from a trap caused by a floating point exception and then proceed, one must clear the sticky bit. If the sticky bit is not cleared before the next trap occurs, a wrong exception type may be signaled.

SEE ALSO

[isnan\(3C\)](#).

fsetpos(3C)**NAME**

`fsetpos`, `fgetpos`, `fsetpos64`, `fgetpos64` - reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fsetpos(FILE *stream, const fpos_t *pos);
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos64(FILE *stream, const fpos64_t *pos);
int fgetpos64(FILE *stream, fpos64_t *pos); □
```

DESCRIPTION

`fsetpos()` sets the position of the next input or output operation on the *stream* according to the value of the object pointed to by *pos*. The object pointed to by *pos* must be a value returned by an earlier call to `fgetpos()` on the same stream.

`fsetpos()` clears the end-of-file indicator for the stream and undoes any effects of the `ungetc()` function on the same stream. After `fsetpos()`, the next operation on a file opened for update may be either input or output.

`fsetpos()` stores the current value of the file position indicator for *stream* in the object pointed to by *pos*. The value stored contains information usable by `fsetpos()` for repositioning the stream to its position at the time of the call to `fsetpos()`.

There is no functional difference between `fsetpos()/fgetpos()` and `fsetpos64()/fgetpos64()`, except for the interpretation of `fpos_t/fpos64_t` [see [lfs\(5\)](#)].

If successful, both `fsetpos()` and `fsetpos64()` return zero. Otherwise, they both return nonzero.

SEE ALSO

[lseek\(2\)](#), [fseek\(3S\)](#), [ungetc\(3S\)](#), [lfs\(5\)](#), [stdio\(5\)](#).

NAME

ftime - get date and time

SYNOPSIS

```
#include <sys/types.h>
#include <sys/timeb.h>
ftime(struct timeb *tp);
```

DESCRIPTION

The `ftime()` function fills in a structure pointed to by its argument, as defined by `<sys/timeb.h>`:

```
struct timeb
{
    time_t          time;
    unsigned short millitm;
    short           timezone;
    short           dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

RETURN VALUE

Upon successful completion, the `ftime()` function returns 0. Otherwise -1 is returned.

ERRORS

No errors are defined.

SEE ALSO

[date\(1\)](#), [ctime\(3C\)](#), [gettimeofday\(3C\)](#), [timeb\(5\)](#), [types\(5\)](#).

NAME

`ftw`, `nftw`, `ftw64`, `nftw64` - walk a file tree

SYNOPSIS

```
#include <ftw.h>

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

int nftw(const char *path,
        int (*fn) (const char *, const struct stat *, int, struct FTW *),
        int depth, int flags);

int ftw64(const char *path,
        int (*fn) (const char *, const struct stat64 *, int), int ndirs);

int nftw64(const char *path,
           int (*fn) (const char *, const struct stat64 *, int, struct FTW *),
           int depth, int flags);
```

DESCRIPTION

`ftw()` 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 file, 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 will not be processed.

`FTW_SL`

A symbolic link (but see also `FTW_NS` below).

`FTW_NS`

An object other than a symbolic link on which `stat()` could not successfully be executed. If the object is a symbolic link and `stat()` failed, it is unspecified whether `ftw()` passes `FTW_SL` or `FTW_NS` to the user-supplied function.

If the integer is `FTW_DNR`, descendants of that directory will not be processed. If the integer is `FTW_NS`, the `stat` structure will contain undefined values. An example of an object that would cause `FTW_NS` to be passed to the function pointed to by *fn* would be a file in a directory with read but without execute (search) permission.

The `ftw()` function visits a directory before visiting any of its descendants.

The `ftw()` function uses at most one file descriptor for each level in the tree.

The argument *ndirs* should be in the range of 1 to `OPEN_MAX`.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a non-zero value, or some error, other than `EACCES`, is detected within `ftw()`.

The *ndirs* argument specifies the maximum number of directory streams or file descriptors or both available for use by `ftw()` while traversing the tree. When `ftw()` returns it closes any directory streams and file descriptors it uses not counting any opened by the application-supplied *fn* function.

The function `nftw()` is similar to `ftw()` except that it takes an additional argument, *flags*, which is a bitwise

inclusive-OR of zero or more of the following flags:

`FTW_CHDIR`

If set, `nftw()` will change the current working directory to each directory as it reports files in that directory. If clear, `nftw()` will not change the current working directory.

`FTW_DEPTH`

If set, `nftw()` will report all files in a directory before reporting the directory itself. If clear, `nftw()` will report any directory before reporting the files in that directory.

`FTW_MOUNT`

If set, `nftw()` will only report files in the same file system as *path*. If clear, `nftw()` will report all files encountered during the walk.

`FTW_NONFS`

If set, `nftw()` does not go into directories that have been mounted with NFS. If `nftw()` is started with this option in an NFS directory, it ends without performing any action.

`FTW_PHYS`

If set, `nftw()` performs a physical walk and does not follow symbolic links. If clear, `nftw()` will follow links instead of reporting them, and will not report the same file twice.

The function `nftw()` calls *fn* with four arguments at each file and directory. The first argument is the pathname of the object, the second is a pointer to the `stat` buffer, the third is an integer giving additional information, and the fourth is a `struct FTW` that contains the following members:

```
int base;
int level;
```

`base` is the offset into the pathname of the base name of the object. `level` specifies which levels in the hierarchy have already been traversed, and which are still to visit. The lowest level has the value 0.

The meanings of the third argument values are as follows:

`FTW_F`

The object is a file.

`FTW_D`

The object is a directory.

`FTW_DP`

The object is a directory and subdirectories have already been traversed. (This condition will only occur if the `FTW_DEPTH` flag is included in *flags*.)

`FTW_SL`

The object is a symbolic link. (This condition will only occur 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 will only occur if the `FTW_PHYS` flag is not included in *flags*.)

`FTW_DNR`

The object is a directory that cannot be read. *fn* will not be called for any of its descendants.

`FTW_NS`

`stat` cannot process the object because of lack of appropriate permission. The `stat` buffer passed to *fn* is undefined. `stat` failures for other reasons are considered an error and `nftw()` will return `-1`.

The *depth* argument sets the maximum number of file descriptors that will be used by `nftw()` while traversing the file tree. At most one file descriptor will be used for each directory level.

There is no functional difference between `ftw()/nftw()` and `ftw64()/nftw64()`, except for the interpretation of `stat/stat64` [see [stat\(5\)](#)].

RETURN VALUE

If the tree is exhausted, `ftw()` returns 0. If the function pointed to by *fn* returns a non-zero value, `ftw()` stops its tree traversal and returns whatever value was returned by the function pointed to by *fn*. If `ftw()` detects an error, it returns `-1` and sets `errno` to indicate the error.

If `ftw()` encounters an error other than `EACCES` (see `FTW_DNR` and `FTW_NS` above), it returns `-1` and `errno` is set to indicate the error. The external variable `errno` may 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.

The same applies for the `nftw()` function.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `ftw()` and `nftw()` function will fail if:

`ENAMETOOLONG`

The length of the *path* 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 *path* is not a directory.

The `ftw()` function will fail if:

`EACCES`

Search permission is denied for any component of *path* or read permission is denied for *path*.

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 `ftw()` and `nftw()` function may fail if:

`ELOOP`

Too many symbolic links were encountered in resolving *path*.

`ENAMETOOLONG`

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

The `ftw()` function may fail if:

`EINVAL`

The value of the *ndirs* argument is invalid.

The `nftw()` function may fail if:

`EMFILE`

`OPEN_MAX` file descriptors are currently open in the calling process.

`ENFILE`

Too many files are currently open in the system.

In addition, if the function pointed to by *fn* encounters system errors, `errno` may be set accordingly.

NOTES

Because `ftw()` is recursive, it is possible for it to terminate with a memory fault when applied to file trees which have too many hierarchical levels. `ftw()` uses [malloc\(3C\)](#) to allocate dynamic storage during its operation. If `ftw()` is forcibly terminated, such as by `longjmp()` being executed by *fn* or an interrupt routine, `ftw()` will not have a chance to free that storage, so it will remain permanently allocated. Interrupts can be handled by arranging that *fn* returns a nonzero value at its next invocation.

SEE ALSO

[stat\(2\)](#), [malloc\(3C\)](#), [ftw\(5\)](#), [lfs\(5\)](#), [stat\(5\)](#).

fwide(3C)**NAME**

`fwide` - wide-character 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 zero, the function first attempts to make the stream wide oriented. If *mode* is less than zero, the function first attempts to make the stream byte oriented. (If the orientation of the stream has already been determined, `fwide()` does not change it.) Otherwise, *mode* is zero 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 VALUE

The `fwide()` function returns a value greater than zero if, after the call, the stream has wide orientation, a value less than zero if the stream has byte orientation, or zero if the stream has no orientation.

ERRORS

The following error code description is function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `fwide()` function may fail if:

`EBADF`

The *stream* argument is not a valid stream.

APPLICATION USAGE

A call to `fwide()` with *mode* set to zero can be used to determine the current orientation of a stream.

SEE ALSO

[stdlib\(5\)](#), [wchar\(5\)](#).

getcwd(3C) - grantpt(3C)**getcwd(3C)****NAME**

`getcwd` - get pathname of current working directory

SYNOPSIS

```
#include <unistd.h>
char *getcwd(char *buf, int size);
```

DESCRIPTION

`getcwd()` returns a pointer to the current directory pathname. The value of *size* must be at least one greater than the length of the pathname to be returned.

If *buf* is not `NULL`, the pathname will be stored in the space pointed to by *buf*.

If *buf* is a null pointer, `getcwd()` will obtain *size* bytes of space using `malloc(3C)`. In this case, the pointer returned by `getcwd()` may be used as the argument in a subsequent call to `free()`.

EXAMPLES

Here is a program that 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);
    return(0);
}
```

RETURN VALUE

Returns `NULL` with `errno` set if *size* is not large enough, or if an error occurs in a lower-level function.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`getcwd()` will fail if one or more of the following are true:

`EACCES`

A parent directory cannot be read to get its name.

`EINVAL`

size is equal to 0.

`ERANGE`

size is less than 0 or is greater than 0 and less than the length of the pathname plus 1.

SEE ALSO

`malloc(3C)`. `unistd(4)`.

getdate(3C)**NAME**

getdate, getdate_err - convert date and time

SYNOPSIS

```
#include <time.h>
struct tm *getdate(const char *string);
extern int getdate_err;
```

DESCRIPTION

getdate() converts user-definable date and/or time specifications pointed to by *string* into a `tm` structure. The structure declaration is in the `<time.h>` header file [see also [ctime\(3C\)](#)].

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 `DATMSK`. Each line in the template represents an acceptable date and/or time specification using some of the same field descriptors as the ones used by the `date` command. The first line in the template that matches the input specification is used for interpretation and conversion into the internal time format. If successful, the function `getdate()` returns a pointer to a `tm` structure; otherwise, it returns `NULL` and sets the global variable `getdate_err` to indicate the error.

The following field descriptors are supported [cf. [LC_TIME\(4\)](#)]:

%%	same as %
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	locale's appropriate date and time representation
%C	century number (00-99; leading zeros are permitted but not required)
%d	day of month (01 – 31; the leading 0 is optional)
%e	same as %d
%D	date as %m/%d/%y
%h	abbreviated month name
%H	hour (00 – 23)
%I	hour (01 – 12)
%m	month number (01 – 12)
%M	minute (00 – 59)
%n	same as \n
%p	locale's equivalent of either AM or PM
%r	time as %I:%M:%S %p
%R	time as %H:%M
%S	seconds (00 – 59)
%t	insert a tab
%T	time as %H:%M:%S
%w	weekday number (0 – 6; Sunday = 0)
%x	locale's appropriate date representation
%X	locale's appropriate time representation

- `%y` year with century (00 – 99). 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 as *ccyy* (e.g., 1994)
- `%Z` time zone name or no characters if no time zone exists. If the time zone supplied by `%Z` is not the time zone that `getdate()` expects, an invalid input specification error will result. The `getdate()` function calculates an expected time zone based on information supplied to the function (such as the hour, day, and month).

The match between the template and input specification performed by `getdate()` is case insensitive.

The month and weekday names 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 categories `LC_TIME` and `LC_CTYPE` of `setlocale()`.

Leading 0's are not necessary for the descriptors that allow leading 0's. However, at most two digits are allowed for those descriptors, including leading 0's. Extra white-space in either the template file or in *string* is ignored.

The field descriptors `%c`, `%x`, and `%X` will not be supported if they include unsupported field descriptors.

The following rules are applied for converting the input specification into the internal format:

- If `%Z` is being scanned, then `getdate()` initializes the broken-down time to be the current time in the scanned time zone. Otherwise it initializes the broken-down time based on the current local time as if `localtime()` had been called.
- 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 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.

This interface need not be thread-safe.

RETURN VALUE

On failure `getdate()` returns `NULL` and sets the variable `getdate_err` to indicate the error.

ERRORS

The `getdate()` function will fail in the following cases, setting `getdate_err` to the value shown in the list below. Any changes to `errno` are unspecified.

- 1 The `DATETIME` 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 was encountered while reading the template file.
- 6 `malloc()` could not be executed successfully (not enough memory available).
- 7 There is no line in the template that matches the input.
- 8 Invalid input specification. For example, February 31; or a time is specified that can not be represented in a `time_t` (representing the time in seconds since 00:00:00 UTC, January 1, 1970).

EXAMPLES

Example 1:

The following example shows the possible contents of a template:

```
%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 the %d. %B %Y %H.%M o'clock
```

Example 2:
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 German, the following is valid:
getdate("Freitag den 10. Oktober 1986 10.30 Uhr")

Example 3:
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

Example 4:
The following examples illustrate the above rules. Assume that the current date is Mon Sep 22 12:19:47 EDT 1986 and the LANG environment variable is not set.

Input	Line in Template	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 1:19:49 EDT 1986
January	%B	Thu Jan 1:19:49 EST 1987
December	%B	Mon Dec 1:19:49 EST 1986
Sep Mon	%b %a	Mon Sep 1:19:50 EDT 1986
Jan Fri	%b %a	Fri Jan 2 12:19:50 EST 1987
Dec Mon	%b %a	Mon Dec 1:19:50 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

NOTES

Subsequent calls to `getdate()` alter the contents of `getdate_err`.
`getdate()` makes explicit use of macros described in `ctype(3C)`.

FILES

`/usr/lib/locale/locale/LC_TIME`
 language specific printable files
`/usr/lib/locale/locale/LC_CTYPE`
 code set specific printable files

SEE ALSO

`ctime(3C)`, `ctype(3C)`, `localtime(3C)`, `setlocale(3C)`, `strftime(3C)`, `LC_TIME(4)`,
`environ(5)`, `time(5)`.

getdtablesize(3C)**NAME**

getdtablesize - get descriptor table size

SYNOPSIS

```
#include <unistd.h>
int getdtablesize(void);
```

DESCRIPTION

Each process has a 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.

This interface need not be thread-safe.

RETURN VALUE

The `getdtablesize()` function returns the current maximum size of this table by calling the `getrlimit()` system call.

ERRORS

No errors are defined.

SEE ALSO

`close(2)`, `dup(2)`, `getrlimit(2)`, `open(2)`, `unistd(4)`.

NAME

getenv - return value for environment name

SYNOPSIS

```
#include <stdlib.h>
char *getenv(const char *name);
```

DESCRIPTION

getenv() searches the environment list [see [environ\(5\)](#)] for a string of the form *name=value* and returns a pointer to a string containing the *value* for the specified name. If the specified name cannot be found, a null pointer is returned. The string pointed to must not be modified by the application, but may be overwritten by a subsequent call to getenv() or putenv() but will not be overwritten by a call to any other function.

This interface need not be thread-safe.

RETURN VALUE

Upon successful completion, getenv() returns a pointer to a string containing the *value* for the specified *name*. If the specified name cannot be found a null pointer is returned.

The return value from getenv() may point to static data which may be overwritten by subsequent calls to getenv() or putenv().

ERRORS

No errors are defined.

SEE ALSO

[exec\(2\)](#), [putenv\(3C\)](#), [environ\(5\)](#), [stdlib\(5\)](#).

getgrent(3C)**NAME**

getgrent, setgrent, endgrent, getgrgid, getgrgid_r, getgrnam, getgrnam_r,
fgetgrent - get group file entry

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrent(void);
void setgrent(void);
void endgrent(void);
struct group *getgrgid(gid_t gid);
int getgrgid_r(gid_t gid, struct group *grp, char *buffer,
               □□□ size_t bufsize, struct group **result);
struct group *getgrnam(const char *name);
int getgrnam_r(const char *name, struct group *grp, char *buffer,
               □□□ size_t bufsize, struct group **result);
struct group *fgetgrent(FILE *f);
```

DESCRIPTION

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 first entry in the group database. Thereafter, it returns a pointer to a `group` structure containing the next group structure in the group database, so successive calls may be used to search the entire database.

The `setgrent()` function effectively rewinds the group database to allow repeated searches.

The `endgrent()` function may be called to close the group database when processing is complete.

The `getgrgid()` function searches the group database for an entry with a matching *gid*.

The `getgrnam()` function searches the group database for an entry with a matching *name*.

The `getgrgid_r()` and `getgrnam_r()` functions update the `group` structure pointed to by *grp* and stores a pointer to that structure at the location pointed to by *result*. The structure contains an entry from the group database with a matching *gid* or *name*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the `_SC_GETGR_R_SIZE_MAX` `sysconf()` parameter. A `NULL` pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

The `getgrent()`, `getgrgid()`, `getgrnam()`, `setgrent()`, and `endgrent()` interfaces need not be thread-safe.

`fgetgrent()` returns a pointer to the next group structure in the stream *f*, which matches the format of `/etc/group`.

RETURN VALUE

When first called, `getgrent()` will return a pointer to the first group structure in the group database. Upon subsequent calls it returns the next group structure in the group database. The `getgrent()` function returns a null pointer on end-of-file or an error and `errno` may be set to indicate the error.

Upon successful completion, `getgrgid()` and `getgrnam()` return a pointer to a `group` structure defined in `<grp.h>` with a matching entry if one is found. The `getgrgid()` and `getgrnam()` functions return a null pointer if either the requested entry was not found, or an error occurred. On error, `errno` will be set to indicate the error.

The return value may point to a static area which is overwritten by a subsequent call to `getgrent()`, `getgrgid()` or `getgrnam()`.

If successful, the `getgrgid_r()` and `getgrnam_r()` functions return zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `getgrent()`, `getgrgid()` and `getgrnam()` functions may fail if:

EINTR

A signal was caught during the operation.

EIO An I/O error has occurred.

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 `getgrgid_r()` and `getgrnam_r()` functions may fail if:

ERANGE

Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting `group` structure.

NOTES

All information is contained in a static area, so it must be copied if it is to be saved.

Applications wishing to check for error situations must set `errno` to zero before calling the appropriate function.

FILES

`/etc/group`

SEE ALSO

[getlogin\(3C\)](#), [getpwent\(3C\)](#), [pthread_intro\(3T\)](#), [group\(4\)](#), [limits\(4\)](#), [grp\(5\)](#), [types\(5\)](#).

gethostname(3C-ucb)
BSD Compatibility Package**NAME**

gethostname, sethostname - get/set name of current host

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <unistd.h>
int gethostname(char *name, int namelen);
int sethostname(char *name, int namelen);
```

DESCRIPTION

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

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

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The following error may be returned by these calls:

EFAULT

The *name* or *namelen* parameter gave an invalid address.

EPERM

The caller was not the privileged user. This error only applies to sethostname().

NOTES

Host names are limited to MAXHOSTNAMELEN characters, currently 256 (see the `param.h` header file).

SEE ALSO

[uname\(2\)](#), [gethostid\(3N\)](#), [unistd\(4\)](#).

NAME

getitimer, setitimer - get/set value of interval timer

SYNOPSIS

```
#include <sys/time.h>
int getitimer(int which, struct itimerval *value);
int setitimer(int which, struct itimerval *value,
              struct itimerval *ovalue);
```

DESCRIPTION

The system provides each process with three interval timers, defined in `sys/time.h`. The `getitimer()` call stores the current value of the timer specified by `which` into the structure pointed to by `value`. The `setitimer()` call sets the value of the timer specified by `which` to the value specified in the structure pointed to by `value`, and if `ovalue` is not `NULL`, stores the previous value of the timer in the structure pointed to by `ovalue`.

A timer value is defined by the `itimerval` structure (see [gettimeofday\(3C\)](#) for the definition of `timeval`), which includes the following members:

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

If `it_value` is non-zero, it indicates the time to the next timer expiration. If `it_interval` is non-zero, it specifies a value to be used in reloading `it_value` when the timer expires. Setting `it_value` to zero disables a timer, regardless of the value of `it_interval`. Setting `it_interval` to zero disables a timer after its next expiration (assuming `it_value` is non-zero).

Time values smaller than the resolution of the system clock are rounded up to this resolution.

The three timers are:

ITIMER_REAL

Decrements in real time. A `SIGALRM` signal is delivered when this timer expires.

ITIMER_VIRTUAL

Decrements in process virtual time (user mode). It runs only when the process is executing. A `SIGVTALRM` signal is delivered when it expires.

ITIMER_PROF

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

The interaction between `setitimer()` and any of `alarm()`, `sleep()` or `usleep()` is unspecified.

RESULT

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

Under the following conditions, the functions `getitimer()` and `setitimer()` fail and set `errno` to:

EINVAL

The specified number of seconds is greater than 100,000,000, the number of microseconds is greater than or equal to 1,000,000, or the `which` parameter is unrecognized.

NOTES

The microseconds field should not be equal to or greater than one second.

`setitimer()` is independent of the `alarm` system call.

Do not use `setitimer()` with the `sleep` routine. A `sleep` following a `setitimer()` wipes out knowledge of the user signal handler.

SEE ALSO

`alarm(2)`, `usleep(3)`, `gettimeofday(3C)`, `sleep(3C)`, `sleep(3C-ucb)`, `signal(5)`,
`sys_time(5)`, `time(5)`.

getlogin(3C)**NAME**

getlogin, getlogin_r - get login name

SYNOPSIS

```
#include <unistd.h>
char *getlogin(void);
int getlogin_r(char *name, size_t namesize);
```

DESCRIPTION

The `getlogin()` function returns a pointer to a string giving a user name associated with the calling process, which is the login name associated with the calling process. If `getlogin()` returns a non-null pointer, then that pointer points to the name that the user logged in under, even if there are several login names with the same user ID.

The `getlogin()` interface need not be thread-safe.

The `getlogin_r()` function puts the name associated by the login activity with the control terminal of the current process in the character array pointed to by *name*. The array is *namesize* characters long and should have space for the name and the terminating null character. The maximum size of the login name is `LOGIN_NAME_MAX`.

If `getlogin_r()` is successful, *name* points to the name the user used at login, even if there are several login names with the same user ID.

RETURN VALUE

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 return value may point to static data whose content is overwritten by each call.

If successful, the `getlogin_r()` function returns zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `getlogin()` function may 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.

ENXIO

The calling process has no controlling terminal.

The `getlogin_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.

NOTES

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/utmp`

SEE ALSO

`getgrent(3C)`, `getpwent(3C)`, `cuserid(3S)`, `pthread_intro(3T)`, `unistd(4)`, `utmp(4)`.

getmntent(3C)**NAME**

getmntent, getmntany - get mnttab file entry

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); □
```

DESCRIPTION

getmntent() and getmntany() each fill in the structure pointed to by *mp* with the broken-out fields of a line in the `/etc/mnttab` file. Each line in the file contains a `mnttab` structure, declared in the `sys/mnttab.h` header file:

```
struct mnttab {
    char    *mnt_special;
    char    *mnt_mountp;
    char    *mnt_fstype;
    char    *mnt_mntopts;
    char    *mnt_time;
};
```

The fields have meanings described in `mnttab(4)`.

getmntent() returns a pointer to the next `mnttab` structure in the file; so successive calls can be used to search the entire file. getmntany() searches the file referenced by *fp* until a match is found between a line in the file and *mpref*. *mpref* matches the line if all non-null entries in *mpref* match the corresponding fields in the file. Note that these routines do not open, close, or rewind the file.

RETURN VALUE

If the next entry is successfully read by getmntent() or a match is found with getmntany(), 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:

```
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.
```

NOTES

The members of the `mnttab` structure point to information contained in a static area, so it must be copied if it is to be saved.

FILES

`/etc/mnttab`

SEE ALSO

`mnttab(4)`, `stdio(5)`.

getopt(3C)**NAME**

getopt, optarg, optind, opterr, optopt - get option letter from argument vector

SYNOPSIS

```
#include <unistd.h>
□
□
□
int getopt(int argc, char * const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

DESCRIPTION

getopt() returns the next option letter in *argv* that matches a letter in *optstring*. It supports all the rules of the command syntax standard. Since all new commands are intended to adhere to the command syntax standard, they should use `getopt(1)`, `getopt(3C)`, or `getsubopt(3C)` to parse positional parameters and check for options that are legal for that command.

optstring must contain the option letters the command using getopt() will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which may be separated from it by white space. *optarg* is set to point to the start of the option argument on return from getopt().

getopt() places in *optind* the *argv* index of the next argument to be processed. *optind* is external and is initialized to 1 before the first call to getopt(). Arguments which include spaces must be quoted.

When all options have been processed (i.e., up to the first non-option argument), getopt() returns EOF. The special option -- (two hyphens) may be used to delimit the end of the options; when it is encountered, EOF is returned and -- is skipped. This is useful in delimiting non-option arguments that begin with - (hyphen).

RETURN VALUE

The getopt() function returns the next option character specified on the command line.

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 character not 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

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options a and b, and the option o, which requires an argument:

```
#include <unistd.h>
int
main (int argc, char *argv[])
{
    int c;
    extern char *optarg;
    extern int optind, optopt;
    int aflag, bflag, errflag;
```

```

char *ofile;
char *ifile;
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", optiot);
            errflg++;
    }
}
if (errflg) {
    fprintf(stderr,
        "usage: ...");
    exit (2);
}
for ( ; optind < argc; optind++) {
    if (access (argv[optind], R_OK)) {
        ...
    }
}

```

The following are accepted 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

```

NOTES

The `getopt()` function does not fully check for mandatory arguments. That is, given an option string `a:b` and the input `-a -b`, `getopt()` assumes that `-b` is the mandatory argument to the option `-a` and not that `-a` is missing a mandatory argument.

If a `"?"` (question mark) is required as an option, `opterr` should be set to 0, and the character in `optopt` must be checked. If `optopt` holds the `"?"` character, a question mark was specified as an option. Otherwise

an unrecognized option was given or no argument was given for an option that required it. The error message handling must be done by the program. Although `getopt()` allows such usage, the "?" character should not be used as a valid option.

It is a violation of the command syntax standard for options with arguments to be grouped with other options, as in `cmd -abxxxx file`, where `a` and `b` are options, `o` is an option that requires an argument, and `xxx` 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 is `cmd -ab xxx file`.

The `getopt()` interface need not be thread-safe.

SEE ALSO

[getopts\(1\)](#), [getsubopt\(3C\)](#), [unistd\(4\)](#).

NAME

getpass - read a password

SYNOPSIS

```
#include <unistd.h>
char *getpass(const char *prompt);
```

DESCRIPTION

`getpass()` opens the process controlling terminal, writes the null-terminated string *prompt* to standard error output, disables echoing, reads the string of characters up to the next newline character or `EOF`, restores the terminal state and closes the terminal. A pointer is returned to a null-terminated string of at most 8 characters. If an error occurs, a null pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

NOTES

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

FILES

/dev/tty

SEE ALSO

[unistd\(4\)](#).

getpriority(3C)**NAME**

getpriority, setpriority - get/set program scheduling priority

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

DESCRIPTION

The `getpriority()` function obtains the scheduling priority of a process, process group or user. The `setpriority()` function sets the scheduling priority of a process, process group or user to `prio + NZERO`.

Target processes are specified by the values of the `which` and `who` arguments. The `which` argument may be one of the following values: `PRIO_PROCESS`, `PRIO_PGRP` or `PRIO_USER`, indicating that the `who` argument is to be interpreted as a process ID, a process group ID or an effective user ID, respectively. A 0 value for the `who` argument specifies the current process, process group or user.

The scheduling priority set with `setpriority()` is applied to the process. If the process is multi-threaded, the scheduling priority affects all system scope threads in the process.

If more than one process is specified, `getpriority()` returns value `NZERO` less than the lowest scheduling priority pertaining to any of the specified processes, and `setpriority()` sets the scheduling priorities of all of the specified processes to `prio + NZERO`.

The default scheduling priority is `NZERO`; lower scheduling priorities cause more favourable scheduling. While the range of valid scheduling priorities is `[0, NZERO*2 - 1]`, implementations may enforce more restrictive limits. If `prio + NZERO` is less than the system's lowest supported scheduling priority, `setpriority()` sets the scheduling priority to the lowest supported value; if `prio + NZERO` is greater than the system's highest supported scheduling priority, `setpriority()` sets the scheduling priority to the highest supported value.

Only a process with appropriate privileges can lower its scheduling priority.

The effect of changing the scheduling priority may vary depending on the process-scheduling algorithm in effect.

Because `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 scheduling priority.

RETURN VALUE

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 could be located using the `which` and `who` argument values specified.

EINVAL

The value of the `which` argument was not recognised, or the value of the `who` argument is not a valid process ID, process group ID or user 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 scheduling priority is being changed.

EACCES

A request was made to change the priority to a lower numeric value (that is, to a higher priority) and the current process does not have appropriate privileges.

NOTES

It is not possible for the process executing `setpriority()` to lower any other process down to its current priority, without requiring privileged user privileges.

The `getpriority()` and `setpriority()` functions work with an offset scheduling priority (scheduling priority minus `NZERO`). The scheduling priority is in the range $[0, 2 \cdot \text{NZERO} - 1]$, while the return value for `getpriority()` and the third parameter for `setpriority()` are in the range $[-\text{NZERO}, \text{NZERO} - 1]$.

SEE ALSO

`fork(2)`, `nice(2)`, `nice(3C-ucb)`, `resource(5)`, `time(5)`.

getpw(3C)**NAME**

getpw - get user name

SYNOPSIS

```
#include <stdlib.h>
int getpw(uid_t uid, char *buf);
```

DESCRIPTION

getpw() searches the password file for a user id number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. getpw() returns nonzero if *uid* cannot be found.

This routine is included only for compatibility with prior systems and should not be used; see [getpwent\(3C\)](#) for routines to use instead.

RETURN VALUE

getpw() returns non-zero on error.

FILES

/etc/passwd

SEE ALSO

[getpwent\(3C\)](#), [passwd\(4\)](#), [stdlib\(5\)](#).

getpwent(3C)**NAME**

getpwent, setpwent, endpwent, getpwnam, getpwnam_r, getpwuid, getpwuid_r, fgetpwent - amend password file entry

SYNOPSIS

```
#include <pwd.h>
#include <sys/types.h>
struct passwd *getpwent(void);
void setpwent(void);
void endpwent(void);
struct passwd *getpwnam(const char *name);
int getpwnam_r(const char *nam, struct passwd *pwd, char *buffer,
               □□□ size_t bufsize, struct passwd **result);□
struct passwd *getpwuid(uid_t uid);
int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
               □□□ size_t bufsize, struct passwd **result);□
struct passwd *fgetpwent(FILE *f);□
```

DESCRIPTION

The `getpwent()` function returns a pointer to a structure containing the broken-out fields of an entry in the user database. Each entry in the user database contains a `passwd` structure. When first called, `getpwent()` returns a pointer to a `passwd` structure containing the first entry in the user database. Thereafter, it returns a pointer to a `passwd` structure containing the next entry in the user database. Successive calls can be used to search the entire user database.

If an end-of-file or an error is encountered on reading, `getpwent()` returns a null pointer.

The `setpwent()` function effectively rewinds the user database to allow repeated searches.

The `endpwent()` function may be called to close the user database when processing is complete.

The `getpwnam()` function searches the user database for an entry with a matching *name*.

The `getpwuid()` function searches the user database for an entry with a matching *uid*.

The `getpwnam_r()` and `getpwuid_r()` functions update the `passwd` structure pointed to by *pwd* and stores a pointer to that structure at the location pointed to by *result*. The structure will contain an entry from the user database with a matching *uid* or *name*. Storage referenced by the structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the `_SC_GETPW_R_SIZE_MAX` `sysconf()` parameter. A `NULL` pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Applications wishing to check for error situations should set `errno` to 0 before calling `getpwnam()` or `getpwuid()`. If `getpwnam()` or `getpwuid()` return a null pointer and `errno` is non-zero, an error occurred.

The `getpwent()`, `setpwent()`, `endpwent()`, `getpwnam()`, and `getpwuid()` interfaces need not be thread-safe.

`fgetpwent()` returns a pointer to the next `passwd` structure in the stream *f*, which matches the format of `/etc/passwd`.

RETURN VALUE

`getpwent()` and `fgetpwent()` return a null pointer on EOF or error.

The `getpwnam()` or `getpwuid()` functions return a pointer to a `passwd` structure as defined in `<pwd.h>` with a matching entry if found. 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.

The return value may point to a static area which is overwritten by a subsequent call to `getpwent()`, `getpwnam()` or `getpwuid()`.

If successful, the `getpwnam_r()` or `getpwuid_r()` functions return zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `getpwent()`, `setpwent()`, `endpwent()`, `getpwnam()`, and `getpwuid()` functions may fail if:

`EINTR`

A signal was caught during the operation.

`EIO` An I/O error has occurred.

`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 `getpwnam_r()` and `getpwuid_r()` functions may fail if:

`ERANGE`

Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting `passwd` structure.

NOTES

All information is contained in a static area, so it must be copied if it is to be saved.

Applications wishing to check for error situations must set `errno` to zero before calling the appropriate function.

Three names associated with the current process can be determined:

`getpwuid(geteuid())`

returns the name associated with the effective user ID of the access;

`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

`/etc/passwd`

SEE ALSO

[getgrent\(3C\)](#), [getlogin\(3C\)](#), [pthread_intro\(3T\)](#), [limits\(4\)](#), [passwd\(4\)](#), [pwd\(5\)](#), [types\(5\)](#)

.

getspent(3C)**NAME**

getspent, getsppnam, lckpwwdf, ulckpwwdf, setspent, endspent, fgetspent - amend shadow passwordfile entry

SYNOPSIS

```
#include <shadow.h>
struct spwd *getspent(void);
struct spwd *getsppnam(const char *name);
int lckpwwdf(void);
int ulckpwwdf(void);
void setspent(void);
void endspent(void);
struct spwd *fgetspent(FILE *fp); □
```

DESCRIPTION

The `getspent()` and `getsppnam()` routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the `/etc/shadow` file. Each line in the file contains a "shadow password" structure, declared in the `shadow.h` header file:

```
struct spwd{
    char    *sp_namp;
    char    *sp_pwwdp;
    long    sp_lstchg;
    long    sp_min;
    long    sp_max;
    long    sp_warn;
    long    sp_inact;
    long    sp_expire;
    unsigned long    sp_flag;
};
```

The `getspent()` routine when first called returns a pointer to the first `spwd` structure in the file; thereafter, it returns a pointer to the next `spwd` structure in the file; so successive calls can be used to search the entire file. The `getsppnam()` routine searches from the beginning of the file until a login name matching `name` is found, and returns a pointer to the particular structure in which it was found. The `getspent()` and `getsppnam()` routines populate the `sp_min`, `sp_max`, `sp_lstchg`, `sp_warn`, `sp_inact`, `sp_expire`, or `sp_flag` field with `-1` if the corresponding field in `/etc/shadow` is empty. If end-of-file is reached, or if a reading or format error is encountered in the file, these functions return a null pointer and set `errno` to `EINVAL`.

`/etc/.pwd.lock` is the lock file. It is used to coordinate modification access to the password files `/etc/passwd` and `/etc/shadow`. `lckpwwdf()` and `ulckpwwdf()` are routines that are used to gain modification access to the password files, through the lock file. A process first uses `lckpwwdf()` to lock the lock file, thereby gaining exclusive rights to modify the `/etc/passwd` or `/etc/shadow` password file. Upon completing modifications, a process should release the lock on the lock file via `ulckpwwdf()`. This mechanism prevents simultaneous modification of the password files.

`lckpwwdf()` attempts to lock the file `/etc/.pwd.lock` within 15 seconds. If unsuccessful, e.g., `/etc/.pwd.lock` is already locked, it returns `-1`. If successful, a return code other than `-1` is returned.

`ulckpwwdf()` attempts to unlock the file `/etc/.pwd.lock`. If unsuccessful, e.g., `/etc/.pwd.lock` is already unlocked, it returns `-1`. If successful, it returns `0`.

A call to the `setspent()` routine has the effect of rewinding the shadow password file to allow repeated

searches. The `endspent()` routine may be called to close the shadow password file when processing is complete.

The `fgetspent()` routine returns a pointer to the next `spwd` structure in the stream *fp*, which matches the format of `/etc/shadow`.

RETURN VALUE

`getspent()`, `getspnam()`, `lckpword()`, `ulckpword()`, and `fgetspent()` return a null pointer on end-of-file or error.

NOTES

This routine is for internal use only; compatibility is not guaranteed.

All information is contained in a static area, so it must be copied if it is to be saved.

FILES

`/etc/shadow`
`/etc/passwd`
`/etc/.pwd.lock`

SEE ALSO

`getpwent(3C)`, `putpwent(3C)`, `putspent(3C)`.

getsubopt(3C)

NAME

getsubopt - parse suboptions from a string

SYNOPSIS

```
#include <stdlib.h>

int getsubopt (char **optionp, char * const *tokens, char□
**valuep) ;
```

DESCRIPTION

getsubopt() parses suboptions in a flag argument that was initially parsed by getopt(). These suboptions are separated by commas and may consist of either a single token or a token-value pair separated by an equal sign. Since commas delimit suboptions in the option string, they are not allowed to be part of the suboption or the value of a suboption. A command that uses this syntax is `mount(1M)`, which allows the user to specify mount parameters with the `-o` option as follows:

```
mount -o rw,hard,bg,wsiz=1024 speed:/usr /usr
```

In this example there are four suboptions: `rw`, `hard`, `bg`, and `wsiz`, the last of which has an associated value of 1024.

getsubopt() takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string or `-1` if there was no match. If the option string at *optionp* contains only one suboption, getsubopt() updates *optionp* to point to the null character at the end of the string; otherwise it isolates the suboption by replacing the comma separator with a null character, and updates *optionp* to point to the start of the next suboption. If the suboption has an associated value, getsubopt() updates *valuep* to point to the value's first character. Otherwise it sets *valuep* to `NULL`.

The token vector is organized as a series of pointers to null strings. The end of the token vector is identified by a null pointer.

When getsubopt() returns, if *valuep* is not `NULL`, then the suboption processed included a value. The calling program may use this information to determine if the presence or lack of a value for this suboption is an error.

Additionally, when getsubopt() fails to match the suboption with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed to another program.

EXAMPLE

The following code fragment shows how to process options to the `mount` command using getsubopt().

```
#include <stdlib.h>

char *myopts[] = {
#define READONLY      0
    "ro",
#define READWRITE     1
    "rw",
#define WRITESIZE     2
    "wsiz=",
#define READSIZE      3
    "rsiz=",
    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;
                    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 */
}
...}

```

RETURN VALUE

`getsubopt()` returns `-1` when the token it is scanning is not in the token vector. The variable addressed by *valuep* contains a pointer to the first character of the token that was not recognized rather than a pointer to a value for that token.

The variable addressed by *optionp* points to the next option to be parsed, or a null character if there are no more options.

NOTES

During parsing, commas in the option input string are changed to null characters. White space in tokens or token-value pairs must be protected from the shell by quotes.

SEE ALSO

`getopt(3C)`, `stdlib(5)`.

gettimeofday(3C)**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);
```

DESCRIPTION

gettimeofday() gets and settimeofday() 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.

tp 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.

tzp is an obsolete pointer formerly used to get and set timezone information. *tzp* is now ignored. Timezone information is now handled using the TZ environment variable [see [timezone\(4\)](#)].

Only the privileged user may set the time of day.

RETURN VALUE

A–1 return value indicates that an error occurred and `errno` has been set.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The following error codes may be set in `errno`:

EINVAL

tp specifies an invalid time.

EPERM

A user other than the privileged user attempted to set the time or time zone.

NOTES

The implementation of `settimeofday()` ignores the `tv_usec` field of *tp*. If the time needs to be set with better than one second accuracy, call `settimeofday()` for the seconds and then `adjtime` for finer accuracy.

SEE ALSO

[adjtime\(2\)](#), [ctime\(3C\)](#), [timezone\(4\)](#), [time\(5\)](#).

gettimeofday(3C-ucb)
BSD Compatibility Package**NAME**

gettimeofday, settimeofday - get or set the date and time

SYNOPSIS

```

/usr/ucb/cc [flag ...] file ... -lucb
#include <sys/time.h>
int gettimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;      /* obsolete */
int settimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;      /* obsolete */

```

DESCRIPTION

The system's notion of the current Greenwich time is obtained with the `gettimeofday()` call, and set with the `settimeofday()` call. The current time is expressed in elapsed seconds and microseconds since 00:00 GMT, January 1, 1970 (zero hour). The resolution of the system clock is hardware dependent; the time may be updated continuously, or in ticks.

`tp` 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.

`tzp` is an obsolete pointer formerly used to get and set timezone information. `tzp` is now ignored. Timezone information is now handled using the `TZ` environment variable [see [timezone\(4\)](#)].

Only the privileged user may set the time of day.

RETURN VALUE

A -1 return value indicates an error occurred; in this case an error code is stored in the global variable `errno`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The following error codes may be set in `errno`:

`EINVAL`

`tp` specifies an invalid time.

`EPERM`

A user other than the privileged user attempted to set the time.

NOTES

Time is never correct enough to believe the microsecond values.

`tzp` is ignored.

SEE ALSO

[date\(1\)](#), [adjtime\(2\)](#), [ctime\(3C\)](#), [gettimeofday\(3C\)](#), [timezone\(4\)](#), [time\(5\)](#).

NAME

gettext - retrieve text string from a message file

SYNOPSIS

```
#include <unistd.h>
char *gettext(const char *msgid, const char *dflt_str);
```

DESCRIPTION

gettext() 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 are held 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()` function with the appropriate arguments.

If `gettext()` 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 `gettext()`.

```
<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.

On failure to pass the correct *msgid* or a valid message number to `gettext()` a pointer to the text string "Message not found!!\n" is returned.

EXAMPLES

```
gettext("UX:10", "hello world\n")
gettext("UX:10", "")
```

UX is the name of the file that contains the messages. 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
```

SEE ALSO

[exstr\(1\)](#), [mkmsgs\(1\)](#), [srchtxt\(1\)](#), [fmtmsg\(3C\)](#), [setlocale\(3C\)](#), [unistd\(4\)](#), [environ\(5\)](#).

NAME

getut:
 getutent, getutid, getutline, pututline, setutent, endutent, utmpname - accessutmp
 file entry

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

getutent(), getutid(), getutline(), and pututline() each return a pointer to a structure with 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() reads in the next entry from a utmp-like file. If the file is not already open, the command opens it. If the end of the file is reached, getutent() fails.

getutid() searches forward from the current point in the utmp file until it finds an entry with a ut_type matching id->ut_type if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME, or NEW_TIME. If the type specified in id is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, then getutid() will return a pointer to the first entry whose type is one of these four and whose ut_id field matches id->ut_id. If the end of file is reached without a match, it fails.

getutline() searches forward from the current point in the utmp file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS that also has a ut_line string matching the line->ut_line string. If the end of file is reached without a matching entry being found, the command fails.

pututline() writes out the supplied utmp structure into the utmp file. It uses getutid() to search forward for the proper place. It is expected that normally the user of pututline() will have searched for the proper entry using one of the getut() routines. 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 file. It returns a pointer to the utmp structure.

If the entire file is to be checked, then before starting the search for an entry the file pointer must be reset to the beginning of the file. This reset should be done before each search for a new entry if it is desired that the entire file be examined.

`endutent()` closes the open file.

`utmpname()` allows the user to change the name of the file examined, from `/var/adm/utmp` to any other file. It is most often expected that this other file will be `/var/adm/wtmp`. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. `utmpname()` does not open the file. It just closes the old file if it is currently open and saves the new filename.

RETURN VALUE

A null pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

NOTES

The most current entry is saved in a static structure. If a file is to be accessed repeatedly, the entries must be copied before further accesses are made. On each call to either `getutid()` or `getutline()`, 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, if `getutline()` is 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()` routines, if the user has just modified those contents and passed the pointer back to `pututline()`.

These routines use buffered standard I/O for input, but `pututline()` uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the `utmp` and `wtmp` files.

FILES

`/var/adm/utmp`

`/var/adm/wtmp`

SEE ALSO

`ttyslot(3C)`, `utmp(4)`.

NAME

getutx:
getutxent, getutxid, getutxline, pututxline, setutxent, endutxent, utmpxname,
getutmp, getutmpx, updwtmp, updwtmpx - accessutmpx file entry

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.

getutxent() reads in the next entry from a utmpx-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

getutxid() searches forward from the current point in the utmpx file until it finds an entry with a ut_type matching id->ut_type if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME, or NEW_TIME. If the type specified in id is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, then getutxid() will return a pointer to the first entry whose type is one of these four and whose ut_id field matches id->ut_id. If the end of file is reached without a match, it fails.

The getutxid() or getutxline() may cache data. For this reason, to use getutxline() to search for multiple occurrences, it is necessary to zero out the static data after each success, or getutxline() could just return a pointer to the same utmpx structure over and over again.

There is one exception to the rule about removing 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 user accounting database) will not modify the static structure returned by getutxent(), getutxid() or getutxline(), if the application has just modified this structure and passed the pointer back to pututxline().

For all entries that match a request, the ut_type member indicates the type of the entry. Other members of the entry will contain meaningful data based on the value of the ut_type member as follows:

ut_type member	Other members with meaningful data
EMPTY	No others
BOOT_TIME	ut_tv
OLD_TIME	ut_tv
NEW_TIME	ut_tv
USER_PROCESS	ut_id, ut_user (login name of the user), ut_line, ut_pid, ut_tv
INIT_PROCESS	ut_id, ut_pid, ut_tv

LOGIN_PROCESS	ut_id, ut_user (implementation-specific name of the login process), ut_pid, ut_tv
DEAD_PROCESS	ut_id, ut_pid, ut_tv

`getutxline()` searches forward from the current point in the `utmpx` file until it finds an entry of the type `LOGIN_PROCESS` or `USER_PROCESS` which also has a `ut_line` string matching the `line->ut_line` string. If the end of file is reached without a match, it fails.

`pututxline()` writes out the supplied `utmpx` structure into the `utmpx` file. 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 file. It returns a pointer to the `utmpx` structure.

`setutxent()` resets the input stream to the beginning of the file. This should be done before searching the entire file for a new entry.

`endutxent()` closes the open file.

The `getutxent()`, `getutxid()`, `getutxline()`, `pututxline()`, `setutxent()`, and `endutxent()` interfaces need not be thread-safe.

`utmpxname()` allows the user to change the name of the file examined, from `/var/adm/utmpx` to any other filename. In most cases, this other file will be `/var/adm/wtmpx`. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. `utmpxname()` does not open the file. It just closes the old file if it is currently open and saves the new filename. The new filename must end with the `x` character to allow the name of the corresponding `utmp` file to be easily obtainable. Otherwise an error code of 1 is returned.

`getutmp()` copies the information stored in the fields of the `utmpx` structure to the corresponding fields of the `utmp` structure. If the information in any field of `utmpx` does not fit in the corresponding `utmp` field, the data is truncated.

`getutmpx()` copies the information stored in the fields of the `utmp` structure to the corresponding fields of the `utmpx` structure.

`updwtmp()` checks the existence of `wfile` and its parallel file, whose name is obtained by appending an `x` to `wfile`. If only one of them exists, the second one is created and initialized to reflect the state of the existing file. `utmp` is written to `wfile` and the corresponding `utmpx` structure is written to the parallel file.

`updwtmpx()` checks the existence of `wfilex` and its parallel file, whose name is obtained by truncating the final `x` from `wfilex`. If only one of them exists, the second one is created and initialized to reflect the state of the existing file. `utmpx` is written to `wfilex`, and the corresponding `utmp` structure is written to the parallel file.

RETURN VALUE

Upon successful completion, `getutxent()`, `getutxid()` and `getutxline()` 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.

ERRORS

The following error code description is function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

No errors are defined for the `endutxent()`, `getutxent()`, `getutxid()`, `getutxline()` and `setutxent()` functions.

The `pututxline()` function may fail if:

EPERM The process does not have appropriate privileges.

NOTES

The most current entry is saved in a static structure. Before reaccessing the file, this entry must be copied. 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 identical entries, it would be necessary to zero out the static area after each success, or `getutxline()` would just return the same structure over and over again. There is one exception to this rule: the implicit read done by `pututxline()` (if the correct place in the file has not yet been reached) 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()`.

These routines 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.

FILES

```
/var/adm/utmp
/var/adm/utmpx
/var/adm/wtmp
/var/adm/wtmpx
```

SEE ALSO

`ttyslot(3C)`, `utmp(4)`, `utmpx(4)`.

getvfsent(3C)**NAME**

getvfsent, getvfssfile, getvfssspec, getvfssany - getvfstab file entry

SYNOPSIS

```
#include <stdio.h>
#include <sys/vfstab.h>
int getvfsent(FILE *fp, struct vfstab *vp);
int getvfssfile(FILE *fp, struct vfstab *vp, char *file);
int getvfssspec(FILE *fp, struct vfstab *vp, char *spec);
int getvfssany(FILE *fp, struct vfstab *vp, struct vfstab *vref);
```

DESCRIPTION

getvfsent(), getvfssfile(), getvfssspec(), and getvfssany() 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 file:

```
char    *vfs_fsckdev;
char    *vfs_mountp;
char    *vfs_fstype;
char    *vfs_fsckpass;
char    *vfs_automnt;
char    *vfs_mntopts;
```

The fields have meanings described in [vfstab\(4\)](#).

getvfsent() returns a pointer to the next `vfstab` structure in the file; so successive calls can be used to search the entire file. getvfssfile() 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. getvfssspec() 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. *spec* 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. getvfssany() searches the file referenced by *fp* until a match is found between a line in the file and *vref*. *vref* matches the line if all non-null entries in *vref* match the corresponding fields in the file.

Note that these routines do not open, close, or rewind the file.

RETURN VALUE

If the next entry is successfully read by getvfsent() or a match is found with getvfssfile(), getvfssspec(), or getvfssany(), 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.

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.

FILES

`/etc/vfstab`

SEE ALSO

`stdio(5)`.

NAME

getwd - get current working directory pathname

SYNOPSIS

```
#include <sys/param.h>
#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 $(\text{MAXPATHLEN} + 1)$ including the null byte, `getwd()` fails and returns a null pointer.

RETURN VALUE

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.

SEE ALSO

[getcwd\(3C\)](#), [unistd\(4\)](#).

NAME

`glob`, `globfree` - generate pathnames matching a pattern

SYNOPSIS

```
#include <glob.h>
int glob(const char *pattern, int flags, int (*errfunc)()
[] [] (const char *epath, int errno), glob_t *pglob);
void globfree(glob_t *pglob);
```

DESCRIPTION

The `glob()` function is a pathname generator that implements the rules of the pattern matching notation, with optional support for patterns used for filename expansion.

The structure type `glob_t` is defined in the header `glob.h` and includes at least the following members:

Member type	Member name	Description
<code>size_t</code>	<code>gl_pathc</code>	Count of paths matched by <i>pattern</i>
<code>char **</code>	<code>gl_pathv</code>	Pointer to a list of matched pathnames
<code>size_t</code>	<code>gl_offs</code>	Slots to reserve at the beginning of <code>gl_pathv</code>

The argument *pattern* is a pointer to a pathname pattern to be expanded. The `glob()` function matches all accessible pathnames against this pattern and develops a list of all pathnames that match. In order to have access to a pathname, `glob()` requires search permission on every component of a path except the last, and read permission on each directory of any filename component of pattern that contains any of the following special characters: `*`, `?`, `[`.

The `glob()` function stores the number of matched pathnames into `pglob->gl_pathc` and a pointer to a list of pointers to pathnames into `pglob->gl_pathv`. The pathnames are in sort order as defined by the current setting of the `LC_COLLATE` category. The first pointer after the last pathname is a null pointer. If the pattern does not match any pathnames, the returned number of matched paths is set to zero, and the contents of `pglob->gl_pathv` are implementation-dependent.

It is the callers 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()`.

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 pathnames 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` pathname pointers, followed by a null pointer.

`GLOB_ERR`

Causes `glob()` to return when it encounters a directory that it cannot open or read. Ordinarily, `glob()` continues to find matches.

`GLOB_MARK`

Each pathname that is a directory that matches pattern has a slash appended.

`GLOB_NOCHECK`

Support patterns used for filename expansion. If *pattern* does not match any pathname, then `glob()` returns a list consisting of only *pattern*, and the number of matched pathnames is 1.

`GLOB_NOESCAPE`

Disable backslash escaping.

`GLOB_NOSORT`

Ordinarily, `glob()` sorts the matching pathnames according to the current setting of the `LC_COLLATE` category. When this flag is used the order of pathnames returned is unspecified.

The `GLOB_APPEND` flag can be used to append a new set of pathnames 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:
 - Zero or more null pointers, as specified by `GLOB_DOOFFS` and *pglob*->*gl_offs*.
 - Pointers to the pathnames that were in the *pglob*->*gl_pathv* list before the call, in the same order as before.
 - Pointers to the new pathnames generated by the second call, in the specified order.
4. The count returned in *pglob*->*gl_pathc* will be the total number of pathnames 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.

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:

- The *epath* argument is a pointer to the path that failed.
- The *errno* argument is the value from the failure, as set by the `opendir()`, `readdir()` or `stat()` functions. (Other values may be used to report other errors not explicitly documented for those functions).

The following constants are defined as error values for `glob()`:

`GLOB_ABORTED`

The scan was stopped because `GLOB_ERR` was set or `(*errfunc())` returned non-zero.

`GLOB_BADPAT`

An invalid pattern was specified.

`GLOB_NOMATCH`

The pattern does not match any existing pathname, and `GLOB_NOCHECK` was not set in *flags*.

`GLOB_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 zero, the error is ignored.

This function is not provided for the purpose of enabling utilities to perform pathname 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 pathname expansion on strings obtained from other sources, such as pattern typed by a user or read from a file.

If a utility needs to see if a pathname matches a given pattern, it can use `fnmatch()`.

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 zero, *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 pathname if wildcards are specified, but wants to treat the pattern as just a string otherwise. The `sh` utility might use this for

option-arguments, for example.

The new pathnames generated by a subsequent call with `GLOB_APPEND` are not sorted together with the previous pathnames. This mirrors the way that the shell handles pathname expansion when multiple expansions are done on a command line.

Applications that need tilde and parameter expansion should use the `wordexp()` function.

RETURN VALUE

On successful completion, `glob()` returns zero. The argument `pglob->gl_pathc` returns the number of matched pathnames and the argument `pglob->gl_pathv` contains a pointer to a null-terminated list of matched and sorted pathnames. However, if `pglob->gl_pathc` is zero, the content of `pglob->gl_pathv` is undefined.

The `globfree()` function returns no value.

If `glob()` terminates due to an error, it returns one of the non-zero constants defined in `glob.h`. The arguments `pglob->gl_pathc` and `pglob->gl_pathv` are still set as defined.

EXAMPLES

One use of the `GLOB_DOOFFS` flag is by application that built an argument list for use with the `execv()`, `execve()` or `execvp()` functions. 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_pathvv[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 ("*.c", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
```

SEE ALSO

[execv\(2\)](#), [stat\(2\)](#), [fnmatch\(3C\)](#), [opendir\(3C\)](#), [readdir\(3C\)](#), [wordexp\(3C\)](#), [glob\(5\)](#).

grantpt(3C)**NAME**

grantpt - grant access to the slave pseudo-terminal device

SYNOPSIS

```
#include <stdlib.h>
int grantpt(int fildev);
```

DESCRIPTION

The function `grantpt()` changes the mode and ownership of the slave pseudo-terminal device associated with its master pseudo-terminal counter part. *fildev* is the file descriptor returned from a successful open of the master pseudo-terminal device. A `setuid` root program [see `setuid(2)`] is invoked to change the permissions. The user ID of the slave is set to the real owner 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, writable by the owner and writable by the group.

The `grantpt()` function may also fail if the application has installed a signal handler to catch `SIGCHLD` signals.

RETURN VALUE

Upon successful completion, the function `grantpt()` returns 0; otherwise it returns -1. Failure could occur if *fildev* is not an open file descriptor, if *fildev* is not associated with a master pseudo-terminal device, or if the corresponding slave device could not be accessed.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

The `grantpt()` function may fail if:

`EBADF`

The *fildev* argument is not a valid open file descriptor.

`EINVAL`

The *fildev* argument is not associated with a master pseudo-terminal device.

`EACCES`

The corresponding slave pseudo-terminal device could not be accessed.

SEE ALSO

`open(2)`, `setuid(2)`, `ptsname(3C)`, `unlockpt(3C)`, `stdlib(5)`.

hrtalarm(3C) - lsearch(3C)**hrtalarm(3C)****NAME**

hrtalarm - high resolution timer alarm

SYNOPSIS

```
#include <sys/hrtcntl.h>
int hrtalarm(hrtcmd_t *cmdp, int cmds);
```

DESCRIPTION

`hrtalarm()` provides an interface for the alarm function of high resolution or interval timers. *cmds* determines the number of commands for the system call.

The following `hrtcmd` structure is defined in `<sys/hrtcntl.h>` and is used by `hrtalarm()`:

```
typedef struct hrtcmd {
    int          hrtc_cmd;      /* A timer command          */
    int          hrtc_clk;      /* Which clock to use       */
    hrttime_t    hrtc_int;      /* A time interval          */
    hrttime_t    hrtc_tod;      /* A time of day            */
    int          hrtc_flags;     /* Various flags            */
    int          hrtc_error;     /* An error code            */
    ecb_t        hrtc_ecb;      /* An event control block   */
} hrtcmd_t;
```

`hrtc_cmd` supports the following command types:

`HRT_BSD`

Starts a timer and sends an alarm signal when the time elapses.

`HRT_RBSD`

Sends an alarm signal again when the time elapses.

`HRT_BSD_REP`

Sends an alarm signal again when the time elapses depending on the setting in the `hrtc_tod` field.

`HRT_BSD_PEND`

Calculates the remaining time before the alarm will be issued.

`HRT_BSD_CANCEL`

Removes an alarm from the queue.

`hrtc_clk` determines the clock type and signal issued with the `HRT_BSD` and `HRT_RBSD` commands:

`CLK_STD`

Standard real time clock. The `SIGALRM` signal is sent to the process when the set time elapses.

`CLK_USERVIRT`

Clock measuring process virtual time (user mode). The `SIGVTALRM` signal is sent to the process when the set time elapses.

`CLK_PROCVIRT`

Clock measuring process virtual time and when the system is running on behalf of the process (user and system mode). The `SIGPROF` signal is sent to the process when the set time elapses.

`hrtc_int` determines the time interval, the time resolution and the fraction of the second.

RETURN VALUE

`hrtalarm()` returns the number of successfully executed calls. For each executed command `hrtc_error` contains the error code or 0.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`EINVAL`

cmds is not positive.

`EINVAL`

Unknown timer command.

`EINVAL`

Illegal `hrtc_clk` value.

`EFAULT`

cmdp points outside the allocated address space of the process.

`ENOPKG`

The timer command is not supported.

ERANGE

Value of `hrtc_int` is less 0 or the interval is too small.

EAGAIN

All timer structures are currently used.

NOTES

The `hrtsleep()` and `hrtcanceled()` functions are not supported and simply return the error `ENOPKG`.

SEE ALSO

`signal(2)`, `getitimer(3C)`, `hrtcntl(3C)`.

hrtcntl(3C)**NAME**

hrtcntl - high resolution timer control

SYNOPSIS

```
#include <sys/hrtcntl.h>
int hrtcntl(int cmd, int clk, interval_t *intp,
hrtime_t *hrtp);
```

DESCRIPTION

hrtcntl() provides an interface for high resolution or interval timers.

cmd determines the command:

HRT_GETRES

Gets the resolution of a clock. If the call succeeds, the timer resolution is returned.

HRT_TOFD

Gets the value of time since 00:00:00 GMT, January 1, 1970. The field **hrt_res** determines the timer resolution. When called, *hrtp* contains the time value.

HRT_STARTIT

Start the time for an activity. When called, *intp* contains the current values for the specified clock type.

HRT_GETIT

Returns the elapsed interval since the **HRT_STARTIT** command was called. When called, *hrtp* contains the elapsed time.

clk determines the clock type:

CLK_STD

Standard real time clock.

CLK_USERVIRT

Clock measuring process virtual time (user mode).

CLK_PROCVIRT

Clock measuring process virtual time and when the system is running on behalf of the process (user and system mode).

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Under the following conditions, the function **hrtcntl()** fails and sets **errno** to:

EINVAL

Illegal value for *cmd* or *clk*.

EFAULT

hrtp or *intp* points outside the allocated address space of the process.

ERANGE

Illegal timer resolution.

NOTES

The **hrtsleep()** and **hrtcancel()** functions are not supported and simply return the error **ENOPKG**.

SEE ALSO

`getitimer(3C)`, `hrtalarm(3C)`.

hsearch(3C)

NAME

hsearch, hcreate, hdestroy - manage hash search tables

SYNOPSIS

```
#include <search.h>
□
□
□
ENTRY *hsearch(ENTRY item, ACTION action);
int hcreate(size_t nel);
void hdestroy(void);□
```

DESCRIPTION

`hsearch()` is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry was found. The comparison function used by `hsearch()` is `strcmp` [see [string\(3C\)](#)]. `item` is a structure of type `ENTRY`, defined in the `search.h` header file, containing two pointers: `item.key` points to the comparison key (type `char*`), and `item.data` (`void*`) points to any other data to be associated with that key. Pointers to types other than `void` should be cast to `pointer-to-void`. `action` 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()` turns 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.

`hcreate()` allocates sufficient storage for the table, and must be called before `hsearch()`. `nel` is an estimate of the maximum possible number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

`hdestroy()` destroys the search table. It may be followed by another call to `hcreate()`.

EXAMPLES

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

```
#include <stdio.h>
#include <search.h>
#include <string.h>
#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)
    }
}
return 0;
}

```

RETURN VALUE

`hsearch()` returns a null pointer if either the action is `FIND` and the item could not be found or the action is `ENTER` and the table is full.

`hcreate()` returns zero if it cannot allocate sufficient space for the table.

ERRORS

The function fails if:

`ENOMEM`

Insufficient storage space is available.

NOTES

`hsearch()` and `hcreate()` use `malloc(3C)` to allocate space.

Only one hash search table may be active at any given time.

SEE ALSO

`bsearch(3C)`, `lsearch(3C)`, `malloc(3C)`, `string(3C)`, `tsearch(3C)`, `malloc(3X)`, `search(5)`.

NAME

`iconv` - code conversion function

SYNOPSIS

```
#include <iconv.h>

size_t iconv(iconv_t cd, const char **inbuf, size_t *inbytesleft,
             char **outbuf, size_t *outbytesleft);
```

DESCRIPTION

The `iconv()` function converts the sequence of characters from one codeset, in the array specified by *inbuf*, into a sequence of corresponding characters in another codeset, in the array specified by *outbuf*. The codesets 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-depend 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 codeset, 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 codeset, `iconv()` performs an implementation-defined conversion on this character.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `iconv()` function will fail if:

`EILSEQ`

Input conversion stopped due to an input byte that does not belong to the input codeset.

`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.

EBAD The *cd* argument is not a valid open conversion descriptor.

RETURN VALUE

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 zero. 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.

SEE ALSO

`iconv_close(3C)`, `iconv_open(3C)`, `iconv(5)`.

iconv_close(3C)**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()` function.

ERRORS

The `iconv_close()` function will fail if:

`EBADF`

The conversion descriptor is invalid.

RETURN VALUE

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

SEE ALSO

[iconv\(3C\)](#), [iconv_open\(3C\)](#), [iconv\(5\)](#).

iconv_open(3C)**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()` function.

Settings of *fromcode* and *tocode* and their permitted combinations are implementation-dependent.

A conversion descriptor remains valid in a process until that process closes it.

If a file descriptor is used to implement conversion descriptors, the `FD_CLOEXEC` flag will be set.

`iconv_open()` uses `malloc()` to allocate space for internal buffer areas. `iconv_open()` function will 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.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `iconv_open()` function will fail if:

EMFILE

OPEN_MAX file 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.

RETURN VALUE

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.

SEE ALSO

[iconv\(3C\)](#), [iconv_close\(3C\)](#), [iconv\(5\)](#).

NAME

`index`, `rindex` - string operations

SYNOPSIS

```
#include <strings.h>
char *index(const char *s, int c);
char *rindex(const char *s, int c);
```

DESCRIPTION

These functions operate on NULL-terminated strings. They do not check for overflow of any receiving string. `index` and `rindex` returns a pointer to the first (last) occurrence of character `c` in string `s`, or a NULL pointer if `c` does not occur in the string. The NULL character terminating a string is considered to be part of the string.

NOTES

For user convenience, these functions are declared in the optional `<strings.h>` header file.

On the Sun processor, as well as on many other machines, 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 have a pointer that points to an explicit NULL string. On some implementations of the C language on some machines, a NULL pointer, if dereferenced, would yield a NULL string; this highly non-portable trick was used in some programs. 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.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

SEE ALSO

`bstring(3C)`, `malloc(3C)`, `string(3C)`, `strings(5)`.

initgroups(3C)**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

`initgroups()` reads the group file, using `getgrent()`, to get the group membership for the user specified by *name* and then initializes the supplementary group access list of the calling process using `setgroups()`. The *basegid* group id is also included in the supplementary group access list. This is typically the real group id from the password file.

While scanning the group file, if the number of groups, including the *basegid* entry, exceeds `NGROUPS_MAX`, subsequent group entries are ignored.

ERRORS

`initgroups()` will fail and not change the supplementary group access list if:

`EPERM`

The effective user id is not superuser.

RETURN VALUE

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

SEE ALSO

`setgroups(2)`, `getgrent(3C)`, `grp(5)`, `types(5)`.

insque(3C)**NAME**

`insque`, `remque` - insert or remove an element in a queue

SYNOPSIS

```
#include <search.h>
void insque(void *element, void *pred);
void remque(void *element);
```

DESCRIPTION

The `insque()` and `remque()` functions manipulate queues built from doubly-linked lists. The queue can be either circular or linear. An application using `insque()` or `remque()` must define a structure in which the first two members of the structure are pointers to the same type of structure, and any further members are application-specific. The first member of the structure is a forward pointer to the next entry in the queue. The second member is a backward pointer to the previous entry in the queue. If the queue is linear, the queue is terminated with null pointers. The names of the structure and of the pointer members are not subject to any special restriction.

The `insque()` function inserts the element pointed to by *element* into a queue immediately after the element pointed to by *pred*.

The `remque()` function removes the element pointed to by *element* from a queue.

If the queue is to be used as a linear list, invoking `insque(&element, NULL)`, where *element* is the initial element of the queue, will initialize the forward and backward pointers of *element* to null pointers.

If the queue is to be used as a circular list, the application must initialize the forward pointer and the backward pointer of the initial element of the queue to the element's own address.

RETURN VALUE

The `insque()` and `remque()` functions do not return a value.

APPLICATION USAGE

The historical implementations of these functions described the arguments as being of type `struct gelem *` rather than as being of type `void *` as defined here. In those implementations, `struct gelem` was commonly defined in `<search.h>` as:

```
struct gelem {
    struct gelem *q_forw;
    struct gelem *q_back;
};
```

Applications using these functions, however, were never able to use this structure directly since it provided no room for the actual data contained in the elements. Most applications defined structures that contained the two pointers as the initial elements and also provided space for, or pointers to, the object's data. Applications that used these functions to update more than one type of table also had the problem of specifying two or more different structures with the same name, if they literally used `struct gelem` as specified.

As described here, the implementations were actually expecting a structure type where the first two members were forward and backward pointers to structures. With C compilers that didn't provide function prototypes, applications used structures as specified in the **DESCRIPTION** above and the compiler did what the application expected.

If this method had been carried forward with an ISO C compiler and the historical function prototype, most applications would have to be modified to cast pointers to the structures actually used to be pointers to `struct gelem` to avoid compilation warnings. By specifying `void *` as the argument type, applications won't need to change (unless they specifically referenced `struct gelem` and depended on it being defined

in <search.h>).

SEE ALSO

[search\(5\)](#).

isastream(3C)**NAME**

`isastream` - test a file descriptor

SYNOPSIS

```
#include <stropts.h>
int isastream(int fildes);
```

DESCRIPTION

The function `isastream()` determines if a file descriptor represents a STREAMS file. *fildes* refers to an open file.

RETURN VALUE

If successful, `isastream()` returns 1 if *fildes* represents a STREAMS file, and 0 if not. On failure, `isastream()` returns -1 with `errno` set to indicate an error.

ERRORS

The following error code description is function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Under the following conditions, `isastream()` fails and sets `errno` to:

EBADF

fildes is not a valid open file.

SEE ALSO

[stropts\(5\)](#), [streamio\(7\)](#).

NAME

isnan, 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);
#include <math.h>
int isnan(double dsrc); □
```

DESCRIPTION

isnan(), isnand(), and isnanf() return 1 if the argument *dsrc* or *fsrc* is a NaN; otherwise they return 0. The functionality of isnan() is identical to that of isnand().

isnanf() is implemented as a macro included in the <ieeefp.h> header file.

fpclass() returns the class the *dsrc* belongs to. The 10 possible classes are as follows:

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

finite() returns 1 if the argument *dsrc* is neither infinity nor NaN; otherwise it returns 0.

unordered() returns 1 if one of its two arguments is unordered with respect to the other argument. This is equivalent to reporting that one of the arguments is NaN. If neither of the arguments is NaN, 0 is returned.

ERRORS

No errors are defined.

SEE ALSO

fpgetround(3C), math(5).

NAME

killpg - send signal to a process group

SYNOPSIS

```
#include <signal.h>
int killpg(pid_t pgrp, int sig);
```

DESCRIPTION

killpg() sends the signal *sig* to the process group *pgrp*. See [sigvec\(3\)](#) for a list of signals.

If *pgrp* is greater than 1, killpg(*pgrp*, *sig*) is equivalent to kill(-*pgrp*, *sig*). If *pgrp* is less than or equal to 1, the behavior of killpg() is undefined.

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 VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

killpg() will fail and no signal will be sent if any of the following occur:

EINVAL

sig is not a valid signal number.

ESRCH

No processes were found in the specified process group.

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.

SEE ALSO

[kill\(2\)](#), [setpgrp\(2\)](#), [sigaction\(2\)](#), [sigvec\(3\)](#), [signal\(5\)](#).

NAME

l3tol, ltol3 - convert numbers between types

SYNOPSIS

```
#include <stdlib.h>
□
□
□
void l3tol(long *lp, const char *cp, int n);
void ltol3(char *cp, const long *lp, int n);□
```

DESCRIPTION

l3tol() converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

ltol3() performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

NOTES

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

SEE ALSO

fs(4), [stdlib\(5\)](#).

localeconv(3C)**NAME**

localeconv - get numeric formatting information

SYNOPSIS

```
#include <locale.h>
struct lconv *localeconv(void);
```

DESCRIPTION

localeconv() 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 */
/* CHAR_MAX */
```

The members of the structure with type char * are strings, any of which (except decimal_point) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type char are nonnegative numbers, any of which can be CHAR_MAX (defined in the limits.h header file) 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 in which each element is taken as an integer that indicates the number of digits that comprise the current group in a formatted non-monetary quantity. The elements of 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 value is the number of digits that comprise the current the next group of digits to the left of

the current group.

char `*int_curr_symbol`
The international currency symbol. The operand is a four-character string. The first three characters contain the international currency symbol in accordance with ISO 4217:1987. The fourth character 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. In the ISO-C standard, this component is limited to one byte. A multibyte operand will produce an undefined result.

char `*mon_thousands_sep`
The separator for groups of digits to the left of the decimal point in formatted monetary quantities. In the ISO-C standard, this component is limited to one byte. A multibyte operand will produce an undefined result.

char `*mon_grouping`
Size of all groups of digits in formatted monetary quantities. The operand is a series of integers separated by semicolons. Each integer indicates the number of digits in each group. The first number indicates the size of the group directly preceding the decimal separator; the following numbers define the preceding groups. If the last number is not `-1`, the preceding group (if there is one) is always used for the remaining digits. If the last number is `-1`, no further grouping is performed.

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 to be displayed in an internationally formatted monetary quantity where `int_curr_symbol` is used.

char `frac_digits`
The number of fractional digits to be displayed in a formatted monetary quantity where `currency_symbol` is used.

char `p_cs_precedes`
Set to 1 or 0 if the `currency_symbol` or `int_curr_symbol` respectively precedes or succeeds the value for a nonnegative 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 nonnegative 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`
If this component has the value 1, the `currency_symbol` or `int_curr_symbol` precedes the value for a negative formatted monetary quantity. Otherwise this component is set to 0.

char `n_sep_by_space`
Set to 1 if the `currency_symbol` or `int_curr_symbol` is separated by a space from the value for a nonnegative formatted monetary quantity. Otherwise this component is set to 0.

char `p_sign_posn`
Nonnegative formatted monetary quantity. The values of `p_sign_posn` and `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`.

`char n_sign_posn`
Set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity.

RETURN VALUE

`localeconv()` 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

The following table illustrates the rules used by four countries to format monetary quantities.

Country	Positive format	Negative format	International format
Italy	L.1.234	-L.1.234	ITL.1.234
Netherlands	F 1.234,56	F -1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56-	NOK 1.234,56
Switzerland	SFr _s .1,234.56	SFr _s .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:

	Italy	Netherlands	Norway	Switzerland
<code>int_curr_symbol</code>	"ITL."	"NLG "	"NOK "	"CHF "
<code>currency_symbol</code>	"L."	"F"	"kr"	"SFr _s ."
<code>mon_decimal_point</code>	" "	","	","	"."
<code>mon_thousands_sep</code>	"."	"."	"."	","
<code>mon_grouping</code>	"\3"	"\3"	"\3"	"\3"
<code>positive_sign</code>	" "	" "	" "	" "
<code>negative_sign</code>	"_"	"_"	"_"	"C"
<code>int_frac_digits</code>	0	2	2	2
<code>frac_digits</code>	0	2	2	2
<code>p_cs_precedes</code>	1	1	1	1
<code>p_sep_by_space</code>	0	1	0	0
<code>n_cs_precedes</code>	1	1	1	1
<code>n_sep_by_space</code>	0	1	0	0
<code>p_sign_posn</code>	1	1	1	1
<code>n_sign_posn</code>	1	4	2	2

FILES

`/usr/lib/locale/locale/LC_MONETARY`
LC_MONETARY database for *locale*
`/usr/lib/locale/locale/LC_NUMERIC`
LC_NUMERIC database for *locale*

SEE ALSO

`chrtbl(1M)`, `montbl(1M)`, `setlocale(3C)`, `locale(5)`.

lockf(3C)**NAME**

lockf, lockf64 - lock file records

SYNOPSIS

```
#include <unistd.h>
int lockf(int fildes, int function, off_t size);
int lockf64(int fildes, int function, off64_t size);
```

DESCRIPTION

lockf() allows sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file [see [chmod\(2\)](#)]. Locking calls from other processes 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.

fildes 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.

function 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 return if not implemented.

F_TEST is used to determine if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

size is the number of contiguous bytes to be locked or unlocked. The resource to be locked 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, i.e. from the current offset through the present or any future end-of-file. An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. Locked sections will be unlocked starting at the point of the offset through *size* bytes or to the end of file if *size* is 0. When this situation occurs in the present section or in adjacent ones, the sections are combined to form one section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK will cause the calling process to sleep until the resource is available. F_TLOCK will cause the function to return a -1 and set *errno* to EACCES if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an *errno* is set to ENOLK 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.

Attempting to lock a section of a file that is associated with a buffered stream produces unspecified results.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by requesting another process' locked resource. Thus calls to `lockf()` or `fcntl()` scan for possible deadlocks 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()` system call may be used to provide a timeout facility in applications that require this facility.

The interaction between `fcntl()` and `lockf()` locks is unspecified.

Blocking on a section is interrupted by any signal.

There is no functional difference between `lockf()` and `lockf64()`, except for the interpretation of `off64_t` [see [lfs\(5\)](#)].

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`lockf()` will fail if:

EBADF

fildes is not a valid open descriptor.

EACCES or EAGAIN

function is `F_TLOCK` or `F_TEST` and the section is already locked by another process.

EDEADLK

function is `F_LOCK` and a deadlock would occur.

ENOLK

function is `F_LOCK`, `F_TLOCK`, or `F_ULOCK` and the number of entries in the lock table would exceed the number allocated on the system.

ECOMM

fildes is on a remote machine and the link to that machine is no longer active.

EINTR

A signal was caught during execution of the function.

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()`.

EDEADLK or ENOLK

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 implementation does not support the locking of files of the type indicated by the *fildes* argument.

NOTES

Unexpected results may occur in processes that do buffering in the user address space. The process may later read or write data that is or was locked. The standard I/O package is the most common reason for unexpected buffering.

Because in the future the variable `errno` will be set to `EAGAIN` rather than `EACCES` when a section of a file is already locked by another process, portable application programs should expect and test for either value.

Record-locking should not be used in combination with the `fopen()`, `fread()`, `fwrite()` and other `stdio` functions. Instead, the more primitive, non-buffered functions [such as `open()`] 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()` function may be used to provide a timeout facility in applications requiring it.

SEE ALSO

`alarm(2)`, `chmod(2)`, `close(2)`, `creat(2)`, `fcntl(2)`, `open(2)`, `read(2)`, `write(2)`, `unistd(4)`, `lfs(5)`.

NAME

`lsearch`, `lfind` - linear search and amendment

SYNOPSIS

```
#include <search.h>

void *lsearch(const void *key, void *base, size_t *nel,
             size_t width, int (*compar) (const void *, const void *));

void *lfind(const void *key, const void *base, size_t *nel,
            size_t width, int (*compar) (const void *, const void *));
```

DESCRIPTION

`lsearch()` is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum was found. If the datum does not occur, it is added at the end of the table. *key* points to the datum to be sought in the table. *base* points to the first element in the table. *nel* 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. *width* is the size of an element in bytes. *compar* is a pointer to the comparison function that the user must supply (`strcmp`, for example). Two arguments are expected, that point to the elements being compared. The function returns 0 if the elements are equal and non-zero otherwise.

`lfind()` is the same as `lsearch()` except that if the datum is not found, it is not added to the table. Instead, a null pointer is returned.

EXAMPLES

The following 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>
#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;
}
```

RETURN VALUE

If the searched-for datum is found, both `lsearch()` and `lfind()` return a pointer to it. Otherwise, `lfind()` returns `NULL` and `lsearch()` returns a pointer to the newly added element.

Undefined results can occur if there is not enough room in the table to add a new item.

NOTES

The pointers to the key and the element at the base of the table may be pointers to any type.

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 value returned should be cast into type pointer-to-element.

SEE ALSO

[bsearch\(3C\)](#), [hsearch\(3C\)](#), [string\(3C\)](#), [tsearch\(3C\)](#), [search\(5\)](#).

makecontext(3C) - msync(3C)

Programmer's Reference Manual

makecontext(3C)

NAME

`makecontext`, `swapcontext` - amend user contexts

SYNOPSIS

```
#include <ucontext.h>

void makecontext(ucontext_t *ucp, (void *func)(), int
    argc, ...);

int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

DESCRIPTION

These functions are useful for implementing user-level context switching between multiple threads of control within a process.

`makecontext()` modifies the context specified by `ucp`, which has been initialized using `getcontext()`; when this context is resumed using `swapcontext()` or `setcontext()` [see [getcontext\(2\)](#)], program execution continues by calling the function `func`, passing it the arguments that follow `argc` in the `makecontext()` call.

Before a call is made to `makecontext()`, the context being modified should have a stack allocated for it. The value of The integer value of `argc` must match the number of arguments that follow `argc`. Otherwise the behavior is undefined.

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 initialised prior to the call to `makecontext()`. `swapcontext()` saves the current context in the context structure pointed to by `oucp` and sets the context to the context structure pointed to by `ucp`.

RETURN VALUE

On successful completion, `swapcontext()` returns a value of zero. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `makecontext()` function will fail if:

EFAULT

`ucp` or `oucp` points to an invalid address.

The `swapcontext()` function will fail if:

ENOMEM

`ucp` does not have enough stack left to complete the operation.

SEE ALSO


```
exit(2), getcontext(2), sigaction(2), sigprocmask(2), ucontext(5).
```

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()` routine returns a formatted device number on success and `NODEV` on failure. *maj* is the major number. *min* is the minor number. `makedev()` can be used to create a device number for input to `mknod(2)`.

The `major()` routine returns the major number component from *device*.

The `minor()` routine returns the minor number component from *device*.

ERRORS

`makedev()` will fail if one or more of the following are true:

`EINVAL`

One or both of the arguments *maj* and *min* is too large.

`EINVAL`

The device number created from *maj* and *min* is `NODEV`.

`major()` will fail if one or more of the following are true:

`EINVAL`

The *device* argument is `NODEV`.

`EINVAL`

The major number component of *device* is too large.

`minor()` will fail if the following is true:

`EINVAL`

The *device* argument is `NODEV`.

RETURN VALUE

On failure, `NODEV` is returned and `errno` is set to indicate the error.

SEE ALSO

`mknod(2)`, `stat(2)`, `types(5)`.

malloc(3C)**NAME**

malloc, free, realloc, calloc, memalign, valloc - memory allocator

SYNOPSIS

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
void *calloc(size_t nelem, size_t elsize);
void *memalign(size_t alignment, size_t size);
void *valloc(size_t size); □
```

DESCRIPTION

malloc() and free() provide a simple memory allocation package. malloc() returns a pointer to a block of at least *size* bytes suitably aligned for any use. The space is not initialized.

The argument to free() is a pointer to a block previously allocated by malloc(), calloc(), realloc() or valloc(). After free() is performed this space is made available for further allocation. If *ptr* is a NULL pointer, no action occurs. If *ptr* points to free space, behavior is undefined.

Undefined results will occur if the space assigned by malloc() is overrun or if some random number is handed to free().

realloc() 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 *ptr* is NULL, realloc() behaves like malloc() for the specified *size*. If *size* is zero and *ptr* is not a null pointer, the object pointed to is freed.

calloc() allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

The order and contiguity of storage allocated by successive calls to calloc() is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly freed or reallocated). Each such allocation will yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-dependent; the value returned will be either a null pointer or a unique pointer.

memalign() 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:* the value of *alignment* must be a power of two, and must be greater than or equal to the size of a word.

valloc(*size*) is equivalent to

```
memalign(sysconf(_SC_PAGESIZE), size).
```

Each of the allocation routines returns a pointer to space suitably aligned for storage of any type of object.

malloc(), realloc(), calloc(), memalign(), and valloc() will fail if there is not enough available memory.

RETURN VALUE

If there is no available memory, malloc(), realloc(), calloc(), memalign(), and valloc() return a null pointer. When realloc() returns NULL, the block pointed to by *ptr* is left intact. A pointer to the requested space is returned.

If *size* is 0 for malloc(), or *nelem* or *elsize* is 0 for calloc(), a pointer with a unique value is returned, and

this can be passed to `free()`.

If `size` is 0 for `realloc()`, the space from the address `ptr` is released again. The address must be a valid one, as returned by `malloc()`.

ERRORS

The `malloc()` function will fail if:

`ENOMEM`

Insufficient storage space is available.

NOTES

The algorithm used by `malloc(3C)` is a "best-fit" algorithm. If you want to use the "first-fit" algorithm, you should use `malloc(3X)`.

Applications that use the `memalign()` and `valloc()` routines, must not link in the `libmalloc` library [see `malloc(3X)`], because these routines are based on the `malloc(3C)` algorithm.

If `malloc()` is used in a 64-bit system, the available memory is dependent on the size of all swap areas [cf. `swap(1M)`].

SEE ALSO

`mmap(2)`, `malloc(3X)`, `stdlib(5)`.

mbchar(3C)**NAME**

mbchar:
mbtowc, mblen, wctomb, mbrlen, mbrtowc, wctomb - multibyte character conversion

SYNOPSIS

```
#include <stdlib.h>
#include <wchar.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int mblen(const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
size_t wctomb(char *s, wchar_t wc, mbstate_t *ps);
```

DESCRIPTION

Multibyte characters are used to represent characters in an extended character set. This is needed for locales where 8 bits are not enough to represent all the characters in the character set.

These functions allow multibyte characters to be translated into wide-characters and vice versa.

Wide-characters have type `wchar_t` (defined in `stdlib.h`), which is an integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

A maximum of 3 extended character sets are supported for each locale. The number of bytes in an extended character set is defined by the `LC_CTYPE` category of the locale [see [setlocale\(3C\)](#)]. However, the maximum number of bytes in any multibyte character will never be greater than `MB_LEN_MAX`, a constant defined in `limits.h`. The maximum number of bytes in a character in an extended character set in the current locale is given by the macro `MB_CUR_MAX`, which is defined in `stdlib.h`.

`mbtowc()` determines the number of bytes that comprise the multibyte character pointed to by `s`. Also, if `pwc` is not a null pointer, `mbtowc()` converts the multibyte character to a wide-character and places the result in the object pointed to by `pwc`. The value of the wide-character corresponding to the null character is zero. At most `n` characters will be examined, starting at the character pointed to by `s`.

If `s` is a null pointer, `mbtowc()` simply returns 0. If `s` is not a null pointer, then, if `s` points to the null character, `mbtowc()` returns 0; if the next `n` or fewer bytes form a valid multibyte character, `mbtowc()` returns the number of bytes that comprise the converted multibyte character; otherwise, `s` does not point to a valid multibyte character and `mbtowc()` returns -1.

`mblen()` determines the number of bytes comprising the multibyte character pointed to by `s`. This function is equivalent to the call:

```
mbtowc((wchar_t *)0, s, n);
```

`wctomb()` determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is `wchar` and, if `s` is not a null pointer, stores the multibyte character representation in the array pointed to by `s`. At most `MB_CUR_MAX` characters are stored.

If `s` is a null pointer, `wctomb()` simply returns 0. If `s` is not a null pointer, `wctomb()` returns -1 if the value of `wchar` does not correspond to a valid multibyte character; otherwise it returns the number of bytes that comprise the multibyte character corresponding to the value of `wchar`.

Restartable multibyte/wide-character conversion functions

These functions differ from the corresponding multibyte character functions of `mblen()`, `mbtowc()`, and

`wctomb()` in that they have an extra parameter, *ps*, of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If *ps* is a null pointer, each function uses its own internal `mbstate_t` object instead, which is initialized at program startup to the initial conversion state. The implementation behaves as if no library function calls these functions with a null pointer for *ps*.

The `mbrlen()` function is equivalent to the call:

```
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal)
```

where *internal* is the `mbstate_t` object for the `mbrlen()` function.

If *s* is a null pointer, the `mbrtowc()` function is equivalent to the call:

```
mbrtowc(NULL, "", l, ps)
```

In this case, the values of the parameters *pwc* and *n* are ignored.

If *s* is a null pointer, the `wctomb()` function is equivalent to the call:

```
wctobm(buf, L'\0', ps)
```

where *buf* is an internal buffer.

RETURN VALUE

If *s* is a null pointer, `mbtowc()` or `mblen()` return a non-zero or 0 value, if character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, `mbtowc()` or `mblen()` either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the converted character (if the next *n* or fewer bytes form a valid character), or return -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.

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.

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.

If `s` is a null pointer, `wctomb()` returns a non-zero or 0 value, if character encodings, respectively, do or do not have state-dependent encodings. 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

The `mbtowc()` or `mblen()` function may fail if:

`EILSEQ`

Invalid character sequence is detected.

The `mbrlen()` function may fail if:

`EILSEQ`

Invalid character sequence is detected.

`EINVAL`

`ps` points to an object that contains an invalid conversion state.

The `mbrtowc()` function may fail if:

`EILSEQ`

Invalid character sequence is detected.

`EINVAL`

`ps` points to an object that contains an invalid conversion state.

No errors are defined for `wctomb()`.

SEE ALSO

[chrtbl\(1M\)](#), [mbstring\(3C\)](#), [setlocale\(3C\)](#), [environ\(5\)](#), [stdlib\(5\)](#), [wchar\(5\)](#).

mbsinit(3C)**NAME**

`mbsinit` - initial shift state

SYNOPSIS

```
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

DESCRIPTION

If *ps* is not a null pointer, the `mbsinit()` function determines whether the pointed-to `mbstate_t` object describes an initial shift state.

RETURN VALUES

The `mbsinit()` function returns nonzero if *ps* is a null pointer or if the pointed-to object describes an initial shift state; otherwise, it returns zero.

SEE ALSO

`btowc(3C)`, `wctob(3C)`, `wchar(5)`.

mbstring(3C)**NAME**

mbstring:
mbstowcs, wcstombs, mbsrtowcs, wcsrtombs - multibyte string functions

SYNOPSIS

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

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t*ps);
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t*ps);
```

DESCRIPTION

mbstowcs() converts a sequence of multibyte characters from the array pointed to by *s* into a sequence of corresponding wide-character codes and stores these into the array pointed to by *pwcs*. Execution is halted after *n* characters are stored or a character with value zero (a converted null character) is stored. If an invalid multibyte character is encountered, **mbstowcs()** returns the value `(size_t)-1`. Otherwise, **mbstowcs()** returns the number of array elements modified, not including the terminating zero code, if any.

wcstombs() converts a sequence of wide-character codes from the array pointed to by *pwcs* into a sequence of multibyte characters and stores these multibyte characters into the array pointed to by *s*. Execution is halted if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored. If a wide-character code is encountered that does not correspond to a valid multibyte character, **wcstombs()** returns the value `(size_t)-1`. Otherwise, **wcstombs()** returns the number of bytes modified, not including a terminating null character, if any.

The **mbsrtowcs()** function converts a sequence of multibyte 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 not in the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if *dst* is not a null pointer) when *len* codes have been stored into the array pointed to by *dst*. Thus, the value of *len* is ignored if *dst* is 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 multibyte 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.

The **wcsrtombs()** function converts a sequence of wide-characters from the array indirectly pointed to by *src* into a sequence of corresponding multibyte 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 two cases; when a code is reached that does not correspond to a valid multibyte character, or (if *dst* is not a null pointer) when the next multibyte character would exceed the limit of *len* total bytes to be stored into the array pointed to by *dst*. Each conversion takes place as if by a call to the **wcrtombs()** function. If conversion stops because a terminating null wide-character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte.

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.

RETURN VALUE

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`.

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`.

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).

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 `mbstowcs()` function may fail if:

`EILSEQ`

Invalid byte sequence is detected.

The `wcstombs()` function may fail if:

`EILSEQ`

A wide-character code does not correspond to a valid character.

The `mbsrtowcs()` function may fail if:

`EINVAL`

`ps` points to an object that contains an invalid conversion state.

`EILSEQ`

Invalid character sequence is detected.

The `wcsrtombs()` function may fail if:

`EINVAL`

`ps` points to an object that contains an invalid conversion state.

`EILSEQ`

A wide-character code does not correspond to a valid character.

SEE ALSO

`chrtbl(1M)`, `mbchar(3C)`, `setlocale(3C)`, `environ(5)`, `stdlib(5)`, `wchar(5)`.

NAME

memory:
 memccpy, memchr, memcmp, memcpy, memmove, memset, wmemchr, wmemcmp, wmemcpy,
 wmemmove, wmemset - memory operations

SYNOPSIS

```
#include <string.h>
void *memccpy(void *s1, const void *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 *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
void *memset(void *s, int c, size_t n);□
cc [ flag ... ] file ... -lw [ library ... ]
#include <string.h>
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);□
```

DESCRIPTION

These functions enable memory areas to be efficiently processed. Memory areas are arrays of bytes which are bounded by a count, not terminated by a null character. The specified memory areas are not checked for the overflow.

`memccpy()` copies bytes from memory area `s2` into `s1`, and stops either after the first occurrence of `c` has been copied (which converts it to an `unsigned char`), or after `n` bytes have been copied. `memccpy()` 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`.

`memchr()` returns a pointer to the first occurrence of `c` in the first `n` bytes (each interpreted as an `unsigned char`) of memory area `s`, or a null pointer if `c` does not occur.

`memcmp()` compares the arguments `s1` and `s2`, comparing only the first `n` bytes (each interpreted as an `unsigned char`), and returns an integer less than, equal to, or greater than 0, according to whether `s1` is lexicographically less than, equal to, or greater than `s2`.

`memcpy()` copies `n` bytes from memory area `s2` to `s1`. It returns `s1`.

`memmove()` copies `n` bytes from memory areas `s2` to `s1`. Copying between objects that overlap will take place correctly. It returns `s1`.

`memset()` sets the first `n` bytes in memory area `s` to the value of `c` (converted to an `unsigned char`). It returns `s`.

The `wmemchr()` function locates the first occurrence of `c` in the initial `n` wide-characters of the object pointed to by `s`.

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.

The `wmemcmp()` function compares the first `n` wide-characters of the object pointed to by `s1` to the first `n`

wide-characters of the object pointed to by *s2*.

The `wmemcmp()` function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*.

The `wmemcpy()` function copies *n* wide-characters from the object pointed to by *s2* to the object pointed to by *s1*.

The `wmemcpy()` function returns the value of *s1*.

The `wmemmove()` function copies *n* wide-characters from the object pointed to by *s2* to the object pointed to by *s1*. Copying takes place as if the *n* wide-characters from the object pointed to by *c* are first copied into a temporary array of *n* wide-characters that does not overlap the objects pointed to by *s1* or *s2*, and then the *n* wide-characters from the temporary array are copied into the object pointed to by *s1*.

The `wmemmove()` function returns the value of *s1*.

The `wmemset()` function copies the value of *c* into each of the first *n* wide-characters of the object pointed to by *s*.

The `wmemset()` function returns the value of *s*.

ERRORS

No errors are defined.

SEE ALSO

`string(3C)`, `string(5)`, `wchar(5)`.

mkfifo(3C)**NAME**

mkfifo - create a new FIFO special file

SYNOPSIS

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

DESCRIPTION

The `mkfifo()` routine creates a new FIFO special file named by the pathname pointed to by *path*. The mode of the new FIFO special file is initialized from *mode*. The file permission bits of the *mode* argument are modified by the process' file creation mask [see `umask(2)`].

When bits in *mode* other than the file permission bits are set, the effect is implementation-dependent.

The owner id of the FIFO special file is set to the process' effective user id. The group id of the FIFO special file is set to the process' effective group id, or if the `S_ISGID` bit is set in the parent directory then the group id of the FIFO is inherited from the parent.

Upon successful completion, `mkfifo()` will mark 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.

`mkfifo()` calls the system call `mknod()` to make the file.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

`mkfifo()` (or `mknod()`) will fail and not generate a new file if one of the following applies:

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

Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG

The length of the *path* argument exceeds `PATH_MAX`, or the length of a component of *path* exceeds `NAME_MAX`.

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.

ENOENT

A component of the path prefix does not exist or is an empty pathname.

EROFS

The directory in which the file is to be created is on a read-only file system.

The `mkfifo()` function may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds

`PATH_MAX`.

RESULT

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

NOTES

Bits other than the file permission bits in *mode* are ignored.

SEE ALSO

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

mkstemp(3C)**NAME**

mkstemp - make a unique file name

SYNOPSIS

```
#include <stdlib.h>
int mkstemp(char *template);
```

DESCRIPTION

mkstemp() creates a unique file name, typically in a temporary file system, by replacing *template* with a unique file name, and returns a file descriptor for the template file open for reading and writing. The string in *template* should contain a file name with six trailing **XS**; **mkstemp()** replaces the **XS** with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file. **mkstemp()** avoids the race between testing whether the file exists and opening it for use.

RETURN VALUE

mkstemp() returns **-1** if no suitable file could be created.

NOTES

It is possible to run out of letters.

mkstemp actually changes the template string which you pass; this means that you cannot use the same template string more than once – you need a fresh template for every unique file you want to open.

When **mkstemp()** is creating a new unique filename it checks for the prior existence of a file with that name. This means that if you are creating more than one unique filename, it is bad practice to use the same root template for multiple invocations of **mkstemp()**.

SEE ALSO

[getpid\(2\)](#), [open\(2\)](#), [tmpfile\(3S\)](#), [tmpnam\(3S\)](#), [stdlib\(5\)](#).

NAME

mktemp - make a unique filename

SYNOPSIS

```
#include <stdlib.h>
□
□
char *mktemp(char *template);
```

DESCRIPTION

`mktemp()` replaces the contents of the string pointed to by *template* with a unique filename, and returns the amended *template*. The string in *template* should look like a filename with six trailing X; `mktemp()` will replace the X with a character string that can be used to create a unique filename.

RESULT

`mktemp()` will assign to *template* the empty string if it cannot create a unique name.

NOTES

`mktemp()` can create only 26 unique filenames per process for each unique *template*.

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()` function avoids this problem.

SEE ALSO

`mkstemp(3)`, `tmpfile(3S)`, `tmpnam(3S)`, `stdlib(5)`.

mktime(3C)**NAME**

mktime - convert a `tm` structure to a calendar time

SYNOPSIS

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

DESCRIPTION

`mktime()` 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 has the following format:

```
struct    tm {
    int    tm_sec;        /* seconds after the minute [0, 61] */
    int    tm_min;        /* minutes after the hour [0, 59] */
    int    tm_hour;       /* hours 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 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, the correct timezone is determined and the components are not adjusted.

Local timezone information is used as if `mktime()` had called `tzset()`.

RETURN VALUE

`mktime()` returns the specified calendar time. If the calendar time cannot be represented, the function returns the value `time_t-1`.

ERRORS

No errors are defined.

EXAMPLES

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]);

```

NOTES

`tm_year` of the `tm` structure must be for year 1970 or later. Calendar times before 00:00:00 UTC, January 1, 1970 or after 03:14:07 UTC, January 19, 2038 cannot be represented.

`mktime()` evaluates the timezone variable `TZ` [cf. [sh\(1\)](#), section "Standard shell variables"]. If, for example, `TZ` is set to `MET-1`, the number of seconds is reduced by 3600.

SEE ALSO

[ctime\(3C\)](#), [getenv\(3C\)](#), [timezone\(4\)](#), [time\(5\)](#).

mlock(3C)**NAME**

mlock, munlock - lock (or unlock) pages in memory

SYNOPSIS

```
#include <sys/types.h>
int mlock(caddr_t addr, size_t len);
int munlock(caddr_t addr, size_t len);
```

DESCRIPTION

The function `mlock()` uses the mappings established for the address range `[addr, addr + len)` to identify pages to be locked in memory. The effect of `mlock(addr, len)` is equivalent to `memcntl(addr, len, MC_LOCK, 0, 0, 0)`.

`munlock()` removes locks established with `mlock()`. The effect of `munlock(addr, len)` is equivalent to `memcntl(addr, len, MC_UNLOCK, 0, 0, 0)`.

Locks established with `mlock()` are not inherited by a child process after a `fork()` and are not nested.

RETURN VALUE

Upon successful completion, the functions `mlock()` and `munlock()` return 0; otherwise, they return -1 and set `errno` to indicate the error.

NOTES

To be able to use `mlock()` and `munlock()`, the user must have appropriate privileges.

SEE ALSO

`fork(2)`, `memcntl(2)`, `mmap(2)`, `plock(2)`, `mlockall(3C)`, `sysconf(3C)`, `types(5)`.

mlockall(3C)**NAME**

mlockall, munlockall - lock or unlock address space

SYNOPSIS

```
#include <sys/mman.h>
□
□
□
int mlockall(int flags);
int munlockall(void);
```

DESCRIPTION

The function `mlockall()` causes all pages mapped by an address space to be locked in memory. The effect of `mlockall(flags)` is equivalent to:

```
memcntl(0, 0, MC_LOCKAS, flags, 0, 0)
```

The value of *flags* determines whether the pages to be locked are those currently mapped by the address space, those that will be mapped at some later point in time, or both:

`MCL_CURRENT`

Lock current mappings

`MCL_FUTURE`

Lock future mappings

The function `munlockall()` removes address space locks and locks on mappings in the address space. The effect of `munlockall()` is equivalent to:

```
memcntl(0, 0, MC_UNLOCKAS, 0, 0, 0)
```

Locks established with `mlockall()` are not inherited by a child process after a `fork()` and are not nested.

RETURN VALUE

Upon successful completion, the functions `mlockall()` and `munlockall()` return 0; otherwise, they return -1 and set `errno` to indicate the error.

NOTES

To be able to use `mlock()` and `munlock()`, the user must have appropriate privileges.

SEE ALSO

`fork(2)`, `memcntl(2)`, `mmap(2)`, `plock(2)`, `mlock(3C)`, `sysconf(3C)`, `mman(5)`.

monitor(3C)**NAME**

monitor - prepare execution profile

SYNOPSIS

```
#include <mon.h>

void monitor(int (*lowpc)(), int (*highpc)(), WORD *buffer,
             size_t bufsize, size_t nfunc);
```

DESCRIPTION

monitor() is an interface to **profil()**, and is called automatically with default parameters by any program created by **cc -p**. **monitor()** only needs to be explicitly called if additional information is required for the execution-profile.

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 the compiler option **-p**. The last call to **monitor()** writes this collected data to the output file **mon.out**.

lowpc and **highpc** are the beginning and ending addresses of the region to be profiled.

buffer is used by **monitor()** to store the histogram generated by **profil()** and the call counts.

bufsize identifies the number of array elements in **buffer**.

nfunc is the number of call count cells that have been reserved in **buffer**. Additional call count cells will be allocated automatically as soon as they are needed.

bufsize should be computed using the following (C notation) 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** have the same values as the arguments to **monitor()**;
- **BARSIZE** is the number of program bytes allocated to each histogram bar, or cell, of the **profil()** buffer;
- The **hdr** and **cnt** structures and the type **WORD** are defined in the header file **mon.h**.

The standard invocation of **monitor()** is as follows (C notation):

```
monitor (&eprol, &etext, wbuf, wbufsz, 600);
```

where:

- **eprol** is the beginning of the user's program when compiled with **-p**;
- **etext** is the end of the user's program;
- **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**.

With these parameter settings, an execution-time distribution histogram can be calculated, using **profil()** for the entire program. Initially, 600 cells are reserved in **buffer**, and enough histogram entries are provided to generate significant distribution-measurement results. (For more information on the effects of **bufsize** on execution-distribution measurements, see [profil\(2\)](#).)

To stop execution monitoring and write the results to a file, use the following:

```
monitor((int (*)())0, (int (*)())0, (WORD *)0, 0, 0);
```

NOTES

Additional calls to `monitor()` after `main()` has been called and before `exit()` has been called will add to the function-call count capacity. However, such calls will also replace and restart the `profil()` histogram computation.

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 `-p`, the file created is *dirname/pid.progname* where *progname* is the name of the program.

FILES

`mon.out`

SEE ALSO

`cc(1)`, `profil(2)`.

msync(3C)**NAME**

msync - synchronize storage

SYNOPSIS

```
#include <sys/mman.h>
int msync(void *addr, size_t len, int flags);□
```

DESCRIPTION

The `msync()` function writes all modified data to permanent storage locations, if any, in those whole pages containing any part of the address space of the process starting at address `addr` and continuing for `len` bytes. If no such storage exists, `msync()` need not have any effect. If requested, the `msync()` function then invalidates cached copies of data.

The implementation will require that `addr` be a multiple of the page size as returned by `sysconf()`.

For mappings to files, the `msync()` function ensures that all write operations are completed as defined for synchronised I/O data integrity completion. It is unspecified whether the implementation also writes out other file attributes. When the `msync()` function is called on `MAP_PRIVATE` mappings, any modified data will not be written to the underlying object and will not cause such data to be made visible to other processes. It is unspecified whether data in `MAP_PRIVATE` mappings has any permanent storage locations.

The `flags` argument is constructed from the bitwise inclusive OR of one or more of the following flags defined in the header `<sys/mman.h>`:

Symbolic Constant	Description
<code>MS_ASYNC</code>	Perform asynchronous writes
<code>MS_SYNC</code>	Perform synchronous writes
<code>MS_INVALIDATE</code>	Invalidate cached data

When `MS_ASYNC` is specified, `msync()` returns immediately once all the write operations are initiated or queued for servicing; when `MS_SYNC` is specified, `msync()` will not return until all write operations are completed as defined for synchronised I/O data integrity completion. Either `MS_ASYNC` or `MS_SYNC` is specified, but not both.

When `MS_INVALIDATE` is specified, `msync()` invalidates all cached copies of mapped data that are inconsistent with the permanent storage locations such that subsequent references obtain data that was consistent with the permanent storage locations sometime between the call to `msync()` and the first subsequent memory reference to the data.

The behavior of this function is unspecified if the mapping was not established by a call to `mmap()`.

If `msync()` causes any write to a file, the file's `st_ctime` and `st_mtime` fields are marked for update.

RETURN VALUE

Upon successful completion, the function `msync()` returns 0; otherwise, it returns `-1` and sets `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `msync()` function will fail if:

`EINVAL`

The value in `flags` is invalid.

EINVAL

The value of *addr* is not a multiple of the page size, `PAGESIZE`.

ENOMEM

The addresses in the range starting at *addr* and continuing for *len* bytes are outside the range allowed for the address space of a process or specify one or more pages that are not mapped.

NOTES

`msync()` should be used by programs that require a memory object to be in a known state, for example, in transaction processing.

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.

SEE ALSO

`memcntl(2)`, `mmap(2)`, `sysconf(3C)`, `mman(5)`.

ndbm(3C-ucb) - remove(3C)

Programmer's Reference Manual

ndbm(3C)

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.

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 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()` except that a database opened for write-only access opens the files for read and write access and the behavior of the `O_APPEND` flag is unspecified. The *file_mode* argument

has the same meaning as the third argument of `open()`.

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 1023 bytes.

The `dptr` pointers returned by these functions may point into static storage that may be changed by subsequent calls.

These interfaces need not be thread-safe.

RETURN VALUE

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.

APPLICATION 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 `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()`.

SEE ALSO

`open(2)`, `ndbm(5)`.

NAME

`nice` - change priority of a process

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <types.h>
#include <time.h>
#include <sys/resource.h>
int nice(int incr) ; □
```

DESCRIPTION

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without undue impact on system performance.

Negative increments are illegal, except when specified by the privileged user. The priority is limited to the range -20 (most urgent) to 20 (least). Requests for values above or below these limits result in the scheduling priority being set to the corresponding limit.

The priority of a process is passed to a child process by `fork()`. For a privileged process to return to normal priority from an unknown state, `nice()` should be called successively with arguments -40 (goes to priority -20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

RETURN VALUE

Upon successful completion, `nice()` returns 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

The priority is not changed if:

`EACCES`

The value of *incr* specified was negative, and the effective user ID is not the privileged user.

SEE ALSO

`nice(1)`, `prctl(1)`, `renice(1)`, `fork(2)`, `prctl(2)`, `getpriority(3C)`, `resource(5)`, `time(5)`, `types(5)`.

nl_langinfo(3C)**NAME**

nl_langinfo - get language information

SYNOPSIS

```
#include <langinfo.h>
char *nl_langinfo(nl_item item);
```

DESCRIPTION

nl_langinfo() 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 in <langinfo.h>. For example:

```
nl_langinfo (ABDAY_1);
```

would return a pointer to the string "Dom" if the identified language was Portuguese and a French locale was correctly installed; or to "Sun" if the identified language was English.

Calls to setlocale() with a category corresponding to the category of *item* [see langinfo(5)], or to the category LC_ALL, may overwrite the array pointed to by the return value.

This interface need not be thread-safe.

RETURN VALUE

If setlocale() has not been called successfully, or if "langinfo" data for a supported language is either not available or *item* is not defined therein, then nl_langinfo() returns a pointer to the corresponding string in the C locale. If *item* contains an invalid string, nl_langinfo() always returns a pointer to an empty string.

This pointer may point to static data that may be overwritten on the next call.

ERRORS

No errors are defined.

NOTES

The array pointed to by the return value should not be modified by the program. Subsequent calls to nl_langinfo() may overwrite the array.

Some of the constants defined in <langinfo.h> are recognized in XPG4 mode only. If used as input in XPG3 mode, nl_langinfo() returns a pointer to an empty string. A note to this effect is included in the table for langinfo(5).

SEE ALSO

gettext(3C), localeconv(3C), setlocale(3C), strftime(3C), LC_TIME(4), langinfo(5), nl_types(5).

NAME

`offsetof` - offset of structure member

SYNOPSIS

```
#include <stddef.h>
size_t offsetof(type, member-designator);
```

DESCRIPTION

`offsetof()` is a macro defined in `stddef.h` which expands to an integral constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by *memberdesignator*), from the beginning of its structure (designated by *type*).

perror(3C)**NAME**

`perror` - print system error messages

SYNOPSIS

```
#include <stdio.h>
void perror(const char *s);
```

DESCRIPTION

`perror()` produces a message on the standard error output, describing the last error which occurred during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a newline. However, if *s* is a null pointer or points to a null string, the colon is not printed. For the argument string *s* to be useful, it 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.

The contents of the string returned by `perror()` are determined by the setting of the `LC_MESSAGES` environment variable in the current locale.

If standard error output is redirected to a file, this is marked as having been written (`st_ctime` and `st_mtime` marked for update) after the successful completion of `perror()` and before `exit()`, `abort()`, or after the completion of `fflush()` or `fclose()` on `stderr`.

SEE ALSO

[fmtmsg\(3C\)](#), [strerror\(3C\)](#).

pollbunch(3C)**NAME**

`pollbunch`, `pollwhich` - multiplex input/output: more powerful version of the `poll(2)` system call

SYNOPSIS

```
#include <stropts.h>
#include <poll.h>
int pollbunch(int cmd, struct np_pollfd *fds);
int pollwhich(struct np_pollfd *fds, size_t nfd, int timeout);
```

DESCRIPTION

`pollbunch` and `pollwhich` offer users a powerful mechanism for multiplexing input/output via a set of file descriptors that reference open files. `pollbunch/pollwhich` is to be recommended over `poll(2)` for applications with a large number of files to be multiplexed with relatively low individual event rates.

This new mechanism offers a more efficient event status query than `poll(2)`. With `poll(2)`, the entire list of file descriptors must be transferred as an argument each time, even if this list is rarely modified and even then few changes made. The same size list is returned by `poll(2)` as an event, even if it only contains a few files with "active" events. The application must then find out the location of these file descriptors in the list by reading through it in sequence.

The division of the `poll(2)` system call into the two system calls `pollbunch` and `pollwhich` separates the administrative part from the actual event status query, and by using the asynchrony of the events allows the synchronous proportion of time during the actual event status query to be limited to the file descriptors for which an event already occurred. Only these file descriptors are returned as an event. This reduces the copying time between kernel and application and avoids the long, sequential program runs through all specified file descriptors both in the kernel and the application. This results in better response times in the application and an off-loading of the CPU.

The event status event list also supplies the events in the chronological order in which they occurred. This means that special processing on the part of the application to make sure that no file descriptors are discriminated against is unnecessary.

The `np_pollfd` structure is used to transfer file descriptors and events following `poll(2)` and contains the following:

```
int fd;                                /* file descriptor */
short events;                          /* requested or returned events */
unsigned short userref;                 /* transparent user argument */
```

`fd` specifies an open file descriptor and the `events` field is a bitmask constructed by an ORing of any combination of the following event flags, as in `poll(2)`:

POLLIN

Data other than high priority can be read without blocking. This option is also set for STREAMS if the message is of zero length.

POLLRDNORM

Normal data (priority = 0) can be read without blocking. This option is also set for STREAMS if the message is of zero length.

POLLRDBAND

Data with a non-zero priority can be read without blocking. This option is also set for STREAMS if the message is of zero length.

POLLPRI

High priority data can be read without blocking. This option is also set for STREAMS if the message is of zero length.

POLLOUT

Normal data can be written without blocking.

POLLWRNORM

Like POLLOUT.

POLLWRBAND

Data with a non-zero priority class can be written. This event only checks classes that have been written at least once.

POLLMSG

An `M_SIG` message or `M_PCSIG` message containing a `ASIGPOLL` signal reached the top of the stream head queue.

POLLERR

There is an error message on the stream or device. This flag is only valid in the `events` bitmask in `pollwhich`; it is not used in the `events` field for `pollbunch`.

POLLHUP

A hangup occurred in the stream. This event and `POLLOUT` are mutually exclusive; a stream can never be writable if a hangup occurred. However this event and `POLLIN` or `POLLPRI` are not mutually exclusive. This flag is only valid in the `events` bitmask for `pollwhich`; it is not used in the `events` field for `pollbunch`.

POLLNVAL

The specified `fd` value does not belong to an open file. This flag is only valid in the `events` field for `pollwhich`; it is not used in the `events` field for `pollbunch`.

The layout of the `events` field is different for `pollbunch` and for `pollwhich`.

POLLBUNCH

`pollbunch` is for the construction and manipulation of a list of relevant file descriptors and their respective events within a kernel. Using a `pollbunch`, a single file descriptor can be inserted or deleted, or its events can be modified. The respective action is specified by `cmd`. `fds` is a pointer for the `np_pollfd` structure containing the file descriptor and its events:

Permitted values for `cmd` are:

`NP_B_ADD`

Incorporates the `fd` file descriptor with the `events` into the list of relevant file descriptors. This means that consequent asynchronous events for this file descriptor are marked and displayed by `pollwhich`. The contents of `userref` in the `np_pollfd` structure are transferred transparently and saved within this file descriptor. The `np_pollfd` structure is unchanged when you return from `pollbunch`.

`NP_B_REM`

Deletes the `fd` file descriptor from the list of relevant file descriptors. This means that no more consequent events are noted for this file descriptor and the `pollwhich` file descriptor is no longer displayed. The `events` and `userref` fields in the `np_pollfd` structure are not evaluated. The `np_pollfd` structure is unchanged when you return from `pollbunch`.

`NP_B_MODIFY`

Changes the events for the existing `fd` file descriptor. The `events` value from the `np_pollfd` structure overwrites the value in the kernel. The `userref` field in the `np_pollfd` structure is not evaluated. The `np_pollfd` structure is unchanged when you return from `pollbunch`.

`pollbunch` is unsuccessful if one or more of the following occurs:

`EAGAIN`

Indicates that an internal data structure was unsuccessful. The request should nevertheless be attempted again.

`EBADF`

`fd` is not a valid file descriptor in the `np_pollfd` structure.

`EEXIST`

`fd` in the `np_pollfd` structure (`NP_B_ADD`) specifies a file descriptor that is already contained in the list of relevant file descriptors.

EFAULT

An argument is a pointer to a storage space outside the indicated address area.

EINVAL

cmd contains an invalid action.

events in the `np_pollfd` structure (`NP_B_ADD`, `NP_B_MODIFY`) either contain no events or invalid events.

ENOENT

fd in the `np_pollfd` structure (`NP_B_REM`, `NP_B_MODIFY`) specifies a file descriptor that is not contained in the list of relevant file descriptors.

RESULT

The value 0 is returned after the successful termination of `pollbunch`. Otherwise `-1` is returned and `errno` is set to display the error.

POLLWHICH

`pollwhich` supplies the file descriptors in which one or more events occurred. The results of the `pollwhich` query are saved in the *events* field in the `np_pollfd` structure. Bits are set in the *events* bitmask to display which of the requested events are true.

`pollwhich` displays the maximum number of `np_pollfd` structures in the *fds* vector as were specified by *nfds*. *fds* points to an address book with a size of at least *nfds* `np_pollfd` structures.

If a number of file descriptors are displayed in *fds*, the sequence of the `np_pollfd` structure corresponds to the chronological sequence in which each event occurred. If a number of events were specified for a file descriptor and some of them are parallel, the position of the corresponding `np_pollfd` structure within *fds* is determined by the time when the first event occurred for this file descriptor.

The options `POLLHUP`, `POLLERR`, and `POLLNVAL` which display events are always set in the *events* field if the conditions displays by you are true, this also happens if they were not in the *events* field in `pollbunch`.

`pollwhich` only considers those file descriptors that were previously specified by `pollbunch`.

The *userref* field contains the value that was specified in the corresponding `pollbunch`.

If none of the defined events occur in one of the selected file descriptors, `pollwhich` waits at least *timeout* milliseconds for an error to occur in one of the selected file descriptors. On a computer where the timing facility is not exact to the millisecond, *timeout* is rounded up to the next permitted value available in the system. If the value of *timeout* is 0, `pollwhich` is returned immediately. If the value of *timeout* is equal to `INFTIM` (or `-1`), `pollwhich` causes a block until a queried event occurs, or until the call is terminated. `pollwhich` is not affected by the `O_NDELAY` and `O_NONBLOCK` flags.

`pollwhich` is unsuccessful if one or more of the following occurs:

EAGAIN

Indicates that an internal data structure was unsuccessful. The request should nevertheless be attempted again.

EFAULT

An argument points to a storage space outside the indicated address area.

EINTR

A signal was intercepted during a `pollwhich` system call.

EINVAL

The *nfds* argument is less than 0 or greater than `OPEN_MAX`.

ENOENT

No list of relevant file descriptors exists. `pollbunch` was probably not executed previous to this.

RESULT

A nonnegative result is returned when `pollwhich` is terminated successfully. A positive value displays the total number of valid `np_pollfd` structures (i.e. that can be evaluated). All further structures (up to *nfds*) then

contain undefined (or the old) values.

0 shows that the call has been timed out and no file descriptor was selected.

If the process is unsuccessful, -1 is returned and `errno` is set to display the error.

SEE ALSO

`getmsg(2)`, `getrlimit(2)`, `poll(2)`, `putmsg(2)`, `read(2)`, `write(2)`.

Programmer's Guide: STREAMS.

psignal(3C)**NAME**

`psignal`, `psiginfo` - output system signal messages

SYNOPSIS

```
#include <siginfo.h>
void psignal(int sig, const char *s);
void psiginfo(siginfo_t *pinfo, char *s);
```

DESCRIPTION

`psignal()` and `psiginfo()` produce messages on the standard error output (`stderr`) describing the specified signal. *sig* is a signal that may have been passed as the first argument to a signal handler. *pinfo* 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, then a colon and a blank, then a system message describing the signal, and a newline.

The contents of the string returned by this function are determined by the setting of the `LC_MESSAGES` environment variable in the current locale.

SEE ALSO

[sigaction\(2\)](#), [perror\(3C\)](#), [siginfo\(5\)](#), [signal\(5\)](#).

psignal(3C-ucb)
BSD Compatibility Package**NAME**

`psignal`, `sys_siglist` - system signal messages

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
psignal(sig, s)
unsigned sig;
char *s;
char *sys_siglist[];
```

DESCRIPTION

`psignal` produces a short message on the standard error file describing the indicated signal. First the argument string `s` is printed, then a colon, then the name of the signal and a NEWLINE. Most usefully, the argument string is the name of the program which incurred the signal. The signal number should be from among those found in `<signal.h>`.

To simplify variant formatting of signal names, the vector of message strings `sys_siglist` is provided; the signal number can be used as an index in this table to get the signal name without the newline. The define `NSIG` defined in `<signal.h>` is the number of messages provided for in the table; it should be checked because new signals may be added to the system before they are added to the table.

SEE ALSO

`signal(3-ucb)`, `perror(3C)`.

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 an integer between 0 and 255.

This interface need not be thread-safe.

RETURN VALUE

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.

ERRORS

No errors are defined.

SEE ALSO

[open\(2\)](#), [grantpt\(3C\)](#), [ttyname\(3C\)](#), [unlockpt\(3C\)](#), [stdlib\(5\)](#).

putenv(3C)**NAME**

putenv - change/add environment variables

SYNOPSIS

```
#include <stdlib.h>
int putenv(char *string);
```

DESCRIPTION

The `putenv()` function uses the *string* argument to set environment variable values. The *string* argument points to a string of the form *name=value*. `putenv()` 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. `putenv()` does not set up a copy of *string*. *string* should be declared static if it is declared within a function. The space used by *string* is no longer used once a new string, defined by *name*, is passed to `putenv()`.

This interface need not be thread-safe.

RETURN VALUE

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

ERRORS

The `putenv()` function may fail if:

ENOMEM

Insufficient memory was available.

NOTES

`putenv()` manipulates the environment pointed to by *environ*, and can be used in conjunction with `getenv()`.

This routine uses `malloc(3C)` to enlarge the environment.

After `putenv()` is called, environmental variables are not in alphabetical order. A potential error is to call the function `putenv()` with a pointer to an `auto` variable as the argument and to then exit the calling function while *string* is still part of the environment.

SEE ALSO

`exec(2)`, `getenv(3C)`, `malloc(3C)`, `environ(5)`, `stdlib(5)`.

putpwent(3C)**NAME**

putpwent - write password file entry

SYNOPSIS

```
#include <pwd.h>
int putpwent(const struct passwd *p, FILE *f);
```

DESCRIPTION

putpwent() is the inverse of [getpwent\(3C\)](#). Given a pointer to a passwd structure created by getpwent() (or getpwuid() or getpwnam()), putpwent() writes a line on the stream *f*, which matches the format of */etc/passwd*.

RESULT

putpwent() returns non-zero if an error was detected during its operation, otherwise zero.

SEE ALSO

[getpwent\(3C\)](#), [pwd\(5\)](#).

putspent(3C)**NAME**

`putspent` - write shadow password file entry

SYNOPSIS

```
#include <shadow.h>
int putsent(const struct spwd *p, FILE *fp);
```

DESCRIPTION

The `putspent()` routine is the inverse of `getspent()`. *p* is a pointer to a *spwd* structure created by `getspent()` or `getspnam()`. `putspent()` writes the details of this *spwd* as a line on the stream *fp*. This line matches the format of the shadow password file, */etc/shadow*.

If the `sp_min`, `sp_max`, `sp_lstchg`, `sp_warn`, `sp_inact`, or `sp_expire` field of the *spwd* structure is `-1`, or if `sp_flag` is 0, the corresponding field of the entry written on the stream *fp* is cleared.

RESULT

`putspent()` returns non-zero if an error was detected during its operation, otherwise zero.

SEE ALSO

[getpwent\(3C\)](#), [getspent\(3C\)](#), [putpwent\(3C\)](#).

qsort(3C)**NAME**

`qsort` - sort using quicksort

SYNOPSIS

```
#include <stdlib.h>
void qsort(void *base, size_t nel, size_t width,
           int (*compar) (const void *, const void *));
```

DESCRIPTION

`qsort()` is an implementation of the quicksort 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. *base* points to the element at the base of the table. *nel* is the number of elements in the table. *width* specifies the size of each element in bytes. *compar* is the name of the user-defined comparison function, which is called by `qsort()` with two arguments that point to the elements being compared. This 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.

NOTES

If two elements are the same according to this function, their sequence in the sorted table will be unpredictable.

SEE ALSO

`sort(1)`, `bsearch(3C)`, `lsearch(3C)`, `string(3C)`, `stdlib(5)`.

NAME

`raise` - send signal to program

SYNOPSIS

```
#include <signal.h>
int raise(int sig);
```

DESCRIPTION

`raise()` sends the signal *sig* to the executing thread.

`raise()` uses `kill()` to send the signal to the executing program:

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

See [kill\(2\)](#) for a detailed list of failure conditions. See [signal\(5\)](#) for a list of signals.

RETURN VALUE

`raise()` returns zero if the operation succeeds. Otherwise, `raise()` returns `-1` and `errno` is set to indicate the error.

ERRORS

The `raise()` function will fail if:

`EINVAL`

The value of the *sig* argument is an invalid signal number.

SEE ALSO

[getpid\(2\)](#), [kill\(2\)](#), [signal\(2\)](#), [signal\(5\)](#).

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 congruent random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to `RAND_MAX` (defined in `<stdlib.h>`).

The function `srand()` uses the argument *seed* as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to the function `rand()`. If the function `srand()` is then called with the same *seed* value, the sequence of pseudo-random numbers will be repeated. If the function `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()` interface need not be thread-safe.

The `rand_r()` function computes a sequence of pseudo-random integers in the range 0 to `RAND_MAX`. (The value of the `RAND_MAX` macro will be at least 32767.)

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 returns and calls to `rand_r()`, the same sequence shall be generated.

RETURN VALUE

The `rand()` function returns the next pseudo-random number in the sequence. The `srand()` function returns no value.

The `rand_r()` function returns a pseudo-random integer.

ERRORS

No errors are defined.

NOTES

The spectral properties of `rand()` are limited. `drand48(3C)` provides a much better, though more elaborate, random-number generator.

The following code defines a pair of functions which could be incorporated into applications wishing to ensure that the same sequence of numbers is generated across different machines:

```
static unsigned long int next = 1;
int myrand(void) /* RAND_MAX assumed to be 32767 */
{
    next = next * 1103515245 + 12345;
    return ((unsigned int) (next/65536) % 32768);
}
void mysrand(unsigned int seed)
{
    next = seed;
}
```

SEE ALSO

`drand48(3C)`, `pthread_intro(3T)`, `stdlib(5)`.

rand(3C-ucb)
BSD Compatibility Package**NAME**

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

SYNOPSIS

```
/usr/ucb/cc [ flag... ] file ... -lucb
```

```
srand(seed) □
```

```
int seed; □
```

```
rand();
```

DESCRIPTION

`rand` uses a multiplicative congruent random-number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$.

`srand` can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTES

The spectral properties of `rand` leave a great deal to be desired. `drand48` and `random` provide much better, though more elaborate, random-number generators.

The low bits of the numbers generated are not very random; use the middle bits. In particular the lowest bit alternates between 0 and 1.

SEE ALSO

`drand48(3C)`, `rand(3C)`, `random(3C)`, `srand(3C)`.

random(3C)**NAME**

random, srandom, initstate, setstate - pseudo-random 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 $16 \times (2^{31}-1)$. The size of the state array determines the period of the random-number generator. Increasing the state array size increases the period. With 256 bytes of state information, the period of the random-number generator is greater than 2^{69} .

Like `rand()`, `random()` produces by default a sequence of numbers that can be duplicated by calling `srandom()` with 1 as the seed.

The `srandom()` function initializes the current state array using the value of *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 greater than or equal to 8 or less than 32, `random()` uses a simple linear congruent 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 and < 32 , ≥ 8 and < 32 , then `random()` uses a simple linear congruent 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.

EXAMPLES

```
/* Initialize an array and pass it in to initstate. */
static long state1[32] = {
    3,
    0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
    0x7449e56b, 0xbdb1dbb0, 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;
    seed = 1;
    n = 128;
    initstate(seed, state1, n);
    setstate(state1);
    printf("%d\n", random());
}

```

RETURN VALUE

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.

If `initstate()` is called with size less than 8, it returns `NULL`.

ERRORS

No errors are defined.

APPLICATION 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.

Although some implementations of `random()` have written messages to standard error, such implementations do not conform to this description.

Threaded applications should use `rand_r()`, `erand48()`, `rand48()` or `jrand48()` instead of `random()` when an independent random number sequence in multiple threads is required.

SEE ALSO

`drand48(3C)`, `rand(3C)`, `srand(3C)`, `stdlib(5)`.

readdir(3C)
BSD Compatibility Package**NAME**

readdir - directory operation

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <sys/types.h>
#include <sys/dir.h>
struct direct *readdir(DIR *dirp);
```

DESCRIPTION

`readdir()` returns a pointer to the next active directory entry and positions the directory stream at the next entry. No inactive entries are returned. It returns `NULL` upon reaching the end of the directory or upon detecting an invalid location in the directory. `readdir()` buffers several directory entries per actual read operation; `readdir()` marks for update the `st_atime` field of the directory each time the directory is actually read.

ERRORS

`readdir()` returns `NULL` on failure and sets `errno` to the following value:

ENOENT

The current file pointer for the directory is not located at a valid entry. Stream has been closed.

EXAMPLES

Here is a sample program that prints the names of all the files in the current directory:

```
#include <sys/types.h>
#include <sys/dir.h>
main()
{
    DIR *p;
    struct direct *de;
    int status = 0;
    if ((p = opendir(".")) == NULL) {
        perror("opendir");
        exit(1);
    }
    while (de = readdir(p)) {
        printf("%s\n", de->d_name);
    }
    exit(status);
}
```

SEE ALSO

[directory\(3C\)](#), [direct\(4\)](#), [types\(5\)](#).

realpath(3C)**NAME**

`realpath` - output the real filename

SYNOPSIS

```
#include <stdlib.h>
#include <sys/param.h>
char *realpath(char *file_name, char *resolved_name);
```

DESCRIPTION

`realpath()` resolves all links and references to "." and ".." in *file_name* and stores it in *resolved_name*.

It can handle both relative and absolute path names. For absolute path names and the relative names whose resolved name cannot be expressed relatively (e.g., `../../reldir`), it returns the resolved absolute name. For the other relative path names, it returns the resolved relative name.

resolved_name must be big enough (`MAXPATHLEN`) to contain the fully resolved path name.

RESULT

If there is no error, `realpath()` returns a pointer to the *resolved_name*. Otherwise it returns a null pointer and places the name of the offending file in *resolved_name*. The global variable `errno` is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

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 *path*.

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.

NOTES

`realpath()` operates on null-terminated strings.

One should have execute permission on all the directories in the given and the resolved path.

`realpath()` may fail to return to the current directory if an error occurs.

SEE ALSO

`getcwd(3C)`, `sysconf(3C)`, `stdlib(5)`.

NAME
regcomp, regexexec, regerror, regfree - regular expression matching

SYNOPSIS

```
#include <sys/types.h>
#include <regex.h>
int regcomp(regex_t *preg, const char *pattern, int cflags);
int regexexec(const regex_t *preg, const char *string, size_t nmatch,
regmatch_t pmatch[], int eflags);
size_t regerror(int errcode, const regex_t *preg, char *errbuf,
size_t errbuf_size);
void regfree(regex_t *preg);
```

DESCRIPTION

These functions interpret basic and extended regular expressions as described in [expressions\(5\)](#). If the functions are not supported, errno is set to ENOSYS and the return value REG_ENOSYS (from regcomp()) and regexexec()) or 0 (from regerror()) is reported. The structure type regex_t contains at least the following member:

Type	Name	Description
size_t	re_nsub	Number of parenthesised subexpressions

The structure type reg_match contains at least the following members:

Type	Name	Description
regoff_t	rm_so	Byte offset from start of <i>string</i> to start of substring
regoff_t	rm_eo	Byte offset from start of <i>string</i> to the first character after the end of substring

The regcomp() function will compile the regular expressions 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 regexexec().
- REG_NEWLINE
Change the handling of newline characters, as described in the text.
- REG_LIT_NL
The \n string is treated as a newline character.
- REG_VIMODE
<< or >> are interpreted as the start or the finish of a word.
- REG_NSUB_ANCHOR
The characters ^ and \$ do not function as anchors within subexpressions.
- REG_EMPTY

`regcomp()` returns an error value if *pattern* is a null string or empty.

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. The following must be remembered when formulating the regular expressions

- Equivalence classes can be used in the range specifications.
- Range specifications linked together with the form `[b-d-f]` are not possible. In this case, `-` and `f` are treated as normal characters.
- `{` in simple, and `{ {` in extended regular expressions are treated as normal characters and do not initiate a repeat in this case.
- A repeat statement at the start of a regular expression is treated as a normal character.
- `^` at the start or `$` at the end of a subexpression works as an anchor, unless the `REG_NSUB_ANCHOR` flag is set.

On successful completion, it returns zero; otherwise it returns non-zero, and the content of *preg* is undefined.

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*.

The `regexec()` function compares the null-terminated string specified by *string* with the compiled regular expression *preg* initialized by a previous call to `regcomp()`. If it finds a match, `regexec()` returns zero, otherwise it returns non-zero indicating either no match or an error. 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* substrings, 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 the 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 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 `{ }` appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched zero times)
 - `|` 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 `regexexec()` 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`.

The `regfree()` function frees any memory allocated by `regcomp()` associated with *preg*.

The following constants are defined as error return values:

`REG_NOMATCH`
 `regexexec()` failed to match.

`REG_INVARG`
 An invalid parameter specified.

`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.

REG_EPATTERN

A null or empty pattern was specified as a parameter and `regcomp()` was called with the `REG_EMPTY` flag.

The `regerror()` function provides a mapping from error codes returned by `regcomp()` and `regexexec()` 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 `regexexec()` with the given value of `preg`. If `errcode` is not such a value, the content of the generated string is unspecified.

If `preg` is a null pointer, but `errcode` is a value returned by a previous call to `regexexec()` or `regcomp()`, the `regerror()` still generates an error string corresponding to the value of `errcode`, but might not be as detailed under some implementations.

If the `errbuf_size` argument is not zero, `regerror()` will place the generated string into the buffer of size `errbuf_size` 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 `regexexec()` or `regcomp()` 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()`.

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. 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. Use the `fnmatch()` function for matching a pattern with the pattern matching notation.

RESULT

On successful completion, the `regcomp()` function returns zero. 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 `regexexec()` function returns zero. 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 zero to indicate that the function is not implemented.

The `regfree()` function returns no value.

EXAMPLES

```
#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 = regexexec (&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 */
    error = regexec (&re, buffer + pm.rm:eo, 1, &pm, REG_NOTBOL);
}

```

NOTES

If you use one of these functions, you must link the `libgen` library at compilation (`cc -lgen`).

SEE ALSO

`regcmp(1)`, `fnmatch(3C)`, `glob(3C)`, `regex(3C)`, `regcmp(3G)`, `regexpr(3G)`, `expressions(5)`, `regex(5)`, `regexp(5)`.

NAME

regex:
 re_comp, re_exec - regular expression handler

SYNOPSIS

```
#include <re_comp.h>
char *re_comp(const char *s);
int re_exec(const char *s);
```

DESCRIPTION

re_comp() compiles a string into an internal form suitable for pattern matching. re_exec() checks the argument string against the last string passed to re_comp().

re_comp() returns a NULL pointer if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If re_comp() is passed 0 or a NULL string, it returns without changing the currently compiled regular expression.

re_exec() returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and -1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both re_comp() and re_exec() may have trailing or embedded NEWLINE characters; they are terminated by NULL characters. The regular expressions recognized are described in the manual entry for [ed\(1\)](#), given the above difference.

RETURN VALUE

re_exec() returns -1 for an internal error.

ERRORS

re_comp() returns one of the following strings if an error occurs:

```
No previous regular expression
Regular expression too long
unmatched \(
missing ]
too many \(\) pairs
unmatched \)
```

SEE ALSO

[ed\(1\)](#), [ex\(1\)](#), [grep\(1\)](#), [regcmp\(1\)](#), [regcomp\(3C\)](#), [regcmp\(3G\)](#), [regexpr\(3G\)](#), [expressions\(5\)](#), [re_comp\(5\)](#), [regex\(5\)](#), [regexp\(5\)](#).

remove(3C)**NAME**

`remove` - `remove` file

SYNOPSIS

```
#include <stdio.h>
int remove(const char *path);
```

DESCRIPTION

`remove()` 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.

RESULT

Upon successful completion, `remove()` returns a value of 0; otherwise, it returns a value of -1 and sets `errno` to indicate an error.

SEE ALSO

[rmdir\(2\)](#), [unlink\(2\)](#), [stdio\(5\)](#).

seclib(3C) - syslog(3C)**seclib(3C)****NAME**

`seclib` - security library

SYNOPSIS

```
#include <seclib.h>
cc ... -lseclib
```

DESCRIPTION

The security library provides a common set of interfaces to provide all system Identification and Authentication (I&A) functions. The functions take as an argument a login context structure. This structure contains all the session information required for the I&A of a user.

The library will automatically check to see if the CSP (Commercial Security Package) product has been installed. If it has, it will make the CSP features available to the caller automatically. The library is composed of several functional groups. The groups are:

Login Context Group

The Login Context group is responsible for allocating and deallocating the login context structure that is used by the other security library API calls.

See [seclib_logincontext\(3C\)](#) for more details.

Password Entry Group

The Password Entry Group is responsible for all user interaction of prompting for and typing in passwords. It will check the password for validity, check for various invalid password conditions, read in the "real" user name and password as required and read in a token card password if required. The Password Entry Group is also responsible for updating the `lastlog` file, preparing audit records and

using `syslog(3C)` to log success and failure.

See `seclib_passentry(3C)` for more details.

Password Qualifier Group

The Password Qualifier Group is responsible for the checking of a user password entry (in `/etc/shadow`) for expiration and idle time validity. The qualifier group will also check to make sure that a user is a member of the appropriate groups to log in.

See `seclib_passqual(3C)` for more details.

Secure Terminal Group

The Secure Terminal Group is responsible for checking that the root user is only logging in (or using `su`) at a secure location.

See `seclib_secterm(3C)` for more details.

Autologin Group

The Autologin Group is responsible for the remote-command style of automatic login. This autologin mechanism uses (among others) the `/etc/host` and `$HOME/.rhost` files.

See `seclib_ruserok(3C)` for more details.

Password Setting Group

The Password Setting Group is responsible for giving the user the opportunity to set a new password when their old one has expired or has not been set.

See `seclib_passwd(3C)` for more details.

Time Windows Group

The Time Windows Group is responsible for ensuring that the user only logs in on authorized terminals and at authorized times.

See `seclib_twindows(3C)` for more details.

Miscellaneous Group

The Miscellaneous Group provides indirect interfaces to obtain the proper `umask` and `ulimit` for a process. It also provides access to the auditing system (for CSP) and to the `lastlog` file.

See `seclib_misc(3C)` for more details.

FILES

`/var/adm/lastlog`

SEE ALSO

`seclib_logincontext(3C)`, `seclib_misc(3C)`, `seclib_passentry(3C)`, `seclib_passqual(3C)`, `seclib_passwd(3C)`, `seclib_ruserok(3C)`, `seclib_secterm(3C)`, `seclib_twindows(3C)`.

seclib_logincontext(3C)**NAME**

seclib_logincontext - allocating and deallocating the login context structure

SYNOPSIS

```
cc ... -lseclib
#include <seclib.h>
enum seclib_returnval
seclib_logincontext(const char *pgmname,
                   const char *user,
                   const char *tty,
                   const char *host,
                   void (*messagecb)(const char *),
                   char *(*promptcb)(const char *),
                   char *(*gpass)(const char *),
                   struct logincontext **context);
enum seclib_returnval
seclib_freelogincontext(struct logincontext *context);
```

DESCRIPTION

The Login Context group is responsible for allocating and deallocating the login context structure that is used by the other security library API calls.

The `seclib_logincontext()` call will:

1. If the context argument is not null:
 - `seclib_logincontext()` will save (and later restore) the values of `tries` and of `callerprivate` from the passed context structure. If the `context->reason` is `SECX_TOO_MANY_QUIET`, the value of `reason` will also be saved and restored.
 - `seclib_freelogincontext` will be called with the passed address.
2. The `calloc()` function will be called to allocate space for the new context structure.
3. The following fields in the context structure will be initialized (*NOTE*: all initialized strings and structures are placed in newly allocated storage):

FIELD**VALUE**

<code>tries</code>	from the saved value, otherwise zero
<code>private</code>	from the saved value, otherwise NULL
<code>reason</code>	from the saved value, otherwise <code>SECX_OK</code>
<code>alog_reason</code>	<code>SECAX_FAILED</code>
<code>tok_reason</code>	<code>SECTX_FAILED</code>
<code>rtok_reason</code>	<code>SECTX_FAILED</code>
<code>tty</code>	from the input arg, less the heading <code>/dev/</code>
<code>uname</code>	

from the input arg
 rmt_{host}
 from the input arg
 pgmname
 from the input arg
 pwd from `getpwnam(uname)`
 sp from `getspnam(uname)`
 grp from `getgrgid(pwd->pw_gid)`
 groups
 NULL terminated list. All groups the user is in (from the `/etc/group` file)
 ngroups
 number of groups in the "groups" list (above)
 lastlog
 lastlog entry for `uname` (prev* fields zeroed)
 messagecb
 from the input arg
 promptcb
 from the input arg
 gpasscb
 from the input arg
 umask
 from `/etc/default/login` (if set, `DEF_UMASK` otherwise)
 ulimit
 from `/etc/default/login` (if set, `DEF_ULIMIT` otherwise)

4. If there were any memory allocation or system call errors, the storage that was allocated will be freed and `SECR_CALLFAILED` will be returned.
5. If some of the user information was not available (for instance, the user was not found in the `password` or `shadow` file), the reason will be set to an appropriate error return and as much of the structure as possible will be filled out.
6. `SECR_OK` will be returned.

The `seclib_freelogincontext()` call will free all storage used by a login context structure. Any pointer field within the structure that is not null will have its space freed. Special care will be taken with certain structures which may point to multiple bits of allocated data.

ARGUMENTS

logincontext
 The address of the newly allocated login context structure will be placed in the passed pointer `logincontext`.

pgmname
 The name of the calling program that should be used in calls to the audit system. This name is generally in all upper case, e.g. "LOGIN", "RSHD", or "FTPD".

user The name of the user to be authenticated.

tty The tty port that the session is to be started on. This argument may be `NULL`.

host The name of the remote host that requested the service. This value (if it is not `NULL`) is generally returned via the `gethostbyaddr(3N)` call.

messagecb
 A callback that various security library functions may call when it is necessary to display information on the user's terminal.

promptcb

A callback that various security library functions may call when it is necessary to display a message on the user's terminal, and return a response.

`gpasscb`

A callback that various security library functions may call when it is necessary to display a message on the user's terminal, and return a response. The response string that the user types in should not be echoed.

RETURN VALUE

The functions return the `seclib_returnval` enumeration. The functions will return:

`SECR_OK`

The function completed processing without encountering a system error.

NOTE: The function will return `SECR_OK` even if it sets `context->reason` to a failure value.

`SECR_CALLFAILED`

The function call failed. This is usually because a system call failed (such as a `malloc()` failing when a `brk()` call returned with an error).

`SECR_BADOPTION`

The function was called with an invalid argument.

SEE ALSO

`seclib(3C)`, `seclib_misc(3C)`, `seclib_passentry(3C)`, `seclib_passqual(3C)`,
`seclib_passwd(3C)`, `seclib_ruserok(3C)`, `seclib_scterm(3C)`, `seclib_twindows(3C)`.

NAME

seclib_misc - miscellaneous security library functions

SYNOPSIS

```
cc ... -lseclib
#include <seclib.h>
enum seclib_returnval
seclib_getflag(enum seclib_flag flag, void (*value));
enum seclib_returnval
seclib_setdefaultfile(char *filename);
int
seclib_is_csp_installed(void);
char *
seclib_geterror(struct logincontext *context);
enum seclib_returnval
seclib_duplastlog(const struct lastlog *orig,
struct lastlog **copy);
enum seclib_returnval
seclib_freelastlog(struct lastlog *freeme);
enum seclib_returnval
seclib_read_lastlog(struct logincontext *);
enum seclib_returnval
seclib_write_lastlog(struct logincontext *);
int
seclib_rw_lastlog(uid_t uid, int writeflag, struct lastlog *ll);
enum seclib_returnval
seclib_duppass(const struct passwd *orig, struct passwd **copy);
enum seclib_returnval
seclib_dupshad(const struct spwd *orig, struct spwd **copy);
enum seclib_returnval
seclib_dupgroup(const struct group *orig, struct group **copy);
enum seclib_returnval
seclib_freepass(struct passwd *freeme);
enum seclib_returnval
seclib_freeshad(struct spwd *freeme);
enum seclib_returnval
seclib_freegroup(struct group *freeme);
```

DESCRIPTION

The Miscellaneous group provides the following functionality:

- convenient access to flag values from the `/etc/default/login` file.
- duplication and free of various data structures.
- access to the `/var/adm/lastlog` file.
- composition of error messages from the context reason codes.
- detection of the installation of the CSP product.

`seclib_getflag`
 Reads a flag value from `/etc/default/login`.

`seclib_setdefaultfile`
 Specifies an alternate file for the `seclib_getflag()` function.

`seclib_is_csp_installed`
 Returns the true value (1) if the CSP product is installed, false (0) otherwise.

`seclib_geterror`
 Returns an allocated string which contains a textual message describing the success/failure reasons in the login context.

`seclib_read_lastlog`
`seclib_write_lastlog`
`seclib_rw_lastlog`
 Read/write the `/var/adm/lastlog` file.

`seclib_duppass`
`seclib_dupshad`
`seclib_dupgroup`
`seclib_duplastlog`
 Duplicate the given structure.

`seclib_freepass`
`seclib_freeshad`
`seclib_freegroup`
`seclib_freelastlog`
 Free the storage allocated for the given structure.

ARGUMENTS

`logincontext`
 A value as returned by `seclib_logincontext()`. The structure referenced by this pointer contains all of the session information required to authenticate a user via the security library.

`enum seclib_flag flag`
 The value in `/etc/default/login` that is to be looked up and returned.

`void (*value)`
 A pointer into which the value of the flag (from `/etc/default/login`) will be placed. If this pointer has a NULL value, it will be ignored.

`char *filename`
 An alternate file to be used for `seclib_getflag()`. If NULL is passed, `/etc/default/login` will be used.

`struct lastlog *freeme`
`struct passwd *freeme`
`struct group *freeme`
`struct spwd *freeme`
 A pointer to the (given) structure to be freed.

`struct lastlog *orig`
`struct passwd *orig`
`struct group *orig`
`struct spwd *orig`
 Pointers to the (given) structure to be copied.

`struct lastlog **copy`
`struct passwd **copy`
`struct group **copy`
`struct spwd **copy`
 Pointers to the (given) structure pointers. The allocated new structure copy address will be placed in the pointer.

`struct lastlog *ll`

A pointer to a `lastlog` structure.

`uid_t uid`

The user id for which a `lastlog` structure is to be read or written.

`int writeflag`

A flag indicating whether the `lastlog` structure would be read or written (1==write, 0==read).

RETURN VALUE

Most functions return the `seclib_returnval` enumeration. These functions will return:

`SECR_OK`

The function completed processing without encountering a system error. `SECR_OK` is also returned by `seclib_getflag` when a boolean "yes" value for a flag is read.

NOTE: the function will return `SECR_OK` even if it sets `context->reason` to a failure value.

`SECR_FALSE`

The `seclib_getflag` function will return `SECR_FALSE` when a boolean "no" value for a flag is read.

`SECR_CALLFAILED`

The function call failed. This is usually because a system call failed (such as a `malloc()` failing when a `brk()` call returned with an error).

`SECR_BADOPTION`

The function was called with an invalid argument.

FILES

`/var/adm/lastlog`

SEE ALSO

`seclib(3C)`, `seclib_logincontext(3C)`, `seclib_passentry(3C)`, `seclib_passqual(3C)`,
`seclib_passwd(3C)`, `seclib_ruserok(3C)`, `seclib_secterm(3C)`, `seclib_twindows(3C)`.

seclib_passentry(3C)**NAME**

seclib_passentry - password entry and checking functions

SYNOPSIS

```
cc ... -lseclib
#include <seclib.h>
enum seclib_returnval
seclib_passentry(enum secentry_options, struct logincontext *);
enum seclib_returnval
seclib_passentry_ckfailed(enum secentry_options,
struct logincontext *);
```

DESCRIPTION

The Password Entry Group is responsible for all user interaction of prompting for and typing in passwords. It will check the password for validity, check for various invalid password conditions, read in the "real" user name and password as required and read in a tokencard password if required. The Password Entry Group is also responsible for updating the lastlog file, preparing audit records and using [syslog\(3C\)](#) to log success and failure.

The seclib_passentry() function will:

- SECE_CKPASS
Prompt the user for their password, verify it.
- SECE_DIALPASS
Prompt the user for a dial-in password, verify it.
- SECE_CKRPASS (CSP feature)
Prompt the user for their "real user" password, verify it.
- SECE_GETREALNAME (CSP feature)
Prompt the user for a "real user name" if they are logging in as the root user. Verify their "real user" password.
- SECE_TOKENCHECK (CSP and Tokencard feature)
Attempt to authenticate the user via a token card.

The seclib_passentry_ckfailed() function will:

- SECE_LASTLOG
Update the lastlog file, but save the old lastlog information.
- SECE_LOGINDELAY
Sleep a while (LOGINDELAY)
- SECE_MAXTRIES
Limit the user to MAXTRIES login attempts (zero for MAXTRIES means no limit)
- SECE_SYSLOG
[syslog\(3C\)](#) success and failure of the access request
- SECE_TIMEOUT
If the user takes too long to supply a password, terminate the attempt.
- SECE_AUDIT (CSP feature)
Audit success and failure of the access request.
- SECE_LOCKTTY (CSP feature)

If too many incorrect attempts were made on this tty port, lock it.

— SECE_LOCKUSER (CSP feature)

If too many incorrect attempts on behalf of a particular user, lock the user's password.

ARGUMENTS

`secentry_options`

A bitmask that directs the function to enable particular features. The complete list of values in the bitmask may be found in the `seclib.h` header.

`logincontext`

A value as returned by `seclib_logincontext()`. The structure referenced by this pointer contains all of the session information required to authenticate a user via the security library.

RETURN VALUE

The functions return the `seclib_returnval` enumeration. The functions will return:

`SECR_OK`

The function completed processing without encountering a system error.

NOTE: the function will return `SECR_OK` even if it sets `context->reason` to a failure value.

`SECR_CALLFAILED`

The function call failed. This is usually because a system call failed (such as a `malloc()` failing when a `brk()` call returned with an error).

`SECR_BADOPTION`

The function was called with an invalid argument.

SEE ALSO

`seclib(3C)`, `seclib_logincontext(3C)`, `seclib_misc(3C)`, `seclib_passqual(3C)`,
`seclib_passwd(3C)`, `seclib_ruserok(3C)`, `seclib_secterm(3C)`, `seclib_twindows(3C)`.

seclib_passqual(3C)**NAME**

`seclib_passqual` - supplemental checking of a user password entry

SYNOPSIS

```
cc ... -lseclib
#include <seclib.h>
enum seclib_returnval
seclib_passqual(enum secqual_options, struct logincontext *);
```

DESCRIPTION

The Password Qualifier Group is responsible for supplemental checking of a user password entry. It will check:

- `SECQ_EXPIRE`
the user must not have an expired login
- `SECQ_IDLE`
the user must not have been idle too long
- `SECQ_INVPASS`
the user must not have a locked or invalid password
- `SECQ_CKGROUP` (CSP feature)
a user must be a member of their group
- `SECQ_CKRGROUP` (CSP feature)
a user must be a member of the root group if he attempts to login or `su` to root
- `SECQ_PASSREQ` (CSP feature)
the user must not have a null password

ARGUMENTS

`secqual_options`

A bitmask that directs the function to enable particular features. The complete list of values in the bitmask may be found in the `seclib.h` header.

`logincontext`

A value as returned by `seclib_logincontext()`. The structure referenced by this pointer contains all of the session information required to authenticate a user via the security library.

RETURN VALUE

The functions return the `seclib_returnval` enumeration. The functions will return:

`SECR_OK`

The function completed processing without encountering a system error.

NOTE: the function will return `SECR_OK` even if it sets `context->reason` to a failure value.

`SECR_CALLFAILED`

The function call failed. This is usually because a system call failed (such as a `malloc()` failing when a `brk()` call returned with an error).

`SECR_BADOPTION`

The function was called with an invalid argument.

SEE ALSO

[seclib\(3C\)](#), [seclib_logincontext\(3C\)](#), [seclib_misc\(3C\)](#), [seclib_passentry\(3C\)](#),

`seclib_passwd(3C)`, `seclib_ruserok(3C)`, `seclib_scterm(3C)`, `seclib_twindows(3C)`.

seclib_passwd(3C)**NAME**

`seclib_passwd` - set a new user password as required

SYNOPSIS

```
cc ... -lseclib
#include <seclib.h>
enum seclib_returnval
seclib_passwd(enum secpass_options, struct logincontext *);
```

DESCRIPTION

The Password Setting Group is responsible for giving the user the opportunity to set a new password when their old one has expired or has not been set. The `seclib_passwd()` function checks for the following:

- `SECP_EXPIRED`
If the user's password is too old, but not to "very" old, change it. If the user's password is too "very" old, deny login.
- `SECP_PASSREQ`
If the user's password is `NULL`, set it.
- `SECP_WARNEXPIRE`
If the user's password will soon expire, warn them.

ARGUMENTS

`secpass_options`

A bitmask that directs the function to enable particular features. The complete list of values in the bitmask may be found in the `seclib.h` header.

`logincontext`

A value as returned by `seclib_logincontext()`. The structure referenced by this pointer contains all of the session information required to authenticate a user via the security library.

RETURN VALUE

The functions return the `seclib_returnval` enumeration. The functions will return:

`SECR_OK`

The function completed processing without encountering a system error.

NOTE: the function will return `SECR_OK` even if it sets `context->reason` to a failure value.

`SECR_CALLFAILED`

The function call failed. This is usually because a system call failed (such as a `malloc()` failing when a `brk()` call returned with an error).

`SECR_BADOPTION`

The function was called with an invalid argument.

SEE ALSO

[seclib\(3C\)](#), [seclib_logincontext\(3C\)](#), [seclib_misc\(3C\)](#), [seclib_passentry\(3C\)](#),
[seclib_passqual\(3C\)](#), [seclib_ruserok\(3C\)](#), [seclib_secterm\(3C\)](#), [seclib_twindows\(3C\)](#).

NAME

seclib_ruserok - checking automatic login

SYNOPSIS

```
cc ... -lseclib
#include <seclib.h>
enum seclib_returnval
seclib_ruserok(enum secato_options, char *ruser,
               struct logincontext *);
```

DESCRIPTION

The Autologin Group is responsible for the remote-command style of automatic login. This autologin mechanism uses (among others) the `/etc/host` and `$HOME/.rhost` files.

In addition to the normal `ruserok` functionality, the `seclib_ruserok()` function will:

- **SECA_AUTOLOGIN**
Master switch for autologin functionality. If the `STRICT_RHOST_MODES` flag is set in `/etc/default/login`, the autologin will be disallowed if the `$HOME/.rhost` file is not mode 600 and owned by the user -- or owned by root.
- **SECA_EQUIVHOSTS**
Allow autologin if the remote machine is in the `/etc/hosts.equiv` file.
- **SECA_DENYRHOSTS (CSP feature)**
Disallow autologin if the remote machine is in the `/etc/hosts.deny` file
- **SECA_DISABLERHOST (CSP feature)**
Disallow use of the `$HOME/.rhosts` file.
- **SECA_MINIMUMUID (CSP feature)**
Disallow autologin if the user's uid is below the `MINIMUMUID` set in `/etc/default/login`.

ARGUMENTS

`secato_options`

A bitmask that directs the function to enable particular features. The complete list of values in the bitmask may be found in the `seclib.h` header.

`logincontext`

A value as returned by `seclib_logincontext()`. The structure referenced by this pointer contains all of the session information required to authenticate a user via the security library.

`ruser`

The name of the user on the remote host that is requesting access.

RETURN VALUE

The functions return the `seclib_returnval` enumeration. The functions will return:

SECR_OK

The function completed processing without encountering a system error.

NOTE: the function will return `SECR_OK` even if it sets `context->reason` to a failure value.

SECR_CALLFAILED

The function call failed. This is usually because a system call failed (such as a `malloc()` failing when a `brk()` call returned with an error).

SECR_BADOPTION

The function was called with an invalid argument.

SEE ALSO

`seclib(3C)`, `seclib_logincontext(3C)`, `seclib_misc(3C)`, `seclib_passentry(3C)`,
`seclib_passqual(3C)`, `seclib_passwd(3C)`, `seclib_secterm(3C)`, `seclib_twindows(3C)`,
`hosts.equiv(4)`.

seclib_scterm(3C)**NAME**

`seclib_scterm` - the root user may only log in at a secure location

SYNOPSIS

```
cc ... -lseclib
#include <seclib.h>
enum seclib_returnval
seclib_term(enum scterm_options options,
            struct logincontext *context);
```

DESCRIPTION

The Secure Terminal Group is responsible for checking that the root user is only logging in (or using su) at a secure location. It will:

- `SECT_CONSOLE`
- `SECT_SECURETTY` (CSP feature) If the context is for the root or `sso` ("System Security Officer") user, this function will check to ensure that the user is logging in on a terminal that is in the `/etc/securetty` file or the terminal that is specified by the `CONSOLE` flag in `/etc/default/login`.

ARGUMENTS

`secentry_options`

A bitmask that directs the function to enable particular features. The complete list of values in the bitmask may be found in the `seclib.h` header.

`logincontext`

A value as returned by `seclib_logincontext()`. The structure referenced by this pointer contains all of the session information required to authenticate a user via the security library.

RETURN VALUE

The functions return the `seclib_returnval` enumeration. The functions will return:

`SECR_OK`

The function completed processing without encountering a system error.

NOTE: the function will return `SECR_OK` even if it sets `context->reason` to a failure value.

`SECR_CALLFAILED`

The function call failed. This is usually because a system call failed (such as a `malloc()` failing when a `brk()` call returned with an error).

`SECR_BADOPTION`

The function was called with an invalid argument.

SEE ALSO

[seclib\(3C\)](#), [seclib_logincontext\(3C\)](#), [seclib_misc\(3C\)](#), [seclib_passentry\(3C\)](#),
[seclib_passqual\(3C\)](#), [seclib_passwd\(3C\)](#), [seclib_ruserok\(3C\)](#), [seclib_twindows\(3C\)](#).

seclib_twindows(3C)**NAME**

`seclib_twindows` - the user may only log in on authorized terminals and at authorized times

SYNOPSIS

```
cc ... -lseclib
#include <seclib.h>
enum seclib_returnval
seclib_twindows(enum secwin_options, struct logincontext *);
char **
dup_priv_list(char **list);
void
free_priv_list(char **list);
```

DESCRIPTION

The Time Windows Group is responsible for ensuring that the user only logs in on authorized terminals and at authorized times.

`seclib_twindows` will:

- **SECW_CKAUTH (CSP feature)**
Use `getauth()` and `ptime()` functions to determine if the current time, `tty` is valid for the user.
- **SECW_ADDPRIV (CSP feature)**
The supplemental groups specified for the user in the `/etc/auth` file will be added to the list of group in the context structure.
- **SECW_KILLTIME (CSP feature)**
The time to kill the session (from the `/etc/t_windows` file) will be placed in the context structure.

`dup_priv_list` and `free_priv_list` will:

- Copy and free an array of character strings. Both the strings will be copied (freed) and the array itself will be copied (freed).

ARGUMENTS

`secentry_options`

A bitmask that directs the function to enable particular features. The complete list of values in the bitmask may be found in the `seclib.h` header.

`logincontext`

A value as returned by `seclib_logincontext()`. The structure referenced by this pointer contains all of the session information required to authenticate a user via the security library.

`list` An array of character strings.

RETURN VALUE

The `seclib_twindows` returns the `seclib_returnval` enumeration. The function will return:

SECR_OK

The function completed processing without encountering a system error.

NOTE: the function will return **SECR_OK** even if it sets `context->reason` to a failure value.

SECR_CALLFAILED

The function call failed. This is usually because a system call failed (such as a `malloc()` failing when a `brk()` call returned with an error).

SECR_BADOPTION

The function was called with an invalid argument.

The `dup_priv_list()` function will return a pointer to a newly allocated copy of its argument. It will return `NULL` on failure.

SEE ALSO

`seclib(3C)`, `seclib_logincontext(3C)`, `seclib_misc(3C)`, `seclib_passentry(3C)`,
`seclib_passqual(3C)`, `seclib_passwd(3C)`, `seclib_ruserok(3C)`, `seclib_secterm(3C)`.

NAME

`select`, `FD_CLR`, `FD_ISSET`, `FD_SET`, `FD_ZERO` - synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/time.h>
#include <sys/types.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *errorfds, struct timeval *timeout);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

DESCRIPTION

The `select()` function indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending. If the specified condition is false for all of the specified file descriptors, `select()` blocks, up to the specified timeout interval, until the specified condition is true for at least one of the specified file descriptors.

The `select()` function supports regular files, terminal and pseudo-terminal devices, STREAMS-based files, FIFOs and pipes. The behavior of `select()` 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 `select()` function tests file descriptors in the range of 0 to `nfds-1`.

If the `readfds` 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 `writefds` 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.

On successful completion, the objects pointed to by the `readfds`, `writefds`, and `errorfds` arguments are modified to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively. 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 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.

The use of a timeout does not affect any pending timers set up by `alarm()`, `ualarm()` or `settimer()`.

On successful completion, the object pointed to by the `timeout` argument may be modified.

Implementations may place limitations on the maximum timeout interval supported. On all implementations, the maximum timeout interval supported will be at least 31 days. If the `timeout` argument specifies a timeout

interval greater than the implementation-dependent maximum value, the maximum value will be used as the actual timeout value. Implementations may also place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval will be rounded up to the next supported value.

If the *readfs*, *writes*, and *errorfs* arguments are all null pointers and the *timeout* argument is not a null pointer, *select()* blocks for the time specified, or until interrupted by a signal. If the *readfs*, *writes*, and *errorfs* 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*, *writes*, and *errorfs* 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*, *writes*, and *errorfs* arguments have all bits set to 0.

File descriptor masks of type *fd_set* can be initialized. It is unspecified whether each of these is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with any of these names, the behavior is undefined.

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*.

RETURN VALUE

FD_CLR(), *FD_SET()* and *FD_ZERO()* return no value. *FD_ISSET()* 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.

On successful completion, *select()* returns the total number of bits set in the bit masks. Otherwise, *-1* is returned, and *errno* is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Under the following conditions, *select()* fails and sets *errno* to:

EBADF

One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.

EINTR

The *select()* function was interrupted before any of the selected events occurred and before the timeout interval expired.

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.

SEE ALSO

`fcntl(2)`, `poll(2)`, `read(2)`, `write(2)`, `sys_time(5)`, `types(5)`.

setjmp(3C)**NAME**

setjmp, longjmp - non-local goto

SYNOPSIS

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

DESCRIPTION

These functions enable errors and interrupts, encountered in a low-level subroutine of a program, to be handled.

A call to `setjmp()`, saves the calling environment in its `env` argument for later use by `longjmp()`.

It is unspecified whether `setjmp()` is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name `setjmp()` the behavior is undefined.

All accessible objects have values as of the time `longjmp()` was called, except that the values of objects of automatic storage duration which are local to the function containing the invocation of the corresponding `setjmp()` which do not have volatile-qualified type and which are changed between the `setjmp()` invocation and `longjmp()` call are indeterminate.

An invocation of `setjmp()` must appear in one of the following contexts only:

- the entire controlling expression of a selection or iteration statement
- one operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement
- the operand of a unary "!" operator with the resulting expression being the entire controlling expression of a selection or iteration
- the entire expression of an expression statement (possibly cast to `void`).

`longjmp()` restores the environment saved by the last call of `setjmp()` with the corresponding `env` argument. If there is no such invocation, or if the function containing the invocation of `setjmp()` has terminated execution in the interim, the behavior is undefined. It is unspecified whether `longjmp()` restores the signal mask, leaves the signal mask unchanged or restores it to its value at the time `setjmp()` was called.

All accessible objects have values as of the time `longjmp()` was called, except that the values of objects of automatic storage duration are indeterminate if they meet all the following conditions:

- They are local to the function containing the corresponding `setjmp()` invocation.
- They do not have volatile-qualified type.
- They are changed between the `setjmp()` invocation and `longjmp()` call.

As it bypasses the usual function call and return mechanisms, `longjmp()` will execute correctly in contexts of interrupts, signals and any of their associated functions. However, if `longjmp()` is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

The effect of a call to `longjmp()` where initialisation of the `jmp_buf` structure was not performed in the calling thread is undefined.

RETURN VALUE

If the return is from a direct invocation, `setjmp()` returns 0. If the return is from a call to `longjmp()`, `setjmp()` 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.

ERRORS

No errors are defined.

EXAMPLES

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
jmp_buf env;
int i = 0;
main ()
{
    void exit();
    if(setjmp(env) != 0) {
        (void) printf("value of i on 2nd return from setjmp:
            %d\n", i);
        exit(0);
    }
    (void) printf("value of i on 1st return from setjmp: %d\n", i);
    i = 1;
    g();                /* NOTREACHED */
}
g()
{
    longjmp(env, 1);    /* NOTREACHED */
}
```

If the `a.out` resulting from this C language code is run, the output will be:

```
value of i on 1st return from setjmp():0
value of i on 2nd return from setjmp():1
```

NOTES

Applications whose behavior depends on the value of the signal mask should not use `longjmp()` and `setjmp()`, since their effect on the signal mask is unspecified, but should instead use the following alternatives:

- The `_longjmp()` and `_setjmp()` functions (which never modify the signal mask)
- The `siglongjmp()` and `sigsetjmp()` functions (which can save and restore the signal mask under application control).

It is advisable to use the `sigsetjmp(3C)` function to examine errors and interrupts in a subroutine of a program.

SEE ALSO

`signal(2)`, `setjmp(3C-ucb)`, `sigsetjmp(3C)`, `setjmp(5)`.

setjmp(3C-ucb)
BSD Compatibility Package**NAME**`setjmp, longjmp, _setjmp, _longjmp` - non-localgoto**SYNOPSIS**

```

/usr/ucb/cc [flag ...] file ... -lucb
#include <setjmp.h>
int setjmp(jmp_buf env);
longjmp(jmp_buf env, int val);
int _setjmp(jmp_buf env);
_longjmp(jmp_buf env, int val);

```

DESCRIPTION

`setjmp` and `longjmp` are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

`setjmp` saves its stack environment in `env` for later use by `longjmp`. A normal call to `setjmp` returns zero. `setjmp` also saves the register environment. If a `longjmp` call will be made, the routine which called `setjmp` should not return until after the `longjmp` has returned control (see below).

`longjmp` restores the environment saved by the last call of `setjmp`, and then returns in such a way that execution continues as if the call of `setjmp` had just returned the value `val` to the function that invoked `setjmp`; however, if `val` were zero, execution would continue as if the call of `setjmp` had returned one. This ensures that a return from `setjmp` caused by a call to `longjmp` can be distinguished from a regular return from `setjmp`. The calling function must not itself have returned in the interim, otherwise `longjmp` will be returning control to a possibly non-existent environment. All memory-bound data have values as of the time `longjmp` was called. The CPU and floating-point data registers are restored to the values they had at the time that `setjmp` was called. But, because the `register` storage class is only a hint to the C compiler, variables declared as `register` variables may not necessarily be assigned to machine registers, so their values are unpredictable after a `longjmp`. This is especially a problem for programmers trying to write machine-independent C routines.

The `_longjmp()` and `_setjmp()` functions are identical to `longjmp()` and `setjmp()`, 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.

None of these functions save or restore any floating-point status or control registers.

EXAMPLE

The following code fragment indicates the flow of control of the `setjmp` and `longjmp` combination:

```

function declaration . . .
    jmp_buf my_environment;
    ...
    if (setjmp(my_environment)) {
        /* register variables have unpredictable values */
        code after the return from longjmp
    }
    ...
} else {
    /* do not modify register vars in this leg of code */
    this is the return from setjmp
    ...

```



```
}
```

APPLICATION 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()` a pointer to a buffer not created by `sigsetjmp()` 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()` and `sigsetjmp()` respectively.

NOTES

`setjmp` does not save the current notion of whether the process is executing on the signal stack. The result is that a `longjmp` to some place on the signal stack leaves the signal stack state incorrect.

On some systems `setjmp` also saves the register environment. Therefore, all data that are bound to registers are restored to the values they had at the time that `setjmp` was called. All memory-bound data have values as of the time `longjmp` was called. However, because the register storage class is only a hint to the C compiler, variables declared as register variables may not necessarily be assigned to machine registers, so their values are unpredictable after a `longjmp`. When using compiler options that specify automatic register allocation [see `cc(1)`], the compiler will not attempt to assign variables to registers in routines that call `setjmp`.

`longjmp` *never* causes `setjmp` to return zero.

SEE ALSO

`cc(1)`, `signal(2)`, `sigsetmask(3)`, `sigvec(3)`, `signal(3-ucb)`, `setjmp(3C)`, `setjmp(5)`.

setlocale(3C)**NAME**

setlocale - set/query a program's locale

SYNOPSIS

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

DESCRIPTION

setlocale() selects the appropriate piece of the program's locale as specified by the *category* and *locale* arguments, and may be used to change or query the program's entire locale or portions thereof. The value LC_ALL for *category* names the program's entire locale; other values for *category* name only a part of the program's locale. *category* 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 file.

LC_COLLATE

Affects the behavior of regular expressions and the collation functions [strcoll() and strxfrm()].

LC_CTYPE

Affects the behavior of regular expressions, character classification, character conversion functions [isdigit(), tolower(), etc.] and wide-character functions [such as mbtowc() and wctomb()].

LC_MESSAGES

Affects what strings are expected by commands and utilities as affirmative or negative responses, what strings are given by commands and utilities as affirmative or negative responses, and the content of messages [gettext(), catopen(), catclose(), and catgets()].

LC_MONETARY

Affects the monetary formatted information returned by localeconv() [see [localeconv\(3C\)](#)].

LC_NUMERIC

Affects the decimal-point character for the formatted input/output functions and the string conversion functions as well as the non-monetary formatting information returned by localeconv() [see [localeconv\(3C\)](#)].

LC_TIME

Affects the behavior of the time conversion functions [asctime(), cftime(), getdate() and strftime()].

Each category corresponds to a set of databases which contain the relevant information for each defined locale. The location of a database is given by the following path, /usr/lib/locale/locale/category, where *locale* and *category* are the names of locale and category, respectively. For example, the database for the LC_CTYPE category for the "German" locale would be found in /usr/lib/locale/german/LC_CTYPE.

The values "POSIX" or "C" for *locale* specify the default minimal environment.

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.
LC_CTYPE:	LC_CTYPE	LANG
LC_COLLATE:	LC_COLLATE	LANG
LC_TIME:	LC_TIME	LANG
LC_NUMERIC:	LC_NUMERIC	LANG
LC_MONETARY:	LC_MONETARY	LANG
LC_MESSAGES:	LC_MESSAGES	LANG

The locale state is common to all threads within a process.

RETURN VALUE

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. A composite locale is a string beginning with a / followed by the locale of each category separated by a /. If `setlocale()` fails to set the locale for any category, a null pointer is returned and the program's locale for all categories is not changed. Otherwise, *locale* is returned.

ERRORS

No errors are defined.

APPLICATION USAGE

The following code illustrates how a program can initialise the international environment for one language, while selectively modifying the program's locale such that regular expressions and string operations can be applied to text recorded in a different language:

```
setlocale(LC_ALL, "De");  
setlocale(LC_COLLATE, "Fr@dict");
```

Internationalised programs must call `setlocale()` to initiate a specific language operation. This can be done by calling `setlocale()` as follows:

```
setlocale(LC_ALL, "");
```

Changing the setting of `LC_MESSAGES` has no effect on catalogues that have already been opened by calls to `catopen()`.

FILES

```
/usr/lib/locale/C/LC_CTYPE  
/usr/lib/locale/C/LC_NUMERIC  
/usr/lib/locale/C/LC_TIME  
/usr/lib/locale/C/LC_COLLATE  
/usr/lib/locale/C/LC_MESSAGES  
/usr/lib/locale/locale/category  
files containing the locale specific information for each locale and category
```

SEE ALSO

`ctime(3C)`, `ctype(3C)`, `getdate(3C)`, `gettext(3C)`, `localeconv(3C)`, `mbtowc(3C)`, `strcoll(3C)`, `strftime(3C)`, `strtod(3C)`, `strxfrm(3C)`, `wctomb(3C)`, `printf(3S)`, `environ(5)`, `langinfo(5)`, `locale(5)`.

setregid(3C)**NAME**

`setregid` - set real and effective group IDs

SYNOPSIS

```
#include <unistd.h>
int setregid(gid_t rgid, gid_t egid);
```

DESCRIPTION

`setregid` is used to set the real and effective group IDs of the calling process. If *rgid* is `-1`, the real GID is not changed; if *egid* is `-1`, the effective GID is not changed. The real and effective GIDs may be set to different values in the same call.

If the effective user ID of the calling process is super-user, the real GID and the effective GID can be set to any legal value.

A non-privileged process can set either the real group ID to the saved set-group-ID from `exec*()`, or the effective group ID to the saved set-group-ID or the real group ID.

Any supplementary group IDs of the calling process remain unchanged.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `setregid()` function will fail if:

EINVAL

The value of the *rgid* or *egid* argument is invalid or out-of-range.

EPERM

The calling process' effective UID is not the super-user and a change other than changing the real GID to the saved set-GID, or changing the effective GID to the real GID or the saved GID, was specified.

SEE ALSO

[exec\(2\)](#), [getgroups\(2\)](#), [getuid\(2\)](#), [setuid\(2\)](#), [setreuid\(3C\)](#), [unistd\(4\)](#).

setreuid(3C)**NAME**

setreuid - set real and effective user IDs

SYNOPSIS

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
```

DESCRIPTION

`setreuid()` is used to set the real and effective user IDs of the calling process. If *ruid* is `-1`, the real user ID is not changed; if *euid* is `-1`, the effective user ID is not changed. The real and effective user IDs may be set to different values in the same call.

If the effective user ID of the calling process is super-user, the real user ID and the effective user ID can be set to any legal value.

If the effective user ID of the calling process is not super-user, either the real user ID can be set to the effective user ID, or the effective user ID can either be set to the saved set-user ID from `execv` or the real user ID.

Note: If a set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-user ID.

In either case, if the real user ID is being changed (that is, if *ruid* is not `-1`), or the effective user ID is being changed to a value not equal to the real user ID, the saved set-user ID is set equal to the new effective user ID.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in

[intro_prm2\(2\)](#) or in [errno\(5\)](#).

`setreuid` fails and neither of the user IDs is changed.

EINVAL

The value of the *ruid* or *euid* argument is invalid or out-of-range.

EPERM

The calling process' effective user ID is not the super-user and a change other than changing the real user ID to the effective user ID, or changing the effective user ID to the real user ID or the saved set-user ID, was specified.

SEE ALSO

[exec\(2\)](#), [getuid\(2\)](#), [setuid\(2\)](#), [setregid\(3C\)](#), [unistd\(4\)](#).

sigsetjmp(3C)**NAME**

`sigsetjmp`, `siglongjmp` - a non-local goto with signal state

SYNOPSIS

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savemask);
void siglongjmp(sigjmp_buf env, int val);
```

DESCRIPTION

Using these functions, errors and interrupts encountered in a low-level subroutine of a program can be handled.

A call to `sigsetjmp()` saves the calling environment in its `env` argument for later use by `siglongjmp()`. It is unspecified whether `sigsetjmp()` is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name `sigsetjmp()` the behavior is undefined.

If the value of the `savemask` argument is not 0, `sigsetjmp()` will also save the current signal mask of the calling thread as part of the calling environment.

The `siglongjmp()` function restores the environment saved by the most recent invocation of `sigsetjmp()` in the same thread, with the corresponding `sigjmp_buf` argument. If there is no such invocation, or if the function containing the invocation of `sigsetjmp()` has terminated execution in the interim, the behavior is undefined.

All accessible objects have values as of the time `sigsetjmp()` was called, except that the values of objects of automatic storage duration which are local to the function containing the invocation of the corresponding `sigsetjmp()` which do not have volatile-qualified type and which are changed between the `sigsetjmp()` invocation and `siglongjmp()` call are indeterminate.

As it bypasses the usual function call and return mechanisms, `siglongjmp()` will execute correctly in contexts of interrupts, signals and any of their associated functions. However, if `siglongjmp()` is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behaviour is undefined.

The `siglongjmp()` function will restore the saved signal mask if and only if the `env` argument was initialised by a call to `sigsetjmp` with a non-zero `savemask` argument.

The effect of a call to `siglongjmp()` where initialisation of the `jmp_buf` structure was not performed in the calling thread is undefined.

All accessible objects have values as of the time `siglongjmp()` was called, except that the values of objects of automatic storage duration which are local to the function containing the invocation of the corresponding `sigsetjmp()` which do not have volatile-qualified type and which are changed between the `sigsetjmp()` invocation and `siglongjmp()` call are indeterminate.

An invocation of `sigsetjmp()` must appear in one of the following contexts only:

- the entire controlling expression of a selection or iteration statement
- one operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement
- the operand of a unary operator with the resulting expression being the entire controlling expression of a selection or iteration statement
- the entire expression of an expression statement (possibly cast to `void`)

If a signal-catching function interrupts `sleep()` 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' signal mask is restored as part of the environment.

RETURN VALUE

If the return is from a successful direct invocation, `sigsetjmp()` returns 0. If the return is from a call to `siglongjmp()`, `sigsetjmp()` returns a non-zero value.

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 the value 1.

ERRORS

No errors are defined.

NOTES

If `siglongjmp()` is called even though *env* was never primed by a call to `sigsetjmp()`, or when the last such call was in a function that has since returned, absolute chaos is guaranteed.

SEE ALSO

`getcontext(2)`, `prctl(2)`, `sigaction(2)`, `sigaltstack(2)`, `sigprocmask(2)`, `setjmp(3C)`, `setjmp(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(const sigset_t *set, int signo);
```

DESCRIPTION

These functions manipulate `sigset_t` data types, representing the set of signals supported by the implementation.

`sigemptyset()` initializes the set pointed to by *set* to exclude all signals defined by the system.

`sigfillset()` initializes the set pointed to by *set* to include all signals defined by the system.

`sigaddset()` adds the individual signal specified by the value of *signo* to the set pointed to by *set*.

`sigdelset()` deletes the individual signal specified by the value of *signo* from the set pointed to by *set*.

`sigismember()` checks whether the signal specified by the value of *signo* is a member of the set pointed to by *set*.

Applications must call either `sigemptyset()` or `sigfillset()` at least once for each object of type `sigset_t` prior to any other use of that object. If such an object is not initialised in this way, but is nonetheless supplied as an argument to any of `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` or `sigprocmask()`, the results are undefined.

RETURN VALUE

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

ERRORS

`sigaddset()`, `sigdelset()` and `sigismember()` may fail if the following is true:

`EINVAL`

The value of the *signo* argument is not a valid signal number.

SEE ALSO

`sigaction(2)`, `sigprocmask(2)`, `sigpending(2)`, `sigsuspend(2)`, `signal(5)`.

sleep(3C)**NAME**

sleep - suspend execution

SYNOPSIS

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

DESCRIPTION

The `sleep()` function will cause the calling thread to be suspended from execution until either the number of real-time seconds specified by the argument *seconds* 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 due to the scheduling of other activity by the system.

If a `SIGALRM` signal is generated for the calling process during execution of `sleep()` and if the `SIGALRM` signal is being ignored or blocked from delivery, it is unspecified whether `sleep()` returns when the `SIGALRM` signal is scheduled. If the signal is being blocked, it is also unspecified whether it remains pending after `sleep()` returns or it is discarded.

If a `SIGALRM` signal is generated for the calling process during execution of `sleep()`, except as a result of a prior call to `alarm()`, and if the `SIGALRM` signal is not being ignored or blocked from delivery, it is unspecified whether that signal has any effect other than causing `sleep()` to return.

If a signal-catching function interrupts `sleep()` and examines or changes either the time a `SIGALRM` is scheduled to be generated, the action associated with the `SIGALRM` signal, or whether the `SIGALRM` signal is blocked from delivery, the results are unspecified.

If a signal-catching function interrupts `sleep()` and calls `siglongjmp()` or `longjmp()` to restore an environment saved prior to the `sleep()` call, the action associated with the `SIGALRM` signal and the time at which a `SIGALRM` signal is scheduled to be generated are unspecified. It is also unspecified whether the `SIGALRM` signal is blocked, unless the process' signal mask is restored as part of the environment.

Interactions between `sleep()` and any of `setitimer()`, `ualarm()` or `usleep()` are unspecified.

RETURN VALUE

If `sleep()` returns because the requested time has elapsed, the value returned will be 0. If `sleep()` returns because of premature arousal due to delivery of a signal, the return value will be the "unslept" amount (the requested time minus the time actually slept) in seconds.

ERRORS

No errors are defined.

SEE ALSO

`alarm(2)`, `pause(2)`, `sigaction(2)`, `signal(2)`, `wait(2)`, `ualarm(3)`, `usleep(3)`, `getitimer(3C)`, `sigsetjmp(3C)`, `unistd(4)`.

sleep(3C-ucb)
BSD Compatibility Package**NAME**

`sleep` - suspend execution for interval

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb  
sleep(seconds)  
unsigned seconds;
```

DESCRIPTION

`sleep` suspends the current process from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and may be an arbitrary amount longer because of other activity in the system.

`sleep` is implemented by setting an interval timer and pausing until it expires. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous value of the timer, the process sleeps only until the timer would have expired, and the signal which occurs with the expiration of the timer is sent one second later.

SEE ALSO

`sigpause(3)`, `usleep(3)`, `getitimer(3C)`.

ssignal(3C)**NAME**

`ssignal`, `gsignal` - software signals

SYNOPSIS

```
#include <signal.h>
int (*ssignal(int sig, int (*action) (int)) (int));
int gsignal(int sig);
```

DESCRIPTION

`ssignal()` and `gsignal()` implement a software facility similar to [signal\(2\)](#).

The software signals provided are associated with integers in the inclusive range 1 through 17. A call to `ssignal()` associates a procedure, *action*, with the software signal *sig*. *sig* is raised by a call to `gsignal()`. The effect is to execute the action established for that signal.

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) function, *action*, or one of the manifest constants `SIG_DFL` (default) or `SIG_IGN` (ignore). `ssignal()` 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()` raises the signal identified by its argument, *sig*:

- If an action has been established for *sig*, then that action is reset to `SIG_DFL` and the action is entered with argument *sig*. `gsignal()` returns the value returned to it by the action.
- 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.

SEE ALSO

[signal\(2\)](#), [sigset\(2\)](#), [raise\(3C\)](#), [signal\(5\)](#).

NAME

stdipc:
 ftok - standard interprocess communication package

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the `msgget(2)`, `semget(2)`, and `shmget(2)` system calls to obtain interprocess communication identifiers. One possible way of forming a key is to use the `ftok()` subroutine. 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 possible with *all* types of key resolution that two processes will coincidentally still use the same key. The user should therefore ensure that the most significant byte of a key in some sense refers to a project so that keys do not conflict across a given system.

`ftok()` returns a key based on *path* and *id* that is usable in subsequent `msgget()`, `semget()`, and `shmget()` system calls. *path* must be the path name of an existing file that is accessible to the process. *id* is a character that uniquely identifies a project. Note that `ftok()` will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same filename but different *id* identifiers.

The key is made up of the *id*, the minor number [cf. `makedev(3C)`] and the inode of the *path*. Only the eight low-order bits of the *id* and the minor number are significant. Only the sixteen low-order bits of the inode are significant. The behavior of `ftok()` is unspecified if these bits are 0.

RESULT

`ftok()` returns `(key_t)-1` if *path* does not exist or if it is not accessible to the process.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

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`.

NOTES

If the file whose path is passed to `ftok()` is removed when keys still refer to the file, future calls to `ftok()` with the same *path* and *id* will return an error. If the same file is recreated, then `ftok()` is likely to return a different key than it did the original time it was called.

SEE ALSO

`msgget(2)`, `semget(2)`, `shmget(2)`, `stat(2)`, `makedev(3C)`, `ipc(5)`, `types(5)`.

strcoll(3C)**NAME**

strcoll - string collation

SYNOPSIS

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

DESCRIPTION

The `strcoll()` function compares the string pointed to by `s1` to the string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

The `strcoll()` function will not change the setting of `errno` if successful.

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 `strcoll()`, then check `errno`.

RETURN VALUE

Upon successful completion, `strcoll()` returns an integer greater than, equal to or less than 0, according to whether the string pointed to by `s1` is greater than, equal to or less than the string pointed to by `s2` when both are interpreted as appropriate to the current locale. 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.

APPLICATION USAGE

The `strxfrm()` and `strcmp()` functions should be used for sorting large lists.

SEE ALSO

`colltbl(1M)`, `setlocale(3C)`, `string(3C)`, `strxfrm(3C)`, `environ(5)`, `string(5)`.

strerror(3C)**NAME**

`strerror` - get error message

SYNOPSIS

```
#include <string.h>
char *strerror(int errnum);
```

DESCRIPTION

The `strerror()` function maps the error number in *errnum* to a locale-dependent error message string and returns a pointer thereto. The string pointed to must not be modified by the program, but may be overwritten by a subsequent call to `strerror()` or `perror()`.

The contents of the error message strings returned by `strerror()` should be determined by the setting of the `LC_MESSAGES` category in the current locale.

The implementation will behave as if no function calls `strerror()`.

The `strerror()` function will not change the setting of `errno` if successful.

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 `strerror()`, then check `errno`.

This interface need not be thread-safe.

RETURN VALUE

Upon successful completion, `strerror()` returns a pointer to the generated message string. On error `errno` may be set, but no return value is reserved to indicate an error.

ERRORS

The `strerror()` function may fail if:

`EINVAL`

The value of *errnum* is not a valid error number.

SEE ALSO

`perror(3C)`, `string(5)`.

strfmon(3C)**NAME**

strfmon - convert monetary value to string

SYNOPSIS

```
#include <monetary.h>
ssize_t strfmon(char *s, size_t maxsize, const char□
*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
- arequired 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 locales equivalent of + and – are used (for example, in the USA: the empty string if positive and – if negative). 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 within 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 the formatting.

Conversion Characters

The conversion characters and their meanings are:

- `i` The `double` argument is formatted according to the locales international currency format (for example, in the USA: USD 1,234.56).
- `n` The `double` argument is formatted according to the locales national currency format (for example, in the USA: \$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 effects 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 compliant with the ISO 4217:1987 standard.

If the value of `maxsize` is greater than `SSIZE_MAX`, the result is implementation-dependent.

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than `maxsize`, the `strfmon()` function 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:

E2BIG Conversion stopped due to lack of space in the buffer.

EXAMPLES

Given a locale for the USA and the values 123.45, -123.45 and 3456.781:

Conversion Specification	Output	Comments
<code>%n</code>	\$123.45 - \$123.45	default formatting

	\$3,456.78	
%11n	\$123.45 -\$123.45 \$3,456.78	right align within an 11 character field
%#5n	\$123.45 -\$123.45 \$3,456.78	aligned columns for values up to 99,999
%=*#5n	\$***123.45 -\$***123.45 \$*3,456.78	specify a fill character
%=0#5n	\$000123.45 -\$000123.45 \$03,456.78	fill characters do not use grouping even if the fill character is a digit
%^#5n	\$123.45 -\$123.45 \$3456.78	disable the grouping separator
%^#5.0n	\$123 -\$123 \$3457	round off to whole units
Conversion Specification	Output	Comments
%^#5.4n	\$123.4500 -\$123.4500 \$3456.7810	increase the precision
%(#5n	123.45 (\$123.45) \$3,456.78	use an alternative pos/neg style
%!(#5n	123.45 (123.45) 3,456.78	disable the currency symbol

SEE ALSO

`localeconv(3C)`, `monetary(5)`.

strftime(3C)**NAME**

strftime, cftime, ascftime - convert date and time to string

SYNOPSIS

```
#include <time.h>

size_t strftime(char *s, size_t maxsize, const char *format,
    const struct tm *timeptr);

int cftime(char *s, char *format, const time_t *clock);

int ascftime(char *s, const char *format, const struct tm *timeptr);
```

DESCRIPTION

strftime(), ascftime(), and cftime() place characters into the array pointed to by *s* as controlled by the string pointed to by *format*. The *format* string consists of one or more conversion specifications and ordinary characters. All ordinary characters, including the terminating null character, are copied unchanged into the array. For strftime(), no more than *maxsize* characters are placed into the array.

If *format* is (char *)0, then the default format "%c" is used for strftime(), and for cftime() and ascftime() the default format "%C" [XPG3] is used. 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() [cf. LC_TIME(4)]:

%%	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 (year divided by 100 and truncated to an integer (00 – 99) [XPG4]; date and time representation as produced by date(1) [XPG3].
%d	Day of month (01 – 31).
%D	Date as %m/%d/%y.
%e	Day of month (1 – 31; single digits are preceded by a blank).
%f	Date and time representation as produced by date(1) [XPG4].
%h	Locale's abbreviated month name.
%H	Hour (00 – 23).
%I	Hour (01 – 12).
%j	Day number of year (001 – 366).
%m	Month number (01 – 12).
%M	Minute (00 – 59).
%n	Same as \n.
%p	Locale's equivalent of either AM or PM.
%r	Time as %I:%M:%S [AMPM].
%R	Time as %H:%M.

%S	Seconds (00 – 61), allows for leap seconds.
%t	Insert a tab.
%T	Time as %H:%M:%S.
%u	Weekday as a decimal number (1 – 7), with 1 representing Monday [XPG4].
%U	Week number of year (00 – 53), Sunday is the first day of week 1.
%V	Week number of the year (01 – 53), Monday as the first day of the week. If the week in which the 1st of January falls still has four or more days to run in the new year, this week is assigned the number 1. If it does not, then it is given the number 53 for the preceding year, and the following week is assigned number 1 [XPG4].
%w	Week day number (0 – 6), Sunday = 0.
%W	Week number of year (00 – 53), Monday is 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 as <i>ccyy</i> (e.g. 1986).
%Z	Time zone name or <i>no</i> characters if no time zone exists.

The difference between %U and %W lies in which day is counted as the first of the week. Week number 01 is the first week in January starting with a Sunday (for %U) or a Monday for (%W). Week number 00 contains those days before the first Sunday or Monday in January for %U and %W, respectively.

Warning:

%f, %V and %u are recognized in XPG4 mode only. %C has a different effect depending on whether you compiled or are running the programming in XPG3 mode or in XPG4 mode (the environment variable determines the mode if the mode is set dynamically).

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, *strftime()* returns the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, zero is returned and the contents of the array are indeterminate. *cftime()* and *ascftime()* return the number of bytes placed into the array pointed to by *s* not including the terminating null byte.

Modified Conversion Specifiers

Some conversion specifiers can be modified by the *E* or *O* modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier (see the "Programmer's Guide: Internationalization – Localization"). If the alternative format or specification does not exist for the current locale, the behavior will be as if the unmodified conversion specification were used.

Warning:

These conversion specifiers are recognized in XPG4 mode only. They are returned without any modifications in XPG3 mode.

%Ec	The locale's alternative appropriate date and time representation.
%EC	The name of the base year (period) in the locale's alternative representation.
%Ex	The locale's alternative date representation.
%EX	The locale's alternative time representation.
%Ey	The offset from %EC (year only) in the locale's alternative representation.
%EY	The full alternative year representation.
%Od	The day of the month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero, otherwise with leading spaces.
%Oe	The day of month, using the locale's alternative numeric symbols, filled as needed with leading spaces.
%OH	The hour (24-hour clock) using the locale's alternative numeric symbols.

- `%OI` The hour (12-hour clock) using the locale's alternative numeric symbols.
- `%Om` The month using the locale's alternative numeric symbols.
- `%OM` The minutes using the locale's alternative numeric symbols.
- `%OS` The seconds using the locale's alternative numeric symbols.
- `%Ou` The weekday as a number in the the locale's alternative representation (Monday = 1).
- `%OU` The week number of the year (Sunday as the first day of the week, rules corresponding to `%U`) using the locale's alternative numeric symbols.
- `%OV` The week number of the year (Monday as the first day of the week, rule corresponding to `%V`) using the locale's alternative numeric symbols.
- `%Ow` The number of the weekday (Sunday = 0) using the locale's alternative numeric symbols.
- `%OW` The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
- `%Oy` The year (offset from `%C`) in the locale's alternative representation and using the locale's alternative symbols.

Selecting the output language

By default, the output of `strftime()`, `cftime()`, and `ascftime()` appear in US English. The user can request that the output of `strftime()`, `cftime()` or `ascftime()` be in a specific language by using `setlocale()` to set the `LC_TIME` category for the locale.

Timezone

The timezone is taken from the environment variable `TZ` [see [cftime\(3C\)](#)].

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, `strftime()` returns the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, 0 is returned and the contents of the array are indeterminate.

ERRORS

No errors are defined.

EXAMPLES

The example illustrates the use of `strftime()`. 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 in New Jersey.

```
strftime (str, strsize, "%A %b %d %j", tmptr)
```

This results in `str` containing "Thursday Aug 28 240".

NOTES

`cftime()` and `ascftime()` are obsolete. `strftime()` should be used instead.

Since the `cftime()` function uses the [localtime\(3C\)](#) function internally, use of the static objects as described for this function must be observed.

Please note that some conversion specifiers are recognized in XPG4 mode only. These are returned without any modifications in XPG3 mode.

FILES

```
/usr/lib/locale/locale/LC_TIME
file containing locale specific date and time information
```

SEE ALSO

```
ctime(3C), getenv(3C), localtime(3C), setlocale(3C), LC_TIME(4), timezone(4),  
environ(5), time(5).
```

NAME

string:
 strcat, strdup, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr,
 strrchr, strpbrk, strspn, strcspn, strtok, strtok_r, strstr, strcasecmp,
 strncasecmp - manipulate strings

SYNOPSIS

```
#include <string.h>
char *strcat(char *s1, const char *s2);
char *strdup(const char *s1);
char *strncat(char *s1, const char *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
size_t strlen(const char *s);
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
char *strpbrk(const char *s1, const char *s2);
size_t strspn(const char *s1, const char *s2);
size_t strcspn(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
char *strtok_r(char *s, const char *sep, char **lasts);
char *strstr(const char *s1, const char *s2);
#include <strings.h>
int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);□
```

DESCRIPTION

The arguments *s*, *s1*, and *s2* point to strings. The functions `strcat()`, `strncat()`, `strcpy()`, `strncpy()`, and `strtok()` all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

`strcat()` appends a copy of string *s2*, including the terminating null byte, to the end of string *s1*. `strncat()` appends at most *n* bytes. Each returns a pointer to the result (including terminating null byte). The initial byte of *s2* overwrites the null byte at the end of *s1*.

`strcmp()` compares its arguments and returns an integer less than, equal to, or greater than 0, based upon whether *s1* is lexicographically less than, equal to, or greater than *s2*. `strncmp()` makes the same comparison but looks at, at most, *n* bytes. Bytes following a null byte are not compared.

`strcpy()` copies string *s2* to *s1* including the terminating null byte, stopping after the null byte has been copied. `strncpy()` copies exactly *n* bytes, truncating *s2* or adding null bytes to *s1* if necessary. The result will not be terminated with a null byte if the length of *s2* is *n* or more. Each function returns *s1*.

`strdup()` returns a pointer to a new string which is a duplicate of the string pointed to by *s1*. The space for the new string is allocated using `malloc(3C)`. If the new string can not be created, a null pointer is returned.

`strlen()` computes the number of bytes in the string to which *s* points, not including the terminating null

byte.

`strchr()` (or `strrchr()`) returns a pointer to the first (last) occurrence of *c* (converted to a `char`) in string *s*, or a null pointer if *c* does not occur in the string. The null byte terminating a string is considered to be part of the string.

`strpbrk()` returns a pointer to the first occurrence in string *s1* of any byte from string *s2*, or a null pointer if no byte from *s2* exists in *s1*.

`strspn()` returns the length of *s1*; no return value is reserved to indicate an error.

`strcspn()` computes the length of the maximum initial segment of the string pointed to by *s1* which consists entirely of the bytes *not* from the string pointed to by *s2*.

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* may 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 will return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The `strtok()` function saves a pointer to the following byte, 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.

The implementation will behave as if no function calls `strtok()`.

The `strtok()` interface need not be thread-safe.

The function `strtok_r()` considers the null-terminated string *s* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *sep*. 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()`, *s* points to a null-terminated string, *sep* to a null-terminated string of separator characters and the value pointed to by *lasts* is ignored. The function `strtok_r()` returns a pointer to the first character of the first token, writes a null character into *s* immediately following the returned token, and updates the pointer to which *lasts* points.

In subsequent calls, *s* is a `NULL` pointer and *lasts* will be unchanged from the previous call so that subsequent calls will move through the string *s*, returning successive tokens until no tokens remain. The separator string *sep* may be different from call to call. When no token remains in *s*, a `NULL` pointer is returned.

`strstr()` locates the first occurrence in string *s1* of the sequence of characters (excluding the terminating null byte) in string *s2*. `strstr()` 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 (i.e., the empty string), the function returns *s1*.

The `strcasecmp()` function compares, while ignoring differences in case, the string pointed to by *s1* to the string pointed to by *s2*. The `strncasecmp()` function compares, while ignoring differences in case, not more than *n* bytes from the string pointed to by *s1* to the string pointed to by *s2*.

The last two functions assume the ASCII character set when equating lower and upper case characters. In the POSIX locale, `strcasecmp()` and `strncasecmp()` do upper to lower conversions, then a byte comparison. The results are unspecified in other locales.

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

RETURN VALUES

The `strcat()` function returns `s1`; no return value is reserved to indicate an error.

The `strdup()` function returns a pointer to a new string on success. Otherwise it returns a null pointer and sets `errno` to indicate the error.

The `strncat()` function returns `s1`; no return value is reserved to indicate an error.

Upon completion, `strcmp()` 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.

Upon successful completion, `strncmp()` returns an integer greater than, equal to or less than 0, if the possibly null-terminated array pointed to by `s1` is greater than, equal to or less than the possibly null-terminated array pointed to by `s2` respectively.

The `strcpy()` function returns `s1`; no return value is reserved to indicate an error.

The `strncpy()` function returns `s1`; no return value is reserved to indicate an error.

The `strlen()` function returns the length of `s`; no return value is reserved to indicate an error.

Upon completion, `strchr()` returns a pointer to the byte, or a null pointer if the byte was not found.

Upon successful completion, `strrchr()` returns a pointer to the byte or a null pointer if `c` does not occur in the string.

Upon successful completion, `strpbrk()` returns a pointer to the byte or a null pointer if no byte from `s2` occurs in `s1`.

The `strspn()` function returns the length of `s1`; no return value is reserved to indicate an error.

The `strcspn()` function returns the length of `s1`; no return value is reserved to indicate an error.

Upon successful completion, `strtok()` returns a pointer to the first byte of a token. Otherwise, if there is no token, `strtok()` returns a null pointer.

The function `strtok_r()` returns a pointer to the token found, or a `NULL` pointer when no token is found.

Upon successful completion, `strstr()` returns a pointer to the located string or a null pointer if the string is not found.

Upon completion, `strcasecmp()` returns an integer greater than, equal to or less than 0, if the string pointed to by `s1` is, ignoring case, greater than, equal to or less than the string pointed to by `s2` respectively.

Upon successful completion, `strncasecmp()` returns an integer greater than, equal to or less than 0, if the possibly null-terminated array pointed to by `s1` is, ignoring case, greater than, equal to or less than the possibly null-terminated array pointed to by `s2` respectively.

ERRORS

The `strdup()` function may fail if:

`ENOMEM`

Storage space available is insufficient.

NOTES

All of these functions assume the C locale by default. For some locales, `strxfrm()` should be applied to the strings before they are passed to the functions.

SEE ALSO

`bstring(3C)`, `malloc(3C)`, `setlocale(3C)`, `strxfrm(3C)`, `pthread_intro(3T)`, `string(5)`, `strings(5)`.

strptime(3C)**NAME**

`strptime` - date and time conversion

SYNOPSIS

```
#include <time.h>
char *strptime(const char *buf, const char *format, struct
tm *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*.

format is composed of zero or more directives. Each directive is composed of one of the following: one or more white-space characters (as specified by the `isspace()` function); an ordinary character (neither % nor a white-space character); or a conversion specification. Each conversion specification is composed of a % character followed by a conversion character which specifies the replacement required. There must be white-space or other non-alphanumeric characters between any two conversion specifications. The following conversion specifications are supported [cf. `LC_TIME(4)`]:

- %a Is the day of week, using the locales weekday names; either the abbreviated or full name may be specified.
- %A Is the same as %a.
- %b Is the month, using the locales month names; either the abbreviated or full name may be specified.
- %B Is the same as %b.
- %c Is the date and time, using locales date and time format (for example, as %x%X).
- %C Is the century number [0,99]; leading zeros are permitted but not required.
- %d Is the day of the month [1,31]; leading zeros are permitted but not required.
- %D Is the date as %m/%d/%Y.
- %e Is the same as %d.
- %h Is the same as %b.
- %H Is the hour (24-hour clock) [0,23]; leading zeros are permitted but not required.
- %I Is the hour (12-hour clock) [1,12]; leading zeros are permitted but not required.
- %j Is the day number of the year [1,366]; leading zeros are permitted but not required.
- %m Is the month number [1,12]; leading zeros are permitted but not required.
- %M Is the minute [0,59]; leading zeros are permitted but not required.
- %n Any white space.
- %p Is the locales equivalent of a.m. or p.m.
- %r Is the time as %I:%M:%S:%p.
- %R Is the time as %H:%M.
- %S Is the second [0,61]; leading zeros are permitted but not required.
- %t Is any white space.
- %T Is the time as %H:%M:%S.
- %U Is the week number of the year (Sunday as the first day of the week 1) a decimal number [00,53]; leading zeros are permitted but not required.
- %w Is the weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.

- `%W` Is the week number of the year (Monday as first day of the week 1) as a decimal number [00,53]; leading zeros are permitted but not required.
- `%x` Is the date, using the locales date format.
- `%X` Is the time, using the locales time format.
- `%y` Is the 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). Leading zeros are permitted but not required.
- `%Y` Is the year including the century (for example 1988).
- `%%` Is replaced by %.

Modified Directives

Some directives 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 directive (see "Programmer's Guide: Internationalization – Localization"). If the alternative format or specification does not exist in the current locale, the behavior will be as if the unmodified directive were used.

- `%Ec` Is the locales alternative appropriate date and and time representation.
- `%EC` Is the name of the base year (period) in the locales alternative representation.
- `%Ex` Is the locales alternative date representation.
- `%EX` Is the locales alternative time representation.
- `%Ey` Is the offset from `%EC` (year only) in the locales alternative representation.
- `%EY` Is the full alternative year representation.
- `%Od` Is the day of the month using the locales alternative numeric symbols; leading zeros are permitted but not required.
- `%Oe` Is the same as `%Od`.
- `%OH` Is the hour (24-hour clock) using the locales alternative numeric symbols.
- `%OI` Is the hour (12-hour clock) using the locales alternative numeric symbols.
- `%Om` Is the month using the locales alternative numeric symbols.
- `%OM` Is the minutes using the locales alternative numeric symbols.
- `%OS` Is the seconds using the locales alternative numeric symbols.
- `%OU` Is the week number of the year (Sunday as the first day of the week) using the locales alternative numeric symbols.
- `%Ow` Is the number of the weekday (Sunday = 0) using the locales alternative numeric symbols.
- `%OW` Is the week number of the year (Monday is the first day of the week) using the locales alternative numeric symbols.
- `%Oy` Is the year (offset from `%C`) in the locales alternative representation and using the locales alternative numeric symbols.

A directive composed of white-space characters is executed by scanning input up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

A directive 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 directive, the directive fails, and the differing and subsequent characters remain unscanned.

A series of directives 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 directive is scanned, or until no more characters can be scanned. These characters, except the one matching the next directive, are then compared to the locale values associated with the conversion specifier. If a match

is found, values for appropriate *tm* structure members are set to values corresponding to the locale information. Case is ignored when matching items in *buf* such as month or weekday names. If no match is found, `strptime()` fails and no more characters can be scanned.

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()` and `strptime()`.

RETURN VALUE

Upon successful completion, `strptime()` returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

ERRORS

No errors are defined.

SEE ALSO

`time(2)`, `strftime(3C)`, `scanf(3S)`, `LC_TIME(4)`, `time(5)`.

NAME

`strtod`, `atof` - convert string to double-precision number

SYNOPSIS

```
#include <stdlib.h>
double strtod(const char *str, char **endptr);
double atof(const char *nptr); □
```

DESCRIPTION

The `strtod()` function converts the initial portion of the string pointed to by *str* to type `double` representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters [as specified by `isspace()`]; a subject sequence interpreted as a floating-point constant; 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 a floating-point number, and returns the result.

The expected form of the subject sequence is an optional + or – sign, then a non-empty sequence of digits optionally containing a radix character, then an optional exponent part. An exponent part consists of `e` or `E`, followed by an optional sign, followed by one or more decimal digits. 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 is empty if the input string is empty or consists entirely of white-space characters, or if the first character that is not white space is other than a sign, a digit or a radix character.

If the subject sequence has the expected form, the sequence starting with the first digit or the radix 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, a radix character is assumed to follow the last digit in the string. 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.

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 (`.`).

In other than the POSIX locale, other 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 `strtod()` function will not change the setting of `errno` if successful.

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, then call `strtod()`, then check `errno`.

`atof(nptr)` is equivalent to:

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

If the return value of `atof()` cannot be represented, behavior is undefined.

RETURN VALUE

Upon successful completion, `strtod()` returns the converted value. If no conversion could be performed, 0 is returned, and `errno` may be set to `EINVAL`.

If the correct value would cause overflow – and when the `-kansi` or `-kosi` compilation options were used –, `±HUGE_VAL` (defined in `math.h`) is returned (according to the sign of *value*), and `errno` is set to `ERANGE`. [When the `-kcc` (default) compilation option is used, `HUGE` is returned instead of `HUGE_VAL`.]

If the correct value would cause underflow, zero is returned and `errno` is set to `ERANGE`.

ERRORS

The `strtod()` function will fail if:

`ERANGE`

The value to be returned would cause overflow or underflow.

The `strtod()` function may fail if:

`EINVAL`

No conversion could be performed.

NOTES

If the result of `atof()` is undefined because the return value cannot be represented, you should use the `strtod()` function instead.

SEE ALSO

`cc(1)`, `ctype(3C)`, `strtol(3C)`, `scanf(3S)`, `stdlib(5)`.

strtol(3C)**NAME**

`strtol`, `strtoul`, `atol`, `atoi`, `strtoll`, `strtoull`, `atoll` - convert string to integer

SYNOPSIS

```
#include <stdlib.h>

□
□
□

long int strtol(const char *str, char **endptr, int base);
unsigned long int strtoul(const char *str, char **endptr, int base);
long int atol(const char *str);
int atoi(const char *str);
long long strtoll(const char *str, char **ptr, int base);
unsigned long long strtoull(const char *str, char **ptr, int base);
long long atoll(const char *str); □
```

DESCRIPTION

`strtol()` returns as a `long int` the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters [as defined by `isspace()` in [ctype\(3C\)](#)] are ignored.

If the value of *endptr* is not `(char **)NULL`, a pointer to the character terminating the scan is returned in the location pointed to by *endptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored; if *base* is 16, the characters "0x" or "0X" are also ignored.

If *base* is zero, the string itself determines the base as follows: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from `long` to `int` can, of course, take place upon assignment or by an explicit cast.

If the value represented by *str* would cause overflow, `LONG_MAX` or `LONG_MIN` is returned (according to the sign of the value), and `errno` is set to the value `ERANGE`.

`strtoul()` is similar to `strtol()` except that `strtoul()` returns as an unsigned `long` □ `int` the value represented by *str*. If the value represented by *str* would cause overflow, `ULONG_MAX` is returned, and `errno` is set to the value `ERANGE`.

`strtoll()` is similar to `strtol()` except that `strtoll()` returns as a `long long` the value represented by *str*. If the value represented by *str* would cause overflow, `LLONG_MAX` or `LLONG_MIN` is returned (according to the sign of the value), and `errno` is set to the value, `ERANGE`.

`strtoull()` is similar to `strtoul()` except that `strtoull()` returns as an unsigned `long long` the value represented by *str*. If the value represented by *str* would cause overflow, `ULLONG_MAX` is returned, and `errno` is set to the value, `ERANGE`.

Except for behavior on error, `atol(str)` is equivalent to:

```
strtol(str, (char **)NULL, 10).
```

Except for behavior on error, `atoi(str)` is equivalent to:

```
(int) strtol(str, (char **)NULL, 10).
```

Except for behavior on error, `atoll(str)` is equivalent to:

```
strtoll(str, (char **)NULL, 10).
```

RETURN VALUE

Upon successful completion `strtol()` or `strtoul()` return 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, `LONG_MAX` or `LONG_MIN` or `ULONG_MAX` is returned (according to the sign of the value), and `errno` is set to `ERANGE`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `strtol()` function will fail if:

`ERANGE`

The value to be returned is not representable.

The `strtol()` function may fail if:

`EINVAL`

The value of *base* is not supported.

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.

NOTES

The `strtol()` or `strtoul()` function will not change the setting of `errno` if successful.

Because 0, `LONG_MIN` and `LONG_MAX` or `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 the function, then check `errno`.

`strtol()` no longer accepts values greater than `LONG_MAX` as valid input. Use `strtoul()` instead.

As the result is undefined for a non-representable return value for `atol()` and `atoi()`, you should use `strtol()` instead of these functions.

SEE ALSO

[ctype\(3C\)](#), [strtod\(3C\)](#), [scanf\(3S\)](#), [stdlib\(5\)](#).

strxfrm(3C)**NAME**

strxfrm - string transformation

SYNOPSIS

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

DESCRIPTION

The `strxfrm()` function transforms the string pointed to by `s2` and places the resulting string into the array pointed to by `s1`. The transformation is such that if `strcmp()` is applied to two transformed strings, it returns a value greater than, equal to or less than 0, corresponding to the result of `strcoll()` 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 behaviour is undefined.

The `strxfrm()` function will not change the setting of `errno` if successful.

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 `strcoll()`, then check `errno`.

RETURN VALUE

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 an error.

ERRORS

The `strxfrm()` function may fail if:

EINVAL

The string pointed to by the `s2` argument contains characters outside the domain of the collating sequence.

EXAMPLES

The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by `s`.

```
1 + strxfrm(NULL, s, 0);
```

APPLICATION USAGE

The transformation function is such that two transformed strings can be ordered by `strcmp()` 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.

FILES

`/usr/lib/locale/locale/LC_COLLATE`
database for *locale*

SEE ALSO

`colltbl(1M)`, `setlocale(3C)`, `strcoll(3C)`, `string(3C)`, `environ(5)`, `string(5)`.

NAME

swab - swap bytes

SYNOPSIS

```
#include <unistd.h>
void swab(const void *src, void *dest, ssize_t nbytes);
```

DESCRIPTION

`swab()` copies *nbytes* bytes pointed to by *src* to the array pointed to by *dest*, exchanging adjacent even and odd bytes. *nbytes* should be even and non-negative. If *nbytes* is odd, `swab()` uses *nbytes* - 1 instead, 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.

SEE ALSO

`unistd(4)`.

sysconf(3C)**NAME**

sysconf - get configurable system variables

SYNOPSIS

```
#include <unistd.h>
long int sysconf(int name);
```

DESCRIPTION

The `sysconf()` function provides a method for the application to determine the current values of a configurable system limit or option (variable).

The *name* argument represents the system variable to be queried. The following table lists the set of system variables from `<limits.h>` and `<unistd.h>` or `<time.h>` (for `CLK_TCK`) that can be returned by `sysconf()`, and the symbolic constants, defined in `<unistd.h>` that are the corresponding values used for *name*. □

System variable	Value of <i>name</i>	
ARG_MAX	_SC_ARG_MAX	
BC_BASE_MAX	_SC_BC_BASE_MAX	[XPG4]
BC_DIM_MAX	_SC_BC_DIM_MAX	[XPG4]
BC_SCALE_MAX	_SC_BC_SCALE_MAX	[XPG4]
BC_STRING_MAX	_SC_BC_STRING_MAX	[XPG4]
CHILD_MAX	_SC_CHILD_MAX	
CLK_TCK	_SC_CLK_TCK	
COLL_WEIGHTS_MAX	_SC_COLL_WEIGHTS_MAX	[XPG4]
EXPR_NEST_MAX	_SC_EXPR_NEST_MAX	[XPG4]
LINE_MAX	_SC_LINE_MAX	[XPG4]
NGROUPS_MAX	_SC_NGROUPS_MAX	
OPEN_MAX	_SC_OPEN_MAX	
PASS_MAX	_SC_PASS_MAX	
_POSIX2_C_BIND	_SC_2_C_BIND	[XPG4]
_POSIX2_C_DEV	_SC_2_C_DEV	[XPG4]
_POSIX2_C_VERSION	_SC_2_C_VERSION	[XPG4]
_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM	[XPG4]

		]
_POSIX2_FORT_DEV	_SC_2_FORT_DEV	[XPG4]
_POSIX2_FORT_RUN	_SC_2_FORT_RUN	[XPG4]
_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF	[XPG4]

System variable	Value of name	
_POSIX2_SW_DEV	_SC_2_SW_DEV	[XPG4]
_POSIX2_UPE	_SC_2_UPE	[XPG4]
_POSIX2_VERSION	_SC_2_VERSION	[XPG4]
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL	
_CONF_PAGESIZE	_SC_PAGESIZE	
_POSIX_SAVED_IDS	_SC_SAVED_IDS	
_POSIX_VERSION	_SC_VERSION	*)
RE_DUP_MAX	_SC_RE_DUP_MAX	[XPG4]
STREAM_MAX	_SC_STREAM_MAX	[XPG4]
TZNAME_MAX	_SC_TZNAME_MAX	[XPG4]
_XOPEN_CRYPT	_SC_XOPEN_CRYPT	[XPG4]
_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N	[XPG4]
_XOPEN_SHM	_SC_XOPEN_SHM	[XPG4]
_XOPEN_VERSION	_SC_XOPEN_VERSION	*)
_XOPEN_XCU_VERSION	_SC_XOPEN_XCU_VERSION	[XPG4]
_XOPEN_UNIX	_SC_XOPEN_UNIX	[XPG4]
_XOPEN_REALTIME	_SC_XOPEN_REALTIME	
_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS	
LOGNAME_MAX	_SC_LOGNAME_MAX	
ATEXIT_MAX	_SC_ATEXIT_MAX	[XPG4]
IOV_MAX	_SC_IOV_MAX	[XPG4]

PAGESIZE	_SC_PAGESIZE	
PAGE_SIZE	_SC_PAGE_SIZE	
_POSIX_FSYNC	_SC_FSYNC	
_POSIX_MAPPED_FILES	_SC_MAPPED_FILES	
_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION	
Maximum size of getgrgid_r() and getgrnam_r() data buffers	_SC_GETGR_R_SIZE_MAX	
Maximum size of getpwuid_r() and getpwnam_r() data buffers	_SC_GETPW_R_SIZE_MAX	
LOGIN_NAME_MAX	_SC_LOGIN_NAME_MAX	

System variable	Value of name	
PTHREAD_DESTRUCTOR_ITERATIONS	_SC_THREAD_DESTRUCTOR_ITERATIONS	
PTHREAD_KEYS_MAX	_SC_THREAD_KEYS_MAX	
PTHREAD_STACK_MIN	_SC_THREAD_STACK_MIN	
PTHREAD_THREADS_MAX	_SC_THREAD_THREADS_MAX	
TTY_NAME_MAX	_SC_TTY_NAME_MAX	
_POSIX_THREADS	_SC_THREADS	
_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR	
_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE	
_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING	
_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT	
_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT	
_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED	
_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS	

*) These variables provide different values depending on the compilation mode:

_SC_VERSION	XPG3 mode: 3	XPG4 mode: 4
_SC_XOPEN_VERSION	XPG3 mode: 198808	XPG4 mode: 199009

RETURN VALUE

If *name* is an invalid value, `sysconf()` will return `-1` and set `errno` to indicate the error. If `sysconf()` fails due to a value of *name* that is not defined on the system, the function will return a value of `-1` without changing the value of `errno`. This is the case, for example, if compilation was in XPG3 mode, but `sysconf()` was called using a variable that is defined for XPG4 mode only.

ERRORS

The `sysconf()` function will fail if:

EINVAL

The value of the *name* argument is invalid.

NOTES

A call to `setrlimit()` may cause the value of `OPEN_MAX` to change.

If the value of `sysconf(_SC_2_VERSION)` is not equal to the value of the `_POSIX2_VERSION` symbolic

constant, the utilities available via `system()` or `popen()` might not behave as explained in the description of the command. This would mean that the application is not running in an environment that conforms to the description. Some applications might be able to deal with this, others might not.

SEE ALSO

`fpathconf(2)`, `limits(4)`, `unistd(4)`, `time(5)`.

syslog(3C)**NAME**

syslog, openlog, closelog, setlogmask - control system log

SYNOPSIS

```
#include <syslog.h>
void syslog(int level, const char *message, ... /*
arguments */);
void openlog(const char *ident, int logopt, int
facility);
void closelog(void);
int setlogmask(int maskpri);
```

DESCRIPTION

syslog passes *message* to syslogd, which logs it in an appropriate system log, writes it to the system console, forwards it to a list of users, or forwards it to the syslogd on another host over the network. The message looks like a printf string except that %m is replaced by the current error message (collected from errno). A trailing NEWLINE is added if needed.

The message is tagged with a priority. Priorities are composed of *facility* and *level*. The facility describes the part of the system generating the message. The level is selected from an ordered list:

LOG_EMERG

System halt.

LOG_ALERT

Error indicating considerable loss of data and requiring immediate action.

LOG_CRIT

Error indicating loss of data or malfunctions requiring action soon.

LOG_ERR

Error indicating malfunctions but not requiring immediate action.

LOG_WARNING

Warning that normal functioning of the system could be impaired unless the appropriate action is taken.

LOG_NOTICE

Indication of a possible problem.

LOG_INFO

Status message.

LOG_DEBUG

Messages only used in program debugging.

If special processing is needed, openlog can be called to initialize the log file. The parameter *ident* is a string that is prepended to every message. *logopt* is a bit field indicating logging options. Current values for *logopt* are:

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. 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 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* parameter encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded:

`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 `ftpd`, `routed`, etc.

`LOG_AUTH`

The authorization system: `login`, `su`, `getty`, etc.

`LOG_LPR`

The line printer spooling system: `lpr`, `lpc`, etc.

`LOG_NEWS`

Reserved for the USENET network news system.

`LOG_UUCP`

Reserved for the UUCP system; it does not currently use `syslog`.

`LOG_CRON`

The `cron/at` facility; `crontab`, `at`, `cron`, etc.

`LOG_LOCAL0-7`

Reserved for local use.

`closelog` can be used to close the log file.

`setlogmask` sets the log priority mask to *maskpri* and returns the previous mask. If the *maskpri* argument is 0, the current log mask is not modified. Calls to `syslog` with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro `LOG_MASK(pri)`. The mask for all priorities up to and including *toppri* is given by the macro `LOG_UPTO(toppri)`. The default allows all priorities to be logged.

A call to `openlog` is not required prior to calling `setlogmask()`.

Symbolic constants for use as values of the *logopt*, *facility*, *priority*, and *maskpri* arguments are defined in `<syslog.h>`.

EXAMPLES

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` as other messages from system daemons are, and 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:

```
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 as other messages to the facility LOG_LOCAL2 are treated:

```
syslog(LOG_INFO|LOG_LOCAL2, "error: %m");
```

SEE ALSO

at(1), crontab(1), logger(1), login(1), lpr(1), su(1), cron(1M), ftpd(1M), getty(1M), lpc(1M), routed(1M), syslogd(1M), printf(3S), syslog(5).

termios(3C) - wtype(3C)

NAME

termios:
tcgetattr, tcsetattr, tcsendbreak, tcdrain, tcflush, tcflow, cfgetospeed,
cfsetospeed, cfgetispeed, cfsetispeed, tcgetpgrp, tcsetpgrp, tcgetsid - general
terminal interface

SYNOPSIS

```
#include <termios.h>

int tcgetattr(int fildes, struct termios *termios_p);
int tcsetattr(int fildes, int optional_actions,
               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(const struct termios *termios_p, speed_t speed);
speed_t cfgetispeed(const struct termios *termios_p);
int cfsetispeed(const struct termios *termios_p, speed_t speed);
#include <sys/types.h>
#include <unistd.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\(7\)](#), which also describes an [ioctl\(2\)](#) interface that provides the same functionality. However, the function interface described here is the preferred user interface.

Many of the functions described here have a *termios_p* argument that is a pointer to a *termios* structure. This structure contains the following members:

```
tcflag_t   c_iflag;           /* input modes */
tcflag_t   c_oflag;           /* output modes */
```

```

tcflag_t   c_cflag;           /* control modes */
tcflag_t   c_lflag;           /* local modes */
cc_t       c_cc[NCCS];        /* control chars */

```

These structure members are described in detail in [termio\(7\)](#).

Get and write terminal attributes

The `tcgetattr()` function gets the parameters associated with the object referred by *fildev* and stores them in the `termios` structure referenced by *termios_p*. This function may be invoked from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

The `tcsetattr()` function sets the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the `termios` structure referenced by *termios_p* as follows:

- If *optional_actions* is `TCSANOW`, the change occurs immediately.
- If *optional_actions* is `TCSADRAIN`, the change occurs after all output written to *fildev* has been transmitted. This function should be used when changing parameters that affect output.
- If *optional_actions* is `TCSAFLUSH`, the change occurs after all output written to the object referred by *fildev* has been transmitted, and all input that has been received but not read is discarded before the change is made.

The symbolic constants for the values of *optional_actions* are defined in `<termios.h>` [see [termios\(5\)](#)].

Line control

If the terminal is using asynchronous serial data transmission, the `tcsendbreak()` function causes transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is zero, it causes transmission of zero-valued bits for at least 0.25 seconds, and not more than 0.5 seconds. If *duration* is not zero, it behaves in a way similar to `tcdrain()`.

If the terminal is not using asynchronous serial data transmission, the `tcsendbreak()` function sends data to generate a break condition or returns without taking any action.

The `tcdrain()` function waits until all output written to the object referred to by *fildev* has been transmitted.

The `tcflush()` function discards data written to the object referred to by *fildev* 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.

The `tcflow()` function suspends transmission or reception of data on the object referred to by *fildev*, depending on the value of *action*:

- If *action* is `TCOOFF`, it suspends output.
- If *action* is `TCOON`, it restarts suspended output.
- If *action* is `TCIOFF`, the system transmits a STOP character, which causes the terminal device to stop transmitting data to the system.
- If *action* is `TCION`, the system transmits a START character, which causes the terminal device to start transmitting data to the system.

Get and set baud rate

The baud rate functions get and set the values of the input and output baud rates in the `termios` structure. The effects on the terminal device described below do not become effective until the `tcsetattr()` function is successfully called.

The input and output baud rates are stored in the `termios` structure. The values shown in the table are supported. The names in this table are defined in `<termios.h>`.

Name	Description	Name	Description
B0	Hang up	B600	600 baud
B50	50 baud	B1200	1200 baud
B75	75 baud	B1800	1800 baud
B110	110 baud	B2400	2400 baud
B134	134.5 baud	B4800	4800 baud
B150	150 baud	B9600	9600 baud
B200	200 baud	B19200	19200 baud
B300	300 baud	B38400	38400 baud

`cfgetospeed()` gets the output baud rate and stores it in the `termios` structure pointed to by `termios_p`.

`cfsetospeed()` sets the output baud rate stored in the `termios` structure pointed to by `termios_p` to `speed`. The zero baud rate, B0, is used to terminate the connection. If B0 is specified, the modem control lines are no longer be asserted. Normally, this disconnects the line.

`cfgetispeed()` gets the input baud rate and stores it in the `termios` structure pointed to by `termios_p`.

`cfsetispeed()` sets the input baud rate stored in the `termios` structure pointed to by `termios_p` to `speed`. If the input baud rate is set to zero, the input baud rate is specified by the value of the output baud rate. Both `cfsetispeed()` and `cfsetospeed()` return a value of zero if successful and `-1` to indicate an error. Attempts to set unsupported baud rates are ignored. This refers both to changes to baud rates not supported by the hardware, and to changes setting the input and output baud rates to different values if the hardware does not support this.

Get and set foreground process group ID for a terminal

`tcsetpgrp()` sets the foreground process group ID of the terminal specified by `fildev` to `pgid`. The file associated with `fildev` must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. `pgid` must match a process group ID of a process in the same session as the calling process.

`tcgetpgrp()` returns the foreground process group ID of the terminal specified by `fildev`. `tcgetpgrp()` 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.

Get session ID of a terminal

The `tcgetsid()` function obtains the process group ID of the session for which the terminal specified by `fildev` is the controlling terminal.

RETURN VALUE

On success, `tcgetpgrp()` returns the process group ID of the foreground process group associated with the specified terminal. Otherwise, it returns `-1` and sets `errno` to indicate the error.

On success, `tcgetsid()` returns the session ID associated with the specified terminal. Otherwise, it returns `-1` and sets `errno` to indicate the error.

On success, `cfgetispeed()` returns a value of type `speed_t` representing the input baud rate. On success, `cfgetospeed()` likewise returns a value of type `speed_t` representing the output baud rate. Both functions return the value from the `termios` structure without interpretation.

On success, all other functions return a value of 1. Otherwise, they return `-1` and set `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in

`intro_prm2(2)` or in `errno(5)`.

All of the functions fail if one of more of the following are true:

EBADF The *fildev* argument is not a valid file descriptor.

ENOTTY

The file associated with *fildev* is not a terminal.

The `cfsetispeed()` and the `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>`.

`tcsetattr()` also fails if the following is true:

EINVAL

The *optional_actions* argument is not a proper value, or an attempt was made to change an attribute represented in the `termios` structure to an unsupported value.

EINTR `tcsetattr()` was interrupted by a signal.

`tcsendbreak()` also fails if the following is true:

EINVAL

The device does not support the `tcsendbreak()` function.

`tcdrain()` also fails if one or more of the following are true:

EINTR A signal interrupted the `tcdrain()` function.

EINVAL

The device does not support the `tcdrain()` function.

`tcflush()` also fails if the following is true:

EINVAL

The device does not support the `tcflush()` function or the *queue_selector* argument is not a proper value.

`tcflow()` also fails if the following is true:

EINVAL

The device does not support the `tcflow()` function or the *action* argument is not a proper value.

`tcgetpgrp()` also fails if the following is true:

ENOTTY

The calling process does not have a controlling terminal, or *fildev* does not refer to the controlling terminal.

`tcsetpgrp()` also fails if the following is true:

EINVAL

pgid is not a valid process group ID.

ENOTTY

The calling process does not have a controlling terminal, or *fildev* does not refer to the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

EPERM *pgid* does not match the process group of an existing process in the same session as the calling process.

`tcgetsid()` also fails if the following is true:

EBADF The *fildev* 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.

SEE ALSO

`setpgid(2)`, `setsid(2)`, `unistd(4)`, `termios(5)`, `types(5)`, `termio(7)`.

NAME

times - get process times

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <sys/types.h>
#include <sys/times.h>
times(buffer)
struct tms *buffer;
```

DESCRIPTION

times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/HZ seconds, where HZ is 60.

This is the structure returned by times:

```
struct tms {
    time_t tms_utime; /* user time */
    time_t tms_stime; /* system time */
    time_t tms_cutime; /* user time, children */
    time_t tms_cstime; /* system time, children */
};
```

The children's times are the sum of the children's process times and their children's times.

NOTES

times has been superseded by [getrusage\(2\)](#).

SEE ALSO

[time\(1\)](#), [getrusage\(2\)](#), [time\(2\)](#), [wait\(2\)](#), [wait3\(3C\)](#).

timezone(3C)
BSD Compatibility Package**NAME**

`timezone` - get time zone name given offset from GMT

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
char *timezone(zone, dst)
int zone;
int dst;
```

DESCRIPTION

`timezone` attempts to return the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used, otherwise the Daylight Savings Time version. If the required name does not appear in a table built into the routine, the difference from GMT is produced; for instance, in Afghanistan `timezone(-(60*4+30), 0)` is appropriate because it is 4:30 ahead of GMT and the string `GMT+4:30` is produced.

NOTES

The offset westward from Greenwich and an indication of whether Daylight Savings Time is in effect may not be sufficient to determine the name of the time zone, as the name may differ between different locations in the same time zone. Instead of using `timezone` to determine the name of the time zone for a given time, that time should be converted to a `struct tm` using `localtime` and the `tm_zone` field of that structure should be used. `timezone` is retained for compatibility with existing programs.

SEE ALSO

`ctime(3C)`.

towctrans(3C)**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 `towctrans()` that returned the value `desc`.

Each of the following two strings behaves the same as the call to the wide-character case-mapping function `wctrans(3C)` in the comment that follows the string:

```
towctrans(wc, wctrans("tolower"))    /* tolower(wc) */
towctrans(wc, wctrans("toupper"))    /* toupper(wc) */
```

RETURN VALUES

The `towctrans()` function returns the mapped value of `wc` using the mapping described by `desc`.

SEE ALSO

`wctrans(3C)`, `locale(5)`, `wctype(5)`.

truncate(3C)**NAME**

`truncate`, `ftruncate`, `truncate64`, `ftruncate64` - 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);
int truncate64(const char *path, off64_t length);
int ftruncate64(int fildes, off64_t length);
```

DESCRIPTION

The `ftruncate()` function causes the regular file referenced by *fildes* to have a size of *length* bytes.

The `truncate()` function causes the regular file named by *path* to have a size of *length* bytes.

If the file previously was larger than *length*, the extra data is discarded. If it was previously shorter than *length*, it is unspecified whether the file is changed or its size increased. If the file is extended, the extended area appears as if it were zero-filled.

With `ftruncate()`, the file must be open for writing; for `truncate()`, the process must have write permission for the file.

If the effect of truncation is to decrease the size of a file and whole pages beyond the new end were previously mapped, then the whole pages beyond the new end will be discarded. References to the discarded pages result in generation of a `SIGBUS` signal.

If the request would cause the file size to exceed the soft file size limit for the process, the request will fail and the implementation will generate the `SIGXFSZ` signal for the process.

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 may be cleared.

There is no functional difference between `truncate()/ftruncate()` and `truncate64()/ftruncate64()`, except for the interpretation of `off64_t` [see [lfs\(5\)](#)].

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`truncate()` and `ftruncate()` 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 the file system.

`truncate()` will fail if:

EACCES

A component of the path prefix denies search permission, or write permission is denied on the file.

EISDIR

The file referred to by *path* is a directory.

ELOOP

Too many symbolic links were encountered in translating *path*.

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

Either a component of the path prefix or the file referred to by *path* does not exist.

ENOTDIR

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

EROFS

The file referred to by *path* resides on a read-only file system.

The `truncate()` function may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

`ftruncate()` will fail if:

EBADF or EINVAL

fildev 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 *fildev*.

EINVAL

The *fildev* argument references a file that was opened without write permission.

EROFS

The file referred to by *path* resides on a read-only file system.

SEE ALSO

`fcntl(2)`, `open(2)`, `unistd(4)`, `lfs(5)`.

NAME

tsearch, tfind, tdelete, twalk - manage binarysearch 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 *key, void **rootp,
              int (*compar)(const void *, const void *));

void twalk(const void *root,
           void (*action)(const void *, VISIT, int));
```

DESCRIPTION

tsearch(), tfind(), tdelete(), and twalk() 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.

tsearch() is used to build and access the tree. *key* 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. *rootp* 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().

tdelete() 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.

twalk() traverses a binary search tree. *root* 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 structure pointed to by this argument is unspecified and must not be modified. The value of type "pointer-to-node" can be converted to type "pointer-to-pointer-to-element" to access the elements stored in the node.

The second argument is a value from an enumeration data type

```
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

(defined in the search.h header file), 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.

If the calling function alters the pointer to the root, results are unpredictable.

RETURN VALUE

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.

EXAMPLES

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 <search.h>
#include <string.h>
#include <stdio.h>
#define STRSZ 10000
#define NODSZ 500
struct node { /* pointers to these are stored in the tree */
    char *string;
    int length;
};
char string_space[STRSZ]; /* space to store strings */
struct node nodes[NODSZ]; /* nodes to store */
void *root = NULL; /* this points to the root */
int main(int argc, char *argv[])
{
    char *strpstr = string_space;
    struct node *nodeptr = nodes;
    void print_node(const void *, VISIT, int);
    int i = 0, node_compare(const void *, const void *);
    while (gets(strpstr) != NULL && i++ < NODSZ) {
        /* set node */
        nodeptr->string = strpstr;
        nodeptr->length = strlen(strpstr);
        /* put node into the tree */
        (void) tsearch((void *)nodeptr, (void **)&root,
            node_compare);
        /* adjust pointers, so we do not overwrite tree */
        strpstr += nodeptr->length + 1;
        nodeptr++;
    }
    twalk(root, print_node);
    return 0;
}
/* This routine compares two nodes, based on an
   alphabetical ordering of the string field. */
int
node_compare(const void *node1, const void *node2)
{
    return strcmp(((const struct node *) node1)->string,
        ((const struct node *) node2)->string);
}
/* This routine prints out a node, the second time
   twalk encounters it or if it is a leaf. */
void
```

```
print_node(const void *ptr, VISIT order, int level)
{
    const struct node *p = *(const struct node **) ptr;
    if (order == postorder || order == leaf) {
        (void) printf("string = %s, length = %d\n",
                     p->string, p->length);
    }
}
```

NOTES

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, so that `postorder` has a different meaning in this case.

SEE ALSO

[bsearch\(3C\)](#), [hsearch\(3C\)](#), [lsearch\(3C\)](#), [search\(5\)](#).

NAME

`ttyname`, `ttyname_r`, `isatty` - find name of a terminal

SYNOPSIS

```
#include <unistd.h>
char *ttyname(int fdes) ;
int ttyname_r(int fdes, char *name, size_t namesize);
int isatty(int fdes);
```

DESCRIPTION

`ttyname()` returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fdes*. The return value may point to static data whose content is overwritten by each call.

The `ttyname()` interface need not be thread-safe.

The `ttyname_r()` function stores the null-terminated pathname of the terminal associated with the file descriptor *fdes* in the character array referenced by *name*. The array is *namesize* characters long and should have space for the name and the terminating null character. The maximum length of the terminal name is `TTY_NAME_MAX`.

`isatty()` returns 1 if *fdes* is associated with a terminal device, 0 otherwise.

RETURN VALUE

Upon successful completion, `ttyname()` returns a pointer to a string. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

If successful, the `ttyname_r()` function returns zero. Otherwise, an error number is returned to indicate the error.

The `isatty()` function returns 1 if *fdes* is associated with a terminal; otherwise it returns 0 and may set `errno` to indicate the error.

ERRORS

The `ttyname()` function may fail if:

`EBADF`

The *fdes* argument is not a valid file descriptor.

`ENOTTY`

The *fdes* argument does not refer to a terminal device.

The `ttyname_r()` function may fail if:

`EBADF`

The *fdes* argument is not a valid file descriptor.

`ENOTTY`

The *fdes* argument does not refer to a tty.

`ERANGE`

The value of *namesize* is smaller than the length of the string to be returned including the terminating null character.

The `isatty()` function may fail if:

`EBADF`

The *fdes* argument is not a valid open file descriptor.

`ENOTTY`

The *fdes* argument is not associated with a terminal.

FILES

/dev/*

SEE ALSO

[pthread_intro\(3T\)](#), [ttysrch\(4\)](#), [unistd\(4\)](#).

NAME

ttyslot - find the slot in the utmp file of the current user

SYNOPSIS

```
#include <stdlib.h>
int  ttyslot(void);
```

DESCRIPTION

ttyslot() returns the index of the current user's entry in the `/var/adm/utmp` file. The returned index is determined by scanning files in `/dev` for the name of the terminal associated with the standard input, the standard output, or the standard error output (0, 1, or 2).

RESULT

A value of `-1` is returned 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.

FILES

`/var/adm/utmp`

SEE ALSO

[getut\(3C\)](#), [ttyname\(3C\)](#), [stdlib\(5\)](#).

unlockpt(3C)**NAME**

`unlockpt` - unlock a pseudo-terminal master/slave pair

SYNOPSIS

```
#include <stdlib.h>
int unlockpt(int fildev);
```

DESCRIPTION

The `unlockpt()` function unlocks the slave pseudo-terminal device associated with the master to which *fildev* refers.

Portable applications must call `unlockpt()` before opening the slave side of a pseudo-terminal device.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `unlockpt()` function may fail if:

EBADF

The *fildev* argument is not a file descriptor open for writing.

EINVAL

The *fildev* argument is not associated with a master pseudo-terminal device.

SEE ALSO

[open\(2\)](#), [grantpt\(3C\)](#), [ptsname\(3C\)](#), [stdlib\(5\)](#).

NAME

utimes - set file times

SYNOPSIS

```
#include <sys/time.h>
int utimes(const char *path, const struct timeval times[2]);
```

DESCRIPTION

The `utimes()` function sets the access and modification times of the file pointed to by the `path` argument to the value of the `times` argument. The `utimes()` function allows time specifications accurate to the microsecond.

For `utimes()`, the `times` argument is an array of `timeval` structures. The first array member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the `timeval` structure are measured in seconds and microseconds since the Epoch, although rounding toward the nearest second may occur.

If the `times` argument is a null pointer, the access and modification times of the file are set to the current time. The effective user ID of the process must be the same as the owner of the file, or must have write access to the file or appropriate privileges to use this call in this manner. Upon completion, `utimes()` will mark the time of the last file status change, `st_ctime`, for update.

In either case, the *inode-changed* time of the file is set to the current time.

RETURN VALUE

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

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`utimes` fails if one or more of the following is true:

EACCES

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

EACCES

The effective user ID of the process is not privileged user and not the owner of the file, write permission is denied for the file, and `times` is NULL.

ENOTDIR

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

ENAMETOOLONG

The length of the `path` argument exceeds `PATH_MAX` or the length of a `path` component exceeds `NAME_MAX`.

ENOENT

The file referred to by `path` does not exist.

ELOOP

Too many symbolic links were encountered in translating `path`.

EPERM

The `times` argument is not a null pointer and the calling process' effective user ID has write access to the file but does not match the owner of the file and the calling process does not have the appropriate privileges.

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

EROFS

The file system containing the file is mounted read-only.

EFAULT

path or *times* points outside the process' allocated address space.

The `utimes()` function may fail if:

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

NOTES

`utimes()` is a library routine that calls the `utime()` system call.

SEE ALSO

`stat(2)`, `utime(2)`, `sys_time(5)`.

NAME

`wait3` - wait for process to terminate or stop

SYNOPSIS

```
#include <sys/wait.h>
pid_t wait3(int *stat_loc, int options, struct rusage
*resource_usage);
```

DESCRIPTION

The `wait3()` function allows the calling process to obtain status information for specified child processes.

The following call:

```
wait3(stat_loc, options, resource_usage);
```

is equivalent to the call:

```
waitpid((pid_t)-1, stat_loc, options);
```

except that on successful completion, if the *resource_usage* argument to `wait3()` is not a null pointer, the *rusage* structure that the third argument points to is filled in for the child process identified by the return value.

RETURN VALUE

If `wait3()` returns because the status of a child process is available, it will return a value equal to the process ID of the child process for which status is reported. If `wait3()` returns due to the delivery of a signal to the calling process, `-1` will be returned and `errno` will be 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*, `0` will be returned. Otherwise, `(pid_t)-1` will be returned, and `errno` will be set to indicate the error.

ERRORS

In addition to the error conditions specified on `waitpid(2)`, under the following conditions, `wait3()` may fail and set `errno` to:

`ECHILD`

The calling process has no existing unwaited-for child processes, or if the set of processes specified by the argument *pid* can never be in the states specified by the argument *options*.

SEE ALSO

`exec(2)`, `exit(2)`, `fork(2)`, `pause(2)`, `waitpid(2)`, `wait(5)`.

NAME

`wcscoll` - wide character string comparison collating information

SYNOPSIS

```
#include <wchar.h>
int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

The `wcscoll()` function compares 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()` function will not change the setting of `errno` if successful.

Because no return value is reserved to indicate an error, an application wishing to check for error situation should set `errno` to 0, then call `wcscoll()`, then check `errno` and if it is non-zero, assume an error has occurred.

RETURN VALUE

Upon successful completion, the `wcscoll()` function returns an integer greater than, equal to or less than zero, according to whether the wide character string pointed to by `ws1` is greater than, equal or less than the wide character string pointed to by `ws2`, when both are interpreted as appropriate to the current locale. On error, `wcscoll()` may set `errno`, but no return value is reserved to indicate an error.

ERRORS

The `wcscoll()` function may fail if:

`EINVAL`

The `ws1` or `ws2` arguments contain wide character codes outside the domain of the collating sequence.

APPLICATION USAGE

The `wcsxfrm()` and `wcscmp()` functions should be used for sorting large lists.

SEE ALSO

[wcscmp\(3C\)](#), [wcsxfrm\(3C\)](#), [wchar\(5\)](#).

wcsftime(3C)**NAME**

wcsftime - convert date and time to wide character string

SYNOPSIS

```
#include <wchar.h>

size_t wcsftime(wchar_t *wcs, size_t maxsize, const wchar_t *format,
                const struct tm *timptr);

XPG4 mode:
size_t wcsftime(wchar_t *wcs, size_t maxsize, const char *format,
                const struct tm *timptr);
```

DESCRIPTION

The **wcsftime()** function places wide character codes into the array pointed to by *wcs* as controlled by the string pointed to by *format*.

This function behaves as if the character string generated by the **strftime()** function is passed to the **mbstowcs()** function as the character string argument, and the **mbstowcs()** function places the result in the wide character string argument of the **wcsftime()** function up to a limit of *maxsize* wide-character codes.

If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

If the total number of resulting wide character codes including the terminating null wide character code is no longer than *maxsize*, the **wcsftime()** function returns the number of wide character codes placed into the array pointed to by *wcs*, not including the terminating null wide character code. Otherwise zero is returned and the contents of the array are indeterminate. If the function is not implemented, **errno** will be set to indicate the error.

ERRORS

No errors are defined.

SEE ALSO

mbstowcs(3C), **strftime(3C)**, **LC_TIME(4)**, **wchar(5)**.

wcstod(3C)**NAME**

`wcstod` - convert wide character string to double-precision number

SYNOPSIS

```
#include <wchar.h>

double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

DESCRIPTION

The `wcstod()` function converts the initial portion of the wide character string pointed to by `nptr` to double representation. First it decomposes the input wide character string into three parts: an initial, possibly empty, sequence of white-space wide character codes (as specified by the `iswspace()` function); a subject sequence interpreted as a floating-point constant; and a final wide character string of one or more unrecognized wide character codes, including the terminate null wide character code of the input wide character string. Then it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional `+` or `-` sign, then a non-empty sequence of digits optionally containing a radix, then an optional exponent part. An exponent part consists of `e` or `E`, followed by an optional sign, followed by one or more decimal digits. 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 codes, or if the first wide character code that is not white space other than a sign, a digit or a radix.

If the subject sequence has the expected form, the sequence of wide character codes starting with the first digit or the radix (whichever occurs first) is interpreted as a floating constant as defined in the C language, except that the radix is used in place of a period, and that if neither an exponent part nor a radix appears, a radix is assumed to follow the last digit in the wide character string. 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.

The radix is defined in the programs locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix is not defined, the radix defaults to a period (`.`).

In other than POSIX locale, other 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 `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The `wcstod()` function will not change the setting of `errno` if successful.

Because zero 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 `wcstod()`, then check `errno` and if it is non-zero, assume an error has occurred.

RETURN VALUE

The `wcstod()` function returns the converted value, if any. If no conversion could be performed, zero is returned and `errno` may be set to `EINVAL`.

If the correct value is outside the range of representable values, `±HUGE_VAL` is returned (according to the sign of the value), and `errno` is set to `ERANGE`.

If the correct value would cause underflow, zero is returned, and `errno` is set to `ERANGE`.

ERRORS

The `wcstod()` function will fail if:

`ERANGE`

The value to be returned would cause overflow or underflow.

The `wcstod()` function may fail if:

`EINVAL`

No conversion could be performed.

SEE ALSO

`iswspace(3C)`, `localeconv(3C)`, `setlocale(3C)`, `wcstol(3C)`, `scanf(3S)`, `wchar(5)`.

NAME

`wcstol`, `wcstoul` - convert wide character string to long integer

SYNOPSIS

```
#include <wchar.h>
□
□
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr,
□□□□□□□□□□□□□□□□□□□□□□□□ int base);
```

DESCRIPTION

The `wcstol()` function converts the initial portion of the wide character string pointed to by `nptr` to long int representation. First it decomposes the input wide character string into three parts: an initial, possibly empty, sequence of white-space wide character codes (as specified by the `iswspace()` function), a subject sequence interpreted as an integer represented in some radix determined by the value of `base`; and 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 it attempts to convert the subject sequence to an integer, and returns the result. Because zero, `LONG_MIN` and `LONG_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 `wcstol()`, then check `errno` and if it is zero, assume an error has occurred.

The `wcstol()` function will not change the setting of `errno` if successful.

The `wcstoul()` function converts the initial portion of the wide character string pointed to by `nptr` to unsigned long int representation. First it decomposes the input wide character string into three parts: an initial possibly empty, sequence of white-space wide character codes (as specified by the `iswspace()` function), a subject sequence interpreted as an integer represented in some radix determined by the value of `base`; and 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 it attempts to convert the subject sequence to an integer, and returns the result. Because zero and `LONG_MAX` are returned on error and zero is also a valid return on success, an application wishing to check for error situations should set `errno` to 0, then call `wcstoul()`, then check `errno` and if it is zero, assume an error has occurred. Unlike `wcstod()` and `wcstol()`, `wcstoul()` must always return a non-negative number; so, using the return value of `wcstoul()` for out-of-range numbers with `wcstoul()` could cause more severe problems than just of precision if those numbers can ever be negative.

The `wcstoul()` function will not change the setting of `errno` if successful.

If the value of `base` is zero, 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 decimal digits 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 with 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 wide character code that is not 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 white-space is other than a sign or a permitted letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, 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.

RETURN VALUE

Upon successful completion the `wcstol()` and `wcstoul()` functions return the converted value, if any. If no conversion could be performed zero is returned and `errno` may be set to indicate the error. If the correct value is outside the range of the representable values, `LONG_MAX`, `LONG_MIN` or `ULONG_MAX` (according to the used function and the sign of the value) is returned and `errno` is set to `ERANGE`.

ERRORS

The `wcstol()` and `wcstoul()` functions will fail if:

`EINVAL`

The value of *base* is not supported.

`ERANGE`

The value to be returned is not representable.

The `wcstol()` and `wcstoul()` functions may fail if:

`EINVAL`

No conversion could be performed.

SEE ALSO

`iswalpha(3C)`, `wcstod(3C)`, `scanf(3S)`, `wchar(5)`.

NAME

`wcsxfrm` - wide character string transformation

SYNOPSIS

```
#include <wchar.h>

size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

DESCRIPTION

The `wcsxfrm()` function transforms the wide character string pointed to by `ws2` and places the resulting wide character string into the array pointed to by `ws1`. The transformation is such that if the `wscmp()` function is applied to two transformed wide strings, it returns a value greater than, equal to or less than zero, corresponding to the result of the `wscoll()` 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 terminating null wide character code. If `n` is zero, `ws1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

The `wcsxfrm()` function will not change the setting of `errno` if successful.

RETURN VALUE

The `wcsxfrm()` function returns 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, the `wcsxfrm()` function returns `(size_t)-1`, and sets `errno` to indicate the error.

ERRORS

The `wcsxfrm()` function may fail if:

`EINVAL`

The wide-character string pointed to by `ws2` contains wide-character codes outside the domain of the collating sequence.

APPLICATION USAGE

The transformation function is such that two transformed wide character strings can be ordered by the `wscmp()` function as appropriate to collating sequence information in the programs locale (category `LC_COLLATE`).

The fact that when `n` is zero, `ws1` is permitted to be a null pointer, is useful to determine the size of the `ws1` array prior to making the transformation.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to zero, then call `wcsxfrm()`, then check `errno`.

SEE ALSO

[wscmp\(3C\)](#), [wscoll\(3C\)](#), [wchar\(5\)](#).

NAME

wctob - wide-character 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 multibyte character representation is a single-byte when in the initial shift state.

The behaviour of this function is affected by the `LC_CTYPE` category of the current locale.

RETURN VALUE

The `wctob()` returns `WEOF` if `c` does not correspond to a multibyte character with length 1 in the initial shift state. Otherwise, it returns the single-byte representation of that character.

ERRORS

No errors are defined.

SEE ALSO

[btowc\(3C\)](#), [stdio\(5\)](#), [wchar\(5\)](#).

NAME

wctrans - wide-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()`. 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()` that modifies the category `LC_CTYPE`.

RETURN VALUE

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()`.

ERRORS

The `wctrans()` function may fail if:

`EINVAL`

The character mapping name pointed to by *charclass* is not valid in the current locale.

SEE ALSO

[towctrans\(3C\)](#), [locale\(5\)](#), [wctype\(5\)](#).

NAME

wctype - define character class

SYNOPSIS

```
#include <wctype.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", "graph", "lower", "print", "space", "upper" and "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()`. The `wctype()` function determines values of `wctype_t` according to the rules of the coded character set defined by character type information in the programs locale (category `LC_CTYPE`). The values returned by `wctype()` are valid until a call to `setlocale()` that modifies the category `LC_CTYPE`.

RETURN VALUE

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()`.

ERRORS

No errors are defined.

SEE ALSO

[wtype\(3C\)](#), [wctype\(5\)](#).

NAME

`wcwidth`, `wcswidth` - number of column positions of a widecharacter code

SYNOPSIS

```
#include <wchar.h>
int wcwidth(wchar_t wc);
int wcswidth(const wchar_t *pwcs, size_t n);
```

DESCRIPTION

The `wcwidth()` function determines the number of column positions required for the wide character code `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.

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`.

RESULT

The `wcwidth()` function either returns zero (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).

The `wcswidth()` function either returns zero (if `pwcs` is a null wide character code), or returns the number of column positions to be occupied by the wide character string `pwcs`, or returns `-1` (if any wide character code in the wide character string pointed to by `pwcs` is not a printing wide character code).

SEE ALSO

[wchar\(5\)](#).

NAME

wordexp, wordfree - perform word expansions

SYNOPSIS

```
#include <wordexp.h>
int wordexp(const char *words, wordexp_t *pwordexp, int flags);
void wordfree(wordexp_t *pwordexp);
```

DESCRIPTION

The `wordexp()` function performs word expansions and places the list of expanded words into the structure pointed to by `pwordexp`.

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 (command substitution allows the output of a command to be substituted in place of the command name itself). 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`.

The structure `wordexp_t` is defined in the header `wordexp.h` and includes at least the following members:

Member Type	Member Name	Description
size_t	we_wordc	Count of words matched by words
char **	we_wordv	Pointer to list of expanded words
size_t	we_offs	ots to reserve at the beginning of <code>pwordexp->we_wordv</code>

The `wordexp()` function stores the number of generated words into `pword->we_wordc` and a pointer to a list of pointers to words in `pwordexp->we_wordv`. Each individual field created during field splitting or pathname expansion is a separate word in the `pwordexp->we_wordv` list. The first pointer after the last word pointer will be a null pointer. The expansion of special parameters (like `*`, `@`, `#`, `?`, `-`, `$`, `!`, `0`) is undefined.

Field Splitting

After parameter expansion, command substitution and arithmetic expansion the shell will scan the results of expansions and substitutions that did not occur in double-quotes for field-splitting and multiple fields can result. The shell will treat each character of the `IFS` (environment variable for input field separator) as a delimiter and use the delimiters to split the results of parameter expansion and command substitution into fields.

1. If the value of `IFS` is a space, tab and newline character, or if it is unset, any sequence of space, tab or newline characters at the beginning or end of the input will be ignored and any sequence of those characters within the input will delimit a field.
2. If the value of `IFS` is null, no field splitting will be performed.
3. Otherwise, the following rules will be applied in sequence. The term "`IFS` white space" is used to mean any sequence (zero or more instances) of white-space characters that are in the `IFS` value.
 - `IFS` white space is ignored at the beginning and end of the input.

- Each occurrence in the input of an `IFS` character that is not `IFS` white space, along with any adjacent `IFS` white space, will delimit a field.
- Non-zero length `IFS` white space will delimit a field.

Pathname Expansion

After field splitting, if `set -f` is not in effect, each field in the resulting command line will be expanded using the pattern matching notation.

Word Expansions

The order of word expansions is as follows:

1. Tilde expansion, parameter expansion, command substitution and arithmetic expansion are performed beginning to end.
2. Field splitting is performed on the portions of the fields generated by step 1, unless `IFS` is null.
3. Pathname expansion is performed, unless `set -f` is in effect.
4. Quote removal always is performed last.

If the complete expansion appropriate for a word results in an empty field, that empty field will be deleted from the list of fields that form the completely expanded command, unless the original word contained single-quote or double-quote characters.

The `$` character is used to introduce parameter expansion, command substitution or arithmetic evaluation. If an unquoted `$` is followed by a character that is either not numeric, the name of one of the special parameters, a valid first character of a variable name, a left curly brace (`{}`) or a left parenthesis, the result is unspecified.

`IFS` is used for performing field splitting on the results of parameter and command substitution; it is not used for splitting all fields. Previous versions of the shell used it for splitting all fields during field splitting, but this has severe problems because the shell can no longer parse its own script. There are also important security implications caused by this behavior. All useful applications of `IFS` use it for parsing input of the `read` utility and for splitting the results of parameter and command substitution.

The `wordexp()` function is intended to be used by an application that wants to do all of the shells expansions on a word or words obtained from a user. For example, if the application prompts for a filename (or a 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.

It is the callers 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()`.

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 other 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:
 - zero or more null pointers, as specified by `WRDE_DOOFFS` and `pwordexp->we_offs`
 - pointers to the words that were in the `pwordexp->we_wordv` list before the call, in the same order as before
 - 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 appropriate 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 `WRDE_NOCMD` flag is provided for applications that, for security or other reasons, want to prevent a user from executing shell commands. Disallowing unquoted shell special characters also prevents unwanted side effects such as executing a command or writing a file.

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.

RESULT

On successful completion, `wordexp()` returns zero.

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.

SEE ALSO

`fnmatch(3C)`, `glob(3C)`, `wordexp(5)`.

NAME

wstring:
 wscat, wcsncat, wcschr, wcsrchr, wcscmp, wcsncmp, wcsncpy, wcsnncpy, wcssp, wcscsp, wcslen, wcpbrk, wcstok, wswcs, wcsstr - wide character string operations

SYNOPSIS

```
#include <wchar.h>

wchar_t *wscat(wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);
wchar_t *wcschr(const wchar_t *ws1, wint_t wc);
wchar_t *wcsrchr(const wchar_t *ws, wint_t wc);
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 *wcsncpy(wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcsnncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
size_t wcssp(const wchar_t *ws1, const wchar_t *ws2);
size_t wcscsp(const wchar_t *ws1, const wchar_t *ws2);
size_t wcslen(const wchar_t *ws);
wchar_t wcpbrk(const wchar_t *ws1, const wchar_t *ws2);
wchar_t wswcs(const wchar_t *ws1, const wchar_t *ws2);
wchar_t wcsstr(const wchar_t *ws1, const wchar_t *ws2);
wchar_t wcstok(wchar_t *ws1, const wchar_t *ws2, wchar_t **ptr);
XPG4 mode:
wchar_t *wcstok(wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

The `wscat()` function concatenates two wide character strings. In fact it appends 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.

The `wcsncat()` function appends 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. If copying takes place between object that overlap, the behavior is undefined.

The `wcschr()` function is a wide character string scanning operation. It locates the first occurrence of `wc` in the wide character string pointed to by `ws1`. 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.

The `wcsrchr()` function locates 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 of `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.

The `wcscmp()` function compares 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.

The `wcsncmp()` function compares 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.

The `wcscpy()` function copies 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 `wcsncpy()` function copies 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 there is no null wide character code in the first *n* wide character codes of the array pointed to by *ws2*, the result will not be null-terminated. 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 *n* wide character codes in all are written.

The `wcsspn()` function gets the length of a wide substring. It computes 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*.

The `wcscspn()` function gets the length of a complementary wide substring. It computes 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*.

The `wcslen()` function computes the number of wide character codes in the wide character string to which *ws* points, not including the terminating null wide character code.

The `wcspbrk()` function scans wide character strings for wide character codes. It locates 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*.

A sequence of calls to the `wcstok()` function breaks 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*. 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. There is no third argument in XPG4 mode.

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 the `wcstok()` function returns a null pointer. If such a wide character code is found, it is start of the first token.

The `wcstok()` function then searches from there 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()` function saves 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 for the first argument, starts searching for the saved pointer and behaves as described above.

The implementation will behave as if no function calls the `wcstok()` function.

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*.

The `wcsstr()` function locates the first occurrence in the wide character string pointed to by `ws1` of the sequence of wide characters codes (excluding the terminating null wide character code) in the wide character string pointed to by `ws2`.

RETURN VALUES

The `wcscat()` function returns `s1`; no return value is reserved to indicate an error.

The `wcsncat()` function returns `ws1`; no return value is reserved to indicate an error.

Upon completion, `wcschr()` returns a pointer to the wide character code, or a null pointer if the wide character code is not found.

Upon successful completion, `wcsrchr()` returns a pointer to the wide character code or a null pointer if `wc` does not occur in the wide character string.

Upon completion, `wscmp()` returns an integer greater than, equal to or less than 0, 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` respectively.

Upon successful completion, `wcsncmp()` returns an integer greater than, equal to or less than 0, 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` respectively.

The `wcscopy()` function returns `ws1`; no return value is reserved to indicate an error.

The `wcsncpy()` function returns `ws1`; no return value is reserved to indicate an error.

The `wcsspn()` function returns the length of `ws1`; no return value is reserved to indicate an error.

The `wcscspn()` function returns the length of `ws1`; no return value is reserved to indicate an error.

The `wcslen()` function returns `ws`; no return value is reserved to indicate an error.

Upon successful completion, `wcspbrk()` returns a pointer to the wide character code or a null pointer if no wide character code from `ws2` occurs in `ws1`.

Upon successful completion, the `wcstok()` function returns a pointer to the first wide character code of a token. Otherwise, if there is no token, the function returns a null pointer.

Upon successful completion, the `wcswcs()` 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`.

Upon successful completion, the `wcsstr()` 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`.

ERRORS

No errors are defined.

SEE ALSO

[wchar\(5\)](#).

NAME

wtype:
 iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct,
 iswspace, iswupper, iswxdigit, iswctype - classification of wide character codes

SYNOPSIS

```
#include <wctype.h>
int iswalnum(wint_t wc);
int iswalpha(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);
int iswctype(wint_t wc, wctype_t desc);□
```

DESCRIPTION

To ensure applications portability, especially across natural languages, only the following functions and the `setlocale()` function should be used for classification of wide character codes.

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 value, the behavior is undefined.

The `iswalnum()` function tests whether `wc` is a wide character code representing a character of class `alpha` or `digit` in the programs current locale.

The `iswalpha()` function tests whether `wc` is a wide character code representing a character of class `alpha` in the programs current locale.

The `iswcntrl()` function tests whether `wc` is a wide character code representing a character of class `control` in the programs current locale.

The `iswdigit()` function tests whether `wc` is a wide character code representing a character of class `digit` in the programs current locale.

The `iswgraph()` function tests whether `wc` is a wide character code representing a character of class `graph` in the programs current locale.

The `iswlower()` function tests whether `wc` is a wide character code representing a character of class `lower` in the programs current locale.

The `iswprint()` function tests whether `wc` is a wide character code representing a character of class `print` in the programs current locale.

The `iswpunct()` function tests whether `wc` is a wide character code representing a character of class `punct` in the programs current locale.

The `iswspace()` function tests whether `wc` is a wide character code representing a character of class `space` in the programs current locale.

The `iswupper()` function tests whether `wc` is a wide character code representing a character of class `upper` in the programs current locale.

The `iswxdigit()` function tests whether `wc` is a wide character code representing a character of class `xdigit` in the programs current locale.

The `iswctype()` function determines whether the wide character `wc` has the property described in `desc`. The current setting of the `LC_CTYPE` category shall be the same as during the call to `wctype()` that returned the value `desc`.

RETURN VALUE

The `wtype` functions return non-zero if `wc` is the wide character code it is tested for; otherwise they all return zero.

ERRORS

No errors are defined.

SEE ALSO

`setlocale(3C)`, `wctype(5)`.

Functions for the ELF format from the library `libelf(3E)`

NAME

`elf` - object file access library

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
```

DESCRIPTION

Functions in the ELF access library let a program manipulate ELF (Executable and Linking Format) object files, archive files, and archive members. The header file provides type and function declarations for all library services.

Programs communicate with many of the higher-level routines using an "ELF descriptor". That is, when the program starts working with a file, `elf_begin` creates an ELF descriptor through which the program manipulates the structures and information in the file. These ELF descriptors can be used both to read and to write files. After the program establishes an ELF descriptor for a file, it may then obtain *section descriptors* to manipulate the sections of the file [see `elf_getscn(3E)`]. Sections hold the bulk of an object file's real information, such as text, data, the symbol table, and so on. A section descriptor belongs to a particular ELF descriptor, just as a section belongs to a file. Finally, *data descriptors* are available through section descriptors, allowing the program to manipulate the information associated with a section. A data descriptor belongs to a section descriptor.

Descriptors provide private handles to a file and its pieces. In other words, a data descriptor is associated with one section descriptor, which is associated with one ELF descriptor, which is associated with one file. Although descriptors are private, they give access to data that may be shared. Consider programs that combine input files, using incoming data to create or update another file. Such a program might get data descriptors for an input and an output section. It then could update the output descriptor to reuse the input descriptor's data. That is, the descriptors are distinct, but they could share the associated data bytes. This sharing avoids the space overhead for duplicate buffers and the performance overhead for copying data unnecessarily.

FILE CLASSES

ELF provides a framework in which to define a family of object files, supporting multiple processors and architectures. An important distinction among object files is the *class*, or capacity, of the file. The 32-bit class supports architectures in which a 32-bit object can represent addresses or file sizes as in the following example:

Name	Purpose
Elf32_Addr	Unsigned address
Elf32_Half	Unsigned medium integer
Elf32_Off	Unsigned file offset
Elf32_Sword	Signed large integer
Elf32_Word	Unsigned large integer
unsigned char	Unsigned small integer

Other classes will be defined as necessary, to support larger (or smaller) machines. Some library services deal only with data objects for a specific class, while others are class-independent. To make this distinction clear, library function names reflect their status, as described below.

DATA REPRESENTATIONS

Conceptually, two parallel sets of objects support cross compilation environments. One set corresponds to file contents, while the other set corresponds to the native memory image of the program manipulating the file. Type definitions supplied by the header files work on the native machine, which may have different data encodings (size or byte order) than the target machine. Although native memory objects should be at least as big as the file objects (to avoid information loss), they may be bigger if that is more natural for the host machine.

Translation facilities exist to convert between file and memory representations. Some library routines convert data automatically, while others leave conversion as the program's responsibility. Either way, programs that create object files must write file-typed objects to those files; programs that read object files must take a similar view. See [elf_xlate\(3E\)](#) and [elf_fsize\(3E\)](#) for more information.

Programs may translate data explicitly, taking full control over the object file layout and semantics. If the program prefers not to have and exercise complete control, the library provides a higher-level interface that hides many object file details. [elf_begin](#) and related functions let a program deal with the native memory types, converting between memory objects and their file equivalents automatically when reading or writing an object file.

ELF VERSIONS

Object file versions allow ELF to adapt to new requirements. Three independent versions can be important to a program. First, an application program knows about a particular version by virtue of being compiled with certain header files. Second, the access library similarly is compiled with header files that control what versions it understands. Third, an ELF object file holds a value identifying its version, determined by the ELF version known by the file's creator. Ideally, all three versions would be the same, but they may differ.

- If a program's version is newer than the access library, the program might use information unknown to the library. Translation routines might not work properly, leading to undefined behavior. This condition merits installing a new library.
- The library's version might be newer than the program's and the file's. The library understands old versions, thus avoiding compatibility problems in this case.
- Finally, a file's version might be newer than either the program or the library understands. The program might or might not be able to process the file properly, depending on whether the file has extra information and whether that information can be safely ignored. Again, the safe alternative is to install a new library

that understands the file's version.

To accommodate these differences, a program must use `elf_version` to pass its version to the library, thus establishing the "working version" for the process. Using this, the library accepts data from and presents data to the program in the proper representations. When the library reads object files, it uses each file's version to interpret the data. When writing files or converting memory types to the file equivalents, the library uses the program's working version for the file data.

SYSTEM SERVICES

As mentioned above, `elf_begin` and related routines provide a higher-level interface to ELF files, performing input and output on behalf of the application program. These routines assume a program can hold entire files in memory, without explicitly using temporary files. When reading a file, the library routines bring the data into memory and perform subsequent operations on the memory copy. Programs that wish to read or write large object files with this model must execute on a machine with a large process virtual address space. If the underlying operating system limits the number of open files, a program can use `elf_cntl` to retrieve all necessary data from the file, allowing the program to close the file descriptor and reuse it.

Although the `elf_begin` interfaces are convenient and efficient for many programs, they might be inappropriate for some. In those cases, an application may invoke the `elf_xlate` data translation routines directly. These routines perform no input or output, leaving that as the application's responsibility. By assuming a larger share of the job, an application controls its input and output model.

LIBRARY NAMES

Names associated with the library take several forms.

`elf_name`

These class-independent names perform some service, *name*, for the program.

`elf32_name`

Service names with an embedded class, 32 here, indicate they work only for the designated class of files.

`Elf_Type`

Data types can be class-independent as well, distinguished by *Type*.

`Elf32_Type`

Class-dependent data types have an embedded class name, 32 here.

`ELF_C_CMD`

Several functions take commands that control their actions. These values are members of the `Elf_Cmd` enumeration; they range from zero through `ELF_C_NUM-1`.

`ELF_F_FLAG`

Several functions take flags that control library status and/or actions. Flags are bits that may be combined.

`ELF32_FSZ_TYPE`

These constants give the file sizes in bytes of the basic ELF types for the 32-bit class of files. See `elf_fsize` for more information.

`ELF_K_KIND`

The function `elf_kind` identifies the *KIND* of file associated with an ELF descriptor. These values are members of the `Elf_Kind` enumeration; they range from zero through `ELF_K_NUM-1`.

`ELF_T_TYPE`

When a service function, such as `elf_xlate`, deals with multiple types, names of this form specify the desired *TYPE*. Thus, for example, `ELF_T_EHDR` is directly related to `Elf32_Ehdr`. These values are members of the `Elf_Type` enumeration; they range from zero through `ELF_T_NUM-1`.

NOTES

Information in the ELF header files is separated into common parts and processor-specific parts. A program can make a processor's information available by including the appropriate header file: `<sys/elf_NAME.h>`

where *NAME* matches the processor name as used in the ELF file header.

Symbol	Processor
M32	AT&T WE 32100
SPARC	SPARC
386	Intel 80386
486	Intel 80486
860	Intel 80860
68K	Motorola 68000
88K	Motorola 88000
R2000	MIPS R2000

Other processors will be added to the table as necessary. To illustrate, a program could use the following code to see the processor-specific information for the MIPS R2000.

```
#include <libelf.h>
#include <sys/elf_R2000.h>
```

Without the `<sys/elf_R2000.h>` definition, only the common ELF information would be visible.

SEE ALSO

`elf_begin(3E)`, `elf_cntl(3E)`, `elf_end(3E)`, `elf_fill(3E)`, `elf_flag(3E)`, `elf_fsize(3E)`, `elf_getarhdr(3E)`, `elf_getarsym(3E)`, `elf_getbase(3E)`, `elf_getdata(3E)`, `elf_getehdr(3E)`, `elf_getident(3E)`, `elf_getphdr(3E)`, `elf_getscn(3E)`, `elf_getshdr(3E)`, `elf_hash(3E)`, `elf_kind(3E)`, `elf_next(3E)`, `elf_rand(3E)`, `elf_rawfile(3E)`, `elf_strptr(3E)`, `elf_update(3E)`, `elf_version(3E)`, `elf_xlate(3E)`, `a.out(4)`, `ar(4)`.

elf_begin(3E)

NAME

elf_begin - make a file descriptor

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
Elf *elf_begin(int fildes, Elf_Cmd cmd, Elf *ref);
```

DESCRIPTION

elf_begin(), elf_next(), elf_rand(), and elf_end() work together to process ELF object files, either individually or as members of archives. After obtaining an ELF descriptor from elf_begin(), the program may read an existing file, update an existing file, or create a new file. *fildes* is an open file descriptor that elf_begin() uses for reading or writing. The initial file offset [see [lseek\(2\)](#)] is unconstrained, and the resulting file offset is undefined.

cmd may have the following values.

ELF_C_NULL

elf_begin() returns a null pointer, without opening a new descriptor. *ref* is ignored for this command. See [elf_next\(3E\)](#) and the associated examples for more information.

ELF_C_READ

When a program wishes to examine the contents of an existing file, it should set *cmd* to this value. Depending on the value of *ref*, this command examines archive members or entire files. Three cases can occur.

- If *ref* is a null pointer, elf_begin() allocates a new ELF descriptor and prepares to process the entire file. If the file being read is an archive, elf_begin() also prepares the resulting descriptor to examine the initial archive member on the next call to elf_begin(), as if the program had used elf_next() or elf_rand() to 'move' to the initial member.
- If *ref* is a non-null descriptor associated with an archive file, elf_begin() lets a program obtain a separate ELF descriptor associated with an individual member. The program should have used elf_next() or elf_rand() to position *ref* appropriately (except for the initial member, which elf_begin() prepares; see the example below). In this case, *fildes* should be the same file descriptor used for the parent archive.
- Finally, if *ref* is a non-null ELF descriptor that is not an archive, elf_begin() increments the number of activations for the descriptor and returns *ref*, without allocating a new descriptor and without changing the descriptor's read/write permissions. To terminate the descriptor for *ref*, the program must call elf_end() once for each activation. See [elf_next\(3E\)](#) and the examples for more information.

ELF_C_RDWR

This command duplicates the actions of ELF_C_READ and additionally allows the program to update the file image [see [elf_update\(3E\)](#)]. That is, using ELF_C_READ gives a read-only view of the file, while ELF_C_RDWR lets the program read *and* write the file. ELF_C_RDWR is not valid for archive members. If *ref* is non-null, it must have been created with the ELF_C_RDWR command.

ELF_C_WRITE

If the program wishes to ignore previous file contents, presumably to create a new file, it should set *cmd* to this value. *ref* is ignored for this command.

elf_begin() "works" on all files (including files with zero bytes), providing it can allocate memory for its internal structures and read any necessary information from the file. Programs reading object files thus may call elf_kind() or elf_getehdr() to determine the file type (only object files have an ELF header). If the file is an archive with no more members to process, or an error occurs, elf_begin() returns a null pointer.

Otherwise, the return value is a non-null ELF descriptor.

Before the first call to `elf_begin()`, a program must call `elf_version()` to coordinate versions.

System services

When processing a file, the library decides when to read or write the file, depending on the program's requests. Normally, the library assumes the file descriptor remains usable for the life of the ELF descriptor. If, however, a program must process many files simultaneously and the underlying operating system limits the number of open files, the program can use `elf_cntl()` to let it reuse file descriptors. After calling `elf_cntl()` with appropriate arguments, the program may close the file descriptor without interfering with the library.

All data associated with an ELF descriptor remain allocated until `elf_end()` terminates the descriptor's last activation. After the descriptors have been terminated, the storage is released; attempting to reference such data gives undefined behavior. Consequently, a program that deals with multiple input (or output) files must keep the ELF descriptors active until it finishes with them.

EXAMPLES

A prototype for reading a file appears below. If the file is a simple object file, the program executes the loop one time, receiving a null descriptor in the second iteration. In this case, both `elf` and `arf` will have the same value, the activation count will be two, and the program calls `elf_end()` twice to terminate the descriptor. If the file is an archive, the loop processes each archive member in turn, ignoring those that are not object files.

```
if (elf_version(EV_CURRENT) == EV_NONE)
{
    /* library out of date */
    /* recover from error */
}
cmd = ELF_C_READ;
arf = elf_begin(fildes, cmd, (Elf *)0);
while ((elf = elf_begin(fildes, cmd, arf)) != 0)
{
    if ((ehdr = elf32_getehdr(elf)) != 0)
    {
        /* process the file ... */
    }
    cmd = elf_next(elf);
    elf_end(elf);
}
elf_end(arf);
```

Alternatively, the next example illustrates random archive processing. After identifying the file as an archive, the program repeatedly processes archive members of interest. For simplicity, this example omits error checking and ignores simple object files. Additionally, this fragment preserves the ELF descriptors for all archive members, because it does not call `elf_end()` to terminate them.

```
elf_version(EV_CURRENT);
arf = elf_begin(fildes, ELF_C_READ, (Elf *)0);
if (elf_kind(arf) != ELF_K_AR)
{
    /* not an archive */
}
/* initial processing */
/* set offset = ... for desired member header */
while (elf_rand(arf, offset) == offset)
{
    if ((elf = elf_begin(fildes, ELF_C_READ, arf)) == 0)
        break;
    if ((ehdr = elf32_getehdr(elf)) != 0)
```

```

    {
        /* process archive member ... */
    }
    /* set offset = ... for desired member header */
}

```

The following outline shows how one might create a new ELF file. This example is simplified to show the overall flow.

```

elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR|O_TRUNC|O_CREAT, 0666);
if ((elf = elf_begin(fildes, ELF_C_WRITE, (Elf *)0)) == 0)
    return;
ehdr = elf32_newehdr(elf);
phdr = elf32_newphdr(elf, count);
scn = elf_newscn(elf);
shdr = elf32_getshdr(scn);
data = elf_newdata(scn);
elf_update(elf, ELF_C_WRITE);
elf_end(elf);

```

Finally, the following outline shows how one might update an existing ELF file. Again, this example is simplified to show the overall flow.

```

elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR);
elf = elf_begin(fildes, ELF_C_RDWR, (Elf *)0);

/* add new or delete old information ... */
close(creat("path/name", 0666));
elf_update(elf, ELF_C_WRITE);
elf_end(elf);

```

In the example above, the call to `creat()` truncates the file, thus ensuring the resulting file will have the "right" size. Without truncation, the updated file might be as big as the original, even if information were deleted. The library truncates the file, if it can, with `ftruncate` [see [truncate\(3C\)](#)]. Some systems, however, do not support `ftruncate()`, and the call to `creat()` protects against this.

Notice that both file creation examples open the file with write *and* read permissions. On systems that support `mmap()`, the library uses it to enhance performance, and `mmap()` requires a readable file descriptor. Although the library can use a write-only file descriptor, the application will not obtain the performance advantages of `mmap()`.

NOTES

COFF is an object file format that preceded ELF. When a program calls `elf_begin()` on a COFF file, the library translates COFF structures to their ELF equivalents, allowing programs to read (but not to write) a COFF file as if it were ELF. This conversion happens only to the memory image and *not* to the file itself. After the initial `elf_begin()`, file offsets and addresses in the ELF header, the program headers, and the section headers retain the original COFF values (see `elf_getehdr()`, `elf_getphdr()`, and `elf_getshdr()`). A program may call `elf_update()` to adjust these values (without writing the file), and the library will then present a consistent, ELF view of the file. Data obtained through `elf_getdata()` are translated (the COFF symbol table is presented as ELF, etc.). Data viewed through `elf_rawdata()` undergo no conversion, allowing the program to view the bytes from the file itself.

Some COFF debugging information is not translated, though this does not affect the semantics of a running program.

Although the ELF library supports COFF, programmers are strongly encouraged to recompile their programs, obtaining ELF object files.

SEE ALSO

[creat\(2\)](#), [lseek\(2\)](#), [mmap\(2\)](#), [open\(2\)](#), [truncate\(3C\)](#), [elf\(3E\)](#), [elf_cntl\(3E\)](#), [elf_end\(3E\)](#)

```
, elf_getarhdr(3E), elf_getbase(3E), elf_getdata(3E), elf_getehdr(3E),  
elf_getphdr(3E), elf_getscn(3E), elf_kind(3E), elf_next(3E), elf_rand(3E),  
elf_rawfile(3E), elf_update(3E), elf_version(3E), ar(4).
```

NAME

elf_cntl - control a file descriptor

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
int elf_cntl(Elf *elf, Elf_Cmd cmd);
```

DESCRIPTION

elf_cntl() instructs the library to modify its behavior with respect to an ELF descriptor, *elf*. As [elf_begin\(3E\)](#) describes, an ELF descriptor can have multiple activations, and multiple ELF descriptors may share a single file descriptor. Generally, elf_cntl() commands apply to all activations of *elf*. Moreover, if the ELF descriptor is associated with an archive file, descriptors for members within the archive will also be affected as described below. Unless stated otherwise, operations on archive members do not affect the descriptor for the containing archive.

The *cmd* argument tells what actions to take and may have the following values.

ELF_C_FDDONE

This value tells the library not to use the file descriptor associated with *elf*. A program should use this command when it has requested all the information it cares to use and wishes to avoid the overhead of reading the rest of the file. The memory for all completed operations remains valid, but later file operations, such as the initial elf_getdata() for a section, will fail if the data is not in memory already.

ELF_C_FDREAD

This command is similar to ELF_C_FDDONE, except it forces the library to read the rest of the file. A program should use this command when it must close the file descriptor but has not yet read everything it needs from the file. After elf_cntl() completes the ELF_C_FDREAD command, future operations, such as elf_getdata(), will use the memory version of the file without needing to use the file descriptor.

If elf_cntl() succeeds, it returns zero. Otherwise *elf* was null or an error occurred, and the function returns -1.

NOTES

If the program wishes to use the "raw" operations [see elf_rawdata(), which [elf_getdata\(3E\)](#) describes, and [elf_rawfile\(3E\)](#)] after disabling the file descriptor with ELF_C_FDDONE or ELF_C_FDREAD, it must execute the raw operations explicitly beforehand. Otherwise, the raw file operations will fail. Calling elf_rawfile() makes the entire image available, thus supporting subsequent elf_rawdata() calls.

SEE ALSO

[elf\(3E\)](#), [elf_begin\(3E\)](#), [elf_getdata\(3E\)](#), [elf_rawfile\(3E\)](#).

NAME

elf_end - finish using an object file

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]  
#include <libelf.h>  
int elf_end(Elf *elf);
```

DESCRIPTION

A program uses `elf_end()` to terminate an ELF descriptor, *elf*, and to deallocate data associated with the descriptor. Until the program terminates a descriptor, the data remain allocated. *elf* should be a value previously returned by `elf_begin()`; a null pointer is allowed as an argument, to simplify error handling. If the program wishes to write data associated with the ELF descriptor to the file, it must use `elf_update()` before calling `elf_end()`.

As [elf_begin\(3E\)](#) explains, a descriptor can have more than one activation. Calling `elf_end()` removes one activation and returns the remaining activation count. The library does not terminate the descriptor until the activation count reaches zero. Consequently, a zero return value indicates the ELF descriptor is no longer valid.

SEE ALSO

[elf\(3E\)](#), [elf_begin\(3E\)](#), [elf_update\(3E\)](#).

elf_errmsg(3E)**NAME**

elf_errmsg, elf_errno - error handling

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
const char *elf_errmsg(int err);
int elf_errno(void);□
```

DESCRIPTION

If an ELF library function fails, a program may call `elf_errno()` to retrieve the library's internal error number. As a side effect, this function resets the internal error number to zero, which indicates no error.

`elf_errmsg()` takes an error number, *err*, and returns a null-terminated error message (with no trailing new-line) that describes the problem. A zero *err* retrieves a message for the most recent error. If no error has occurred, the return value is a null pointer (not a pointer to the null string). Using *err* of -1 also retrieves the most recent error, except it guarantees a non-null return value, even when no error has occurred. If no message is available for the given number, `elf_errmsg()` returns a pointer to an appropriate message. This function does not have the side effect of clearing the internal error number.

EXAMPLE

The following fragment clears the internal error number and checks it later for errors. Unless an error occurs after the first call to `elf_errno()`, the next call will return zero.

```
(void)elf_errno():
while (more_to_do)
{
    /* processing ... */
    if ((err = elf_errno()) != 0)
    {
        msg = elf_errmsg(err);
        /* print msg */
    }
}
```

SEE ALSO

[elf\(3E\)](#), [elf_version\(3E\)](#).

NAME

elf_fill - set fill byte

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]  
#include <libelf.h>  
void elf_fill(int fill);
```

DESCRIPTION

Alignment constraints for ELF files sometimes require the presence of "holes". For example, if the data for one section are required to begin on an eight-byte boundary, but the preceding section is too "short", the library must fill the intervening bytes. These bytes are set to the *fill* character. The library uses zero bytes unless the application supplies a value. See [elf_getdata\(3E\)](#) for more information about these holes.

NOTES

An application can assume control of the object file organization by setting the `ELF_F_LAYOUT` bit [see [elf_flag\(3E\)](#)]. When this is done, the library does *not* fill holes.

SEE ALSO

[elf\(3E\)](#), [elf_getdata\(3E\)](#), [elf_flag\(3E\)](#), [elf_update\(3E\)](#).

NAME

elf_flag:
 elf_flagdata, elf_flagehdr, elf_flagelf, elf_flagphdr, elf_flagscn, elf_flagshdr
 - manipulate flags

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
unsigned elf_flagdata(Elf_Data *data, Elf_Cmd cmd, unsigned flags);
unsigned elf_flagehdr(Elf *elf, Elf_Cmd cmd, unsigned flags);
unsigned elf_flagelf(Elf *elf, Elf_Cmd cmd, unsigned flags);
unsigned elf_flagphdr(Elf *elf, Elf_Cmd cmd, unsigned flags);
unsigned elf_flagscn(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);
unsigned elf_flagshdr(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);
```

DESCRIPTION

These functions manipulate the flags associated with various structures of an ELF file. Given an ELF descriptor *elf*, a data descriptor *data*, or a section descriptor *scn*, the functions may set or clear the associated status bits, returning the updated bits. A null descriptor is allowed, to simplify error handling; all functions return zero for this degenerate case.

cmd may have the following values.

ELF_C_CLR

The functions clear the bits that are asserted in *flags*. Only the non-zero bits in *flags* are cleared; zero bits do not change the status of the descriptor.

ELF_C_SET

The functions set the bits that are asserted in *flags*. Only the non-zero bits in *flags* are set; zero bits do not change the status of the descriptor.

Descriptions of the defined *flags* bits appear below.

ELF_F_DIRTY

When the program intends to write an ELF file, this flag asserts the associated information needs to be written to the file. Thus, for example, a program that wished to update the ELF header of an existing file would call `elf_flagehdr()` with this bit set in *flags* and *cmd* equal to `ELF_C_SET`. A later call to `elf_update()` would write the marked header to the file.

ELF_F_LAYOUT

Normally, the library decides how to arrange an output file. That is, it automatically decides where to place sections, how to align them in the file, etc. If this bit is set for an ELF descriptor, the program assumes responsibility for determining all file positions. This bit is meaningful only for `elf_flagelf()` and applies to the entire file associated with the descriptor.

When a flag bit is set for an item, it affects all the subitems as well. Thus, for example, if the program sets the `ELF_F_DIRTY` bit with `elf_flagelf()`, the entire logical file is "dirty".

EXAMPLES

The following fragment shows how one might mark the ELF header to be written to the output file.

```
ehdr = elf32_getehdr(elf);
/* dirty ehdr ... */
elf_flagehdr(elf, ELF_C_SET, ELF_F_DIRTY);
```

SEE ALSO

`elf(3E)`, `elf_end(3E)`, `elf_getdata(3E)`, `elf_getehdr(3E)`, `elf_update(3E)`.

NAME

elf_fsize:
elf32_fsize - return the size of an object file

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]  
#include <libelf.h>  
size_t elf32_fsize(Elf_Type type, size_t count, unsigned ver);
```

DESCRIPTION

elf32_fsize() gives the size in bytes of the 32-bit file representation of *count* data objects with the given *type*. The library uses version *ver* to calculate the size [see [elf\(3E\)](#) and [elf_version\(3E\)](#)].
Constant values are available for the sizes of fundamental types.

Elf_Type	File size	Memory size
ELF_T_ADDR	ELF32_FSZ_ADDR	sizeof(Elf32_Addr)
ELF_T_BYTE	1	sizeof(unsigned char)
ELF_T_HALF	ELF32_FSZ_HALF	sizeof(Elf32_Half)
ELT_T_OFF	ELF32_FSZ_OFF	sizeof(Elf32_Off)
ELF_T_SWORD	ELF32_FSZ_SWOR D	sizeof(Elf32_Sword)
ELF_T_WORD	ELF32_FSZ_WORD	sizeof(Elf32_Word)

elf32_fsize() returns zero if the value of *type* or *ver* is unknown. See [elf_xlate\(3E\)](#) for a list of the *type* values.

SEE ALSO

[elf\(3E\)](#), [elf_version\(3E\)](#), [elf_xlate\(3E\)](#).

elf_getarhdr(3E)**NAME**

elf_getarhdr - retrieve archive member header

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
Elf_Arhdr *elf_getarhdr(Elf *elf);
```

DESCRIPTION

elf_getarhdr() returns a pointer to an archive member header, if one is available for the ELF descriptor *elf*. Otherwise, no archive member header exists, an error occurred, or *elf* was null; elf_getarhdr() then returns a null value. The header includes the following members.

```
char          *ar_name;
time_t        ar_date;
long          ar_uid;
long          ar_gid;
unsigned long ar_mode;
off_t         ar_size;
char          *ar_rawname;
```

An archive member name, available through *ar_name*, is a null-terminated string, with the *ar* format control characters removed. The *ar_rawname* member holds a null-terminated string that represents the original name bytes in the file, including the terminating slash and trailing blanks as specified in the archive format.

In addition to "regular" archive members, the archive format defines some special members. All special member names begin with a slash (/), distinguishing them from regular members (whose names may not contain a slash). These special members have the names (*ar_name*) defined below.

- / This is the archive symbol table. If present, it will be the first archive member. A program may access the archive symbol table through elf_getarsym(). The information in the symbol table is useful for random archive processing [see elf_rand(3E)].
- // This member, if present, holds a string table for long archive member names. An archive member's header contains a 16-byte area for the name, which may be exceeded in some file systems. The library automatically retrieves long member names from the string table, setting *ar_name* to the appropriate value.

Under some error conditions, a member's name might not be available. Although this causes the library to set *ar_name* to a null pointer, the *ar_rawname* member will be set as usual.

SEE ALSO

elf(3E), elf_begin(3E), elf_getarsym(3E), elf_rand(3E), ar(4).

NAME

elf_getarsym - retrieve archive symbol table

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
Elf_Arsym *elf_getarsym(Elf *elf, size_t *ptr);
```

DESCRIPTION

elf_getarsym() returns a pointer to the archive symbol table, if one is available for the ELF descriptor *elf*. Otherwise, the archive doesn't have a symbol table, an error occurred, or *elf* was null; elf_getarsym() then returns a null value. The symbol table is an array of structures that include the following members.

```
char          *as_name;
size_t        as_off;
unsigned long  as_hash;
```

These members have the following semantics:

as_name

A pointer to a null-terminated symbol name resides here.

as_off

This value is a byte offset from the beginning of the archive to the member's header. The archive member residing at the given offset defines the associated symbol. Values in *as_off* may be passed as arguments to elf_rand() to access the desired archive member.

as_hash

This is a hash value for the name, as computed by elf_hash().

If *ptr* is non-null, the library stores the number of table entries in the location to which *ptr* points. This value is set to zero when the return value is null. The table's last entry, which is included in the count, has a null *as_name*, a zero value for *as_off*, and ~0UL for *as_hash*.

SEE ALSO

elf(3E), elf_getarhdr(3E), elf_hash(3E), elf_rand(3E), ar(4).

NAME

elf_getbase - get the base offset for an object file

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]  
#include <libelf.h>  
off_t elf_getbase(Elf *elf);
```

DESCRIPTION

elf_getbase() returns the file offset of the first byte of the file or archive member associated with *elf*, if it is known or obtainable, and -1 otherwise. A null *elf* is allowed, to simplify error handling; the return value in this case is -1. The base offset of an archive member is the beginning of the member's information, *not* the beginning of the archive member header.

SEE ALSO

elf(3E), elf_begin(3E), ar(4).

elf_getdata(3E)**NAME**

`elf_getdata`, `elf_newdata`, `elf_rawdata` - get section data

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
Elf_Data *elf_getdata(Elf_Scn *scn, Elf_Data *data);
Elf_Data *elf_newdata(Elf_Scn *scn);
Elf_Data *elf_rawdata(Elf_Scn *scn, Elf_Data *data);
```

DESCRIPTION

These functions access and manipulate the data associated with a section descriptor, *scn*. When reading an existing file, a section will have a single data buffer associated with it. A program may build a new section in pieces, however, composing the new data from multiple data buffers. For this reason, "the" data for a section should be viewed as a list of buffers, each of which is available through a data descriptor.

`elf_getdata()` lets a program step through a section's data list. If the incoming data descriptor, *data*, is null, the function returns the first buffer associated with the section. Otherwise, *data* should be a data descriptor associated with *scn*, and the function gives the program access to the next data element for the section. If *scn* is null or an error occurs, `elf_getdata()` returns a null pointer.

`elf_getdata()` translates the data from file representations into memory representations [see [elf_xlate\(3E\)](#)] and presents objects with memory data types to the program, based on the file's class [see [elf\(3E\)](#)]. The working library version [see [elf_version\(3E\)](#)] specifies what version of the memory structures the program wishes `elf_getdata()` to present.

`elf_newdata()` creates a new data descriptor for a section, appending it to any data elements already associated with the section. As described below, the new data descriptor appears empty, indicating the element holds no data. For convenience, the descriptor's type (*d_type* below) is set to `ELF_T_BYTE`, and the version (*d_version* below) is set to the working version. The program is responsible for setting (or changing) the descriptor members as needed. This function implicitly sets the `ELF_F_DIRTY` bit for the section's data [see [elf_flag\(3E\)](#)]. If *scn* is null or an error occurs, `elf_newdata()` returns a null pointer.

`elf_rawdata()` differs from `elf_getdata()` by returning only uninterpreted bytes, regardless of the section type. This function typically should be used only to retrieve a section image from a file being read, and then only when a program must avoid the automatic data translation described below. Moreover, a program may not close or disable [see [elf_cntl\(3E\)](#)] the file descriptor associated with *elf* before the initial raw operation, because `elf_rawdata()` might read the data from the file to ensure it doesn't interfere with `elf_getdata()`. See [elf_rawfile\(3E\)](#) for a related facility that applies to the entire file. When `elf_getdata()` provides the right translation, its use is recommended over `elf_rawdata()`. If *scn* is null or an error occurs, `elf_rawdata()` returns a null pointer.

The `Elf_Data` structure includes the following members.

```
void          *d_buf;
Elf_Type      d_type;
size_t        d_size;
off_t         d_off;
size_t        d_align;
unsigned      d_version;
```

These members are available for direct manipulation by the program. Descriptions appear below.

d_buf A pointer to the data buffer resides here. A data element with no data has a null pointer.

d_type

This member's value specifies the type of the data to which `d_buf` points. A section's type determines how to interpret the section contents, as summarized below.

`d_size`

This member holds the total size, in bytes, of the memory occupied by the data. This may differ from the size as represented in the file. The size will be zero if no data exist (see the discussion of `SHT_NOBITS`).

`d_off`

This member gives the offset, within the section, at which the buffer resides. This offset is relative to the file's section, not the memory object's.

`d_align`

This member holds the buffer's required alignment, from the beginning of the section. That is, `d_off` will be a multiple of this member's value. For example, if this member's value is four, the beginning of the buffer will be four-byte aligned within the section. Moreover, the entire section will be aligned to the maximum of its constituents, thus ensuring appropriate alignment for a buffer within the section and within the file.

`d_version`

This member holds the version number of the objects in the buffer. When the library originally read the data from the object file, it used the working version to control the translation to memory objects.

Data alignment

As mentioned above, data buffers within a section have explicit alignment constraints. Consequently, adjacent buffers sometimes will not abut, causing "holes" within a section. Programs that create output files have two ways of dealing with these holes.

First, the program can use `elf_fill()` to tell the library how to set the intervening bytes. When the library must generate gaps in the file, it uses the fill byte to initialize the data there. The library's initial fill value is zero, and `elf_fill()` lets the application change that.

Second, the application can generate its own data buffers to occupy the gaps, filling the gaps with values appropriate for the section being created. A program might even use different fill values for different sections. For example, it could set text sections' bytes to no-operation instructions, while filling data section holes with zero. Using this technique, the library finds no holes to fill, because the application eliminated them.

Section and memory types

`elf_getdata()` interprets sections' data according to the section type, as noted in the section header available through `elf_getshdr()`. The following table shows the section types and how the library represents them with memory data types for the 32-bit file class. Other classes would have similar tables. By implication, the memory data types control translation by `elf_xlate()`.

Section type	Elf_Type	32-Bit Type
<code>SHT_DYNAMIC</code>	<code>ELF_T_DYN</code>	<code>Elf32_Dyn</code>
<code>SHT_DYNSYM</code>	<code>ELF_T_SYM</code>	<code>Elf32_Sym</code>
<code>SHT_HASH</code>	<code>ELF_T_WORD</code>	<code>Elf32_Word</code>
<code>SHT_NOBITS</code>	<code>ELF_T_BYTE</code>	unsigned char
<code>SHT_NOTE</code>	<code>ELF_T_BYTE</code>	unsigned char
<code>SHT_NULL</code>	none	none
<code>SHT_PROGBITS</code>	<code>ELF_T_BYTE</code>	unsigned char
<code>SHT_REL</code>	<code>ELF_T_REL</code>	<code>Elf32_Rel</code>
<code>SHT_RELA</code>	<code>ELF_T_RELA</code>	<code>Elf32_Rela</code>
<code>SHT_STRTAB</code>	<code>ELF_T_BYTE</code>	unsigned char
<code>SHT_SYMTAB</code>	<code>ELF_T_SYM</code>	<code>Elf32_Sym</code>
other	<code>ELF_T_BYTE</code>	unsigned char

`elf_rawdata()` creates a buffer with type `ELF_T_BYTE`.

As mentioned above, the program's working version controls what structures the library creates for the application. The library similarly interprets section types according to the versions. If a section type "belongs" to a version newer than the application's working version, the library does not translate the section data. Because the application cannot know the data format in this case, the library presents an untranslated buffer of type `ELF_T_BYTE`, just as it would for an unrecognized section type.

A section with a special type, `SHT_NOBITS`, occupies no space in an object file, even when the section header indicates a non-zero size. `elf_getdata()` and `elf_rawdata()` "work" on such a section, setting the data structure to have a null buffer pointer and the type indicated above. Although no data is present, the `d_size` value is set to the size from the section header. When a program is creating a new section of type `SHT_NOBITS`, it should use `elf_newdata()` to add data buffers to the section. These "empty" data buffers should have the `d_size` members set to the desired size and the `d_buf` members set to null.

EXAMPLES

The following fragment obtains the string table that holds section names (ignoring error checking). See [elf_strptr\(3E\)](#) for a variation of string table handling.

```
ehdr = elf32_getehdr(elf);
scn = elf_getscn(elf, (size_t)ehdr->e_shstrndx);
shdr = elf32_getshdr(scn);
if (shdr->sh_type != SHT_STRTAB)
{
    /* not a string table */
}
data = 0;
if ((data = elf_getdata(scn, data)) == 0 && data->d_size == 0)
{
    /* error or no data */
}
```

The `e_shstrndx` member in an ELF header holds the section table index of the string table. The program gets a section descriptor for that section, verifies it is a string table, and then retrieves the data. When this fragment finishes, `data->d_buf` points at the first byte of the string table, and `data->d_size` holds the string table's size in bytes.

SEE ALSO

[elf\(3E\)](#), [elf_cntl\(3E\)](#), [elf_fill\(3E\)](#), [elf_flag\(3E\)](#), [elf_getehdr\(3E\)](#), [elf_getscn\(3E\)](#), [elf_getshdr\(3E\)](#), [elf_rawfile\(3E\)](#), [elf_version\(3E\)](#), [elf_xlate\(3E\)](#).

elf_getehdr(3E)**NAME**

`elf_getehdr` - retrieve class-dependent object file header

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
Elf32_Ehdr *elf32_getehdr(Elf *elf);
□
□
Elf32_Ehdr *elf32_newehdr(Elf *elf);
```

DESCRIPTION

For a 32-bit class file, `elf32_getehdr()` returns a pointer to an ELF header, if one is available for the ELF descriptor `elf`. If no header exists for the descriptor, `elf32_newehdr()` allocates a "clean" one, but it otherwise behaves the same as `elf32_getehdr()`. It does not allocate a new header if one exists already. If no header exists (for `elf_getehdr`), one cannot be created (for `elf_newehdr`), a system error occurs, the file is not a 32-bit class file, or `elf` is null, both functions return a null pointer.

The header includes the following members.

```
unsigned char    e_ident[EI_NIDENT];
Elf32_Half      e_type;
Elf32_Half      e_machine;
Elf32_Word      e_version;
Elf32_Addr      e_entry;
Elf32_Off       e_phoff;
Elf32_Off       e_shoff;
Elf32_Word      e_flags;
Elf32_Half      e_ehsize;
Elf32_Half      e_phentsize;
Elf32_Half      e_phnum;
Elf32_Half      e_shentsize;
Elf32_Half      e_shnum;
Elf32_Half      e_shstrndx;
```

`elf32_newehdr()` automatically sets the `ELF_F_DIRTY` bit [see [elf_flag\(3E\)](#)]. A program may use `elf_getident()` to inspect the identification bytes from a file.

SEE ALSO

[elf\(3E\)](#), [elf_begin\(3E\)](#), [elf_flag\(3E\)](#), [elf_getident\(3E\)](#).

elf_getident(3E)**NAME**

`elf_getident` - retrieve file identification data

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
char *elf_getident(Elf *elf, size_t *ptr);
```

DESCRIPTION

As [elf\(3E\)](#) explains, ELF provides a framework for various classes of files, where basic objects may have 32 bits, 64 bits, etc. To accommodate these differences, without forcing the larger sizes on smaller machines, the initial bytes in an ELF file hold identification information common to all file classes. Every ELF header's `e_ident` has `EI_NIDENT` bytes with the following interpretation.

e_ident Index	Value	Purpose
EI_MAG0	ELFMAG0	File identification
EI_MAG1	ELFMAG1	
EI_MAG2	ELFMAG2	
EI_MAG3	ELFMAG3	
EI_CLASS	ELFCLASSNONE ELFCLASS32 ELFCLASS64	File class
EI_DATA	ELFDATANONE ELFDATA2LSB ELFDATA2MSB	Data encoding
EI_VERSION	EV_CURRENT	File version
7-15	0	Unused, set to zero

Other kinds of files [see [elf_kind\(3E\)](#)] also may have identification data, though they would not conform to `e_ident`.

`elf_getident()` returns a pointer to the file's "initial bytes". If the library recognizes the file, a conversion from the file image to the memory image may occur. In any case, the identification bytes are guaranteed not to have been modified, though the size of the unmodified area depends on the file type. If `ptr` is non-null, the library stores the number of identification bytes in the location to which `ptr` points. If no data is present, `elf` is null, or an error occurs, the return value is a null pointer, with zero optionally stored through `ptr`.

SEE ALSO

[elf\(3E\)](#), [elf_begin\(3E\)](#), [elf_getehdr\(3E\)](#), [elf_kind\(3E\)](#), [elf_rawfile\(3E\)](#).

elf_getphdr(3E)**NAME**

`elf_getphdr:`
`elf32_getphdr, elf32_newphdr` - retrieveclass-dependent program header table

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
Elf32_Phdr *elf32_getphdr(Elf *elf);
□
□
Elf32_Phdr *elf32_newphdr(Elf *elf, size_t count);
```

DESCRIPTION

For a 32-bit class file, `elf32_getphdr()` returns a pointer to the program execution header table, if one is available for the ELF descriptor *elf*.

`elf32_newphdr()` allocates a new table with *count* entries, regardless of whether one existed previously, and sets the `ELF_F_DIRTY` bit for the table [see [elf_flag\(3E\)](#)]. Specifying a zero *count* deletes an existing table. Note this behavior differs from that of `elf32_newehdr` [see [elf_getehdr\(3E\)](#)], allowing a program to replace or delete the program header table, changing its size if necessary.

If no program header table exists, the file is not a 32-bit class file, an error occurs, or *elf* is null, both functions return a null pointer. Additionally, `elf32_newphdr()` returns a null pointer if *count* is zero.

The table is an array of `Elf32_Phdr` structures, each of which includes the following members.

<code>Elf32_Word</code>	<code>p_type;</code>
<code>Elf32_Off</code>	<code>p_offset;</code>
<code>Elf32_Addr</code>	<code>p_vaddr;</code>
<code>Elf32_Addr</code>	<code>p_paddr;</code>
<code>Elf32_Word</code>	<code>p_filesz;</code>
<code>Elf32_Word</code>	<code>p_memsz;</code>
<code>Elf32_Word</code>	<code>p_flags;</code>
<code>Elf32_Word</code>	<code>p_align;</code>

The ELF header's `e_phnum` member tells how many entries the program header table has [see [elf_getehdr\(3E\)](#)]. A program may inspect this value to determine the size of an existing table; `elf32_newphdr()` automatically sets the member's value to *count*. If the program is building a new file, it is responsible for creating the file's ELF header before creating the program header table.

SEE ALSO

[elf\(3E\)](#), [elf_begin\(3E\)](#), [elf_flag\(3E\)](#), [elf_getehdr\(3E\)](#).

elf_getscn(3E)**NAME**

`elf_getscn`, `elf_ndxscn`, `elf_newscn`, `elf_nextscn` - get section information

DESCRIPTION

```
cc [flag ...] file ... -lelf [library ...]
#include <libelf.h>
Elf_Scn *elf_getscn(Elf *elf, size_t index);
size_t elf_ndxscn(Elf_Scn *scn);
Elf_Scn *elf_newscn(Elf *elf);
Elf_Scn *elf_nextscn(Elf *elf, Elf_Scn *scn);
```

DESCRIPTION

These functions provide indexed and sequential access to the sections associated with the ELF descriptor *elf*. If the program is building a new file, it is responsible for creating the file's ELF header before creating sections; see [elf_getehdr\(3E\)](#).

`elf_getscn()` returns a section descriptor, given an *index* into the file's section header table. Note the first "real" section has index 1. Although a program can get a section descriptor for the section whose *index* is 0 (`SHN_UNDEF`, the undefined section), the section has no data and the section header is "empty" (though present). If the specified section does not exist, an error occurs, or *elf* is null, `elf_getscn()` returns a null pointer.

`elf_newscn()` creates a new section and appends it to the list for *elf*. Because the `SHN_UNDEF` section is required and not "interesting" to applications, the library creates it automatically. Thus the first call to `elf_newscn()` for an ELF descriptor with no existing sections returns a descriptor for section 1. If an error occurs or *elf* is null, `elf_newscn()` returns a null pointer.

After creating a new section descriptor, the program can use `elf_getshdr()` to retrieve the newly created, "clean" section header. The new section descriptor will have no associated data [see [elf_getdata\(3E\)](#)]. When creating a new section in this way, the library updates the `e_shnum` member of the ELF header and sets the `ELF_F_DIRTY` bit for the section [see [elf_flag\(3E\)](#)]. If the program is building a new file, it is responsible for creating the file's ELF header [see [elf_getehdr\(3E\)](#)] before creating new sections.

`elf_nextscn()` takes an existing section descriptor, *scn*, and returns a section descriptor for the next higher section. One may use a null *scn* to obtain a section descriptor for the section whose index is 1 (skipping the section whose index is `SHN_UNDEF`). If no further sections are present or an error occurs, `elf_nextscn()` returns a null pointer.

`elf_ndxscn()` takes an existing section descriptor, *scn*, and returns its section table index. If *scn* is null or an error occurs, `elf_ndxscn()` returns `SHN_UNDEF`.

EXAMPLES

An example of sequential access appears below. Each pass through the loop processes the next section in the file; the loop terminates when all sections have been processed.

```
scn = 0;
while ((scn = elf_nextscn(elf, scn)) != 0)
{
    /* process section */
}
```

SEE ALSO

[elf\(3E\)](#), [elf_begin\(3E\)](#), [elf_flag\(3E\)](#), [elf_getdata\(3E\)](#), [elf_getehdr\(3E\)](#), [elf_getshdr\(3E\)](#).

elf_getshdr(3E)**NAME**

elf_getshdr - retrieve class-dependent section header

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]  
#include <libelf.h>  
Elf32_Shdr *elf32_getshdr(Elf_Scn *scn);
```

DESCRIPTION

For a 32-bit class file, `elf32_getshdr()` returns a pointer to a section header for the section descriptor *scn*. Otherwise, the file is not a 32bit class file, *scn* was null, or an error occurred; `elf32_getshdr()` then returns null.

The header includes the following members.

Elf32_Word	sh_name;
Elf32_Word	sh_type;
Elf32_Word	sh_flags;
Elf32_Addr	sh_addr;
Elf32_Off	sh_offset;
Elf32_Word	sh_size;
Elf32_Word	sh_link;
Elf32_Word	sh_info;
Elf32_Word	sh_addralign;
Elf32_Word	sh_entsize;

If the program is building a new file, it is responsible for creating the file's ELF header before creating sections.

SEE ALSO

[elf\(3E\)](#), [elf_flag\(3E\)](#), [elf_getscn\(3E\)](#), [elf_strptr\(3E\)](#).

elf_hash(3E)**NAME**

elf_hash - compute hash value

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]  
#include <libelf.h>  
unsigned long elf_hash(const char *name);
```

DESCRIPTION

elf_hash() computes a hash value, given a null terminated string, *name*. The returned hash value, *h*, can be used as a bucket index, typically after computing $h \bmod x$ to ensure appropriate bounds.

Hash tables may be built on one machine and used on another because elf_hash() uses unsigned arithmetic to avoid possible differences in various machines' signed arithmetic. Although *name* is shown as char* above, elf_hash() treats it as unsigned char* to avoid sign extension differences. Using char* eliminates type conflicts with expressions such as elf_hash("name").

ELF files' symbol hash tables are computed using this function [see [elf_getdata\(3E\)](#) and [elf_xlate\(3E\)](#)]. The hash value returned is guaranteed not to be the bit pattern of all ones (~0UL).

SEE ALSO

[elf\(3E\)](#), [elf_getdata\(3E\)](#), [elf_xlate\(3E\)](#).

NAME

elf_kind - determine file type

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
Elf_Kind elf_kind(Elf *elf);
```

DESCRIPTION

This function returns a value identifying the kind of file associated with an ELF descriptor, *elf*. Currently defined values appear below.

ELF_K_AR

The file is an archive [see [ar\(4\)](#)]. An ELF descriptor may also be associated with an archive member, not the archive itself, and then `elf_kind()` identifies the member's type.

ELF_K_COFF

The file is a COFF object file. [elf_begin\(3E\)](#) describes the library's handling for COFF files.

ELF_K_ELF

The file is an ELF file. The program may use `elf_getident()` to determine the class. Other functions, such as `elf_getehdr()`, are available to retrieve other file information.

ELF_K_NONE

This indicates a kind of file unknown to the library.

Other values are reserved, to be assigned as needed to new kinds of files. *elf* should be a value previously returned by `elf_begin()`. A null pointer is allowed, to simplify error handling, and causes `elf_kind()` to return `ELF_K_NONE`.

SEE ALSO

[elf\(3E\)](#), [elf_begin\(3E\)](#), [elf_getehdr\(3E\)](#), [elf_getident\(3E\)](#), [ar\(4\)](#).

NAME

elf_next - access archive member sequentially

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]  
#include <libelf.h>  
Elf_Cmd elf_next(Elf *elf);
```

DESCRIPTION

elf_next(), elf_rand(), and elf_begin() manipulate simple object files and archives. *elf* is an ELF descriptor previously returned from elf_begin().

elf_next() provides sequential access to the next archive member. That is, having an ELF descriptor, *elf*, associated with an archive member, elf_next() prepares the containing archive to access the following member when the program calls elf_begin(). After successfully positioning an archive for the next member, elf_next() returns the value ELF_C_READ. Otherwise, the open file was not an archive, *elf* was null, or an error occurred, and the return value is ELF_C_NULL. In either case, the return value may be passed as an argument to elf_begin(), specifying the appropriate action.

SEE ALSO

elf(3E), elf_begin(3E), elf_getarsym(3E), elf_rand(3E), ar(4).

NAME

elf_rand - access random archive member

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]  
#include <libelf.h>  
size_t elf_rand(Elf *elf, size_t offset);
```

DESCRIPTION

elf_rand(), elf_next(), and elf_begin() manipulate simple object files and archives. *elf* is an ELF descriptor previously returned from elf_begin().

elf_rand() provides random archive processing, preparing *elf* to access an arbitrary archive member. *elf* must be a descriptor for the archive itself, not a member within the archive. *offset* gives the byte offset from the beginning of the archive to the archive header of the desired member. See [elf_getarsym\(3E\)](#) for more information about archive member offsets. When elf_rand() works, it returns *offset*. Otherwise it returns 0, because an error occurred, *elf* was null, or the file was not an archive (no archive member can have a zero offset). A program may mix random and sequential archive processing.

EXAMPLE

An archive starts with a "magic string" that has SARMAG bytes; the initial archive member follows immediately. An application could thus provide the following function to rewind an archive (the function returns -1 for errors and 0 otherwise).

```
#include <ar.h>  
#include <libelf.h>  
  
int  
rewindelf(Elf *elf)  
{  
    if (elf_rand(elf, (size_t)SARMAG) == SARMAG)  
        return 0;  
    return -1;  
}
```

SEE ALSO

[elf\(3E\)](#), [elf_begin\(3E\)](#), [elf_getarsym\(3E\)](#), [elf_next\(3E\)](#), [ar\(4\)](#).

elf_rawfile(3E)**NAME**

`elf_rawfile` - retrieve uninterpreted file contents

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]  
#include <libelf.h>  
char *elf_rawfile(Elf *elf, size_t *ptr);
```

DESCRIPTION

`elf_rawfile()` returns a pointer to an uninterpreted byte image of the file. This function should be used only to retrieve a file being read. For example, a program might use `elf_rawfile()` to retrieve the bytes for an archive member.

A program may not close or disable [see [elf_cntl\(3E\)](#)] the file descriptor associated with *elf* before the initial call to `elf_rawfile()`, because `elf_rawfile()` might have to read the data from the file if it does not already have the original bytes in memory. Generally, this function is more efficient for unknown file types than for object files. The library implicitly translates object files in memory, while it leaves unknown files unmodified. Thus asking for the uninterpreted image of an object file may create a duplicate copy in memory.

`elf_rawdata` [see [elf_getdata\(3E\)](#)] is a related function, providing access to sections within a file.

If *ptr* is non-null, the library also stores the file's size, in bytes, in the location to which *ptr* points. If no data is present, *elf* is null, or an error occurs, the return value is a null pointer, with zero optionally stored through *ptr*.

NOTES

A program that uses `elf_rawfile()` and that also interprets the same file as an object file potentially has two copies of the bytes in memory. If such a program requests the raw image first, before it asks for translated information (through such functions as `elf_kind()`, `elf_getdata()`, and so on), the library "freezes" its original memory copy for the raw image. It then uses this frozen copy as the source for creating translated objects, without reading the file again. Consequently, the application should view the raw file image returned by `elf_rawfile()` as a read-only buffer, unless it wants to alter its own view of data subsequently translated. In any case, the application may alter the translated objects without changing bytes visible in the raw image. Multiple calls to `elf_rawfile()` with the same ELF descriptor return the same value; the library does not create duplicate copies of the file.

SEE ALSO

[elf\(3E\)](#), [elf_begin\(3E\)](#), [elf_cntl\(3E\)](#), [elf_getdata\(3E\)](#), [elf_getehdr\(3E\)](#),
[elf_getident\(3E\)](#), [elf_kind\(3E\)](#).

elf_strptr(3E)**NAME**

elf_strptr - make a string pointer

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

□
□
□
char *elf_strptr(Elf *elf, size_t section, size_t offset);□
```

DESCRIPTION

This function converts a string section *offset* to a string pointer. *elf* identifies the file in which the string section resides, and *section* gives the section table index for the strings. `elf_strptr()` normally returns a pointer to a string, but it returns a null pointer when *elf* is null, *section* is invalid or is not a section of type `SHT_STRTAB`, the section data cannot be obtained, *offset* is invalid, or an error occurs.

EXAMPLE

A prototype for retrieving section names appears below. The file header specifies the section name string table in the `e_shstrndx` member. The following code loops through the sections, printing their names.

```
if ((ehdr = elf32_getehdr(elf)) == 0)
{
    /* handle the error */
    return;
}
ndx = ehdr->e_shstrndx;
scn = 0;
while ((scn = elf_nextscn(elf, scn)) != 0)
{
    char    *name = 0;
    if ((shdr = elf32_getshdr(scn)) != 0)
        name = elf_strptr(elf, ndx, (size_t)shdr->sh_name);
    printf("%s'\n", name? name: "(null)");
}
```

NOTES

A program may call `elf_getdata()` to retrieve an entire string table section. For some applications, that would be both more efficient and more convenient than using `elf_strptr()`.

SEE ALSO

[elf\(3E\)](#), [elf_getdata\(3E\)](#), [elf_getshdr\(3E\)](#), [elf_xlate\(3E\)](#).

NAME

elf_update - update an ELF descriptor

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
off_t elf_update(Elf *elf, Elf_Cmd cmd);
```

DESCRIPTION

elf_update() causes the library to examine the information associated with an ELF descriptor, *elf*, and to recalculate the structural data needed to generate the file's image.
cmd may have the following values.

- ELF_C_NULL
This value tells elf_update() to recalculate various values, updating only the ELF descriptor's memory structures. Any modified structures are flagged with the ELF_F_DIRTY bit. A program thus can update the structural information and then reexamine them without changing the file associated with the ELF descriptor. Because this does not change the file, the ELF descriptor may allow reading, writing, or both reading and writing [see elf_begin(3E)].
- ELF_C_WRITE
If *cmd* has this value, elf_update() duplicates its ELF_C_NULL actions and also writes any "dirty" information associated with the ELF descriptor to the file. That is, when a program has used elf_getdata() or the elf_flag() facilities to supply new (or update existing) information for an ELF descriptor, those data will be examined, coordinated, translated if necessary [see elf_xlate(3E)], and written to the file. When portions of the file are written, any ELF_F_DIRTY bits are reset, indicating those items no longer need to be written to the file [see elf_flag(3E)]. The sections' data items are written in the order of their section header entries, and the section header table is written to the end of the file.

When the ELF descriptor was created with elf_begin(), it must have allowed writing the file. That is, the elf_begin() command must have been either ELF_C_RDWR or ELF_C_WRITE.

If elf_update() succeeds, it returns the total size of the file image (not the memory image), in bytes. Otherwise an error occurred, and the function returns -1.
When updating the internal structures, elf_update() sets some members itself. Members listed below are the application's responsibility and retain the values given by the program.

ELF Header	Member	Notes
	e_ident[EI_DATA]	Library controls other e_ident values
	e_type	
	e_machine	
	e_version	
	e_entry	
	e_phoff	Only when ELF_F_LAYOUT asserted
	e_shoff	Only when ELF_F_LAYOUT asserted
	e_flags	
	e_shstrndx	

□

Program Header	Member	Notes
	p_type	The application controls all program header entries
	p_offset	
	p_vaddr	
	p_paddr	
	p_filesz	
	p_memsz	
	p_flags	
	p_align	

□

Section Header	Member	Notes
	sh_name	Only when <code>ELF_F_LAYOUT</code> asserted Only when <code>ELF_F_LAYOUT</code> asserted Only when <code>ELF_F_LAYOUT</code> asserted
	sh_type	
	sh_flags	
	sh_addr	
	sh_offset	
	sh_size	
	sh_link	
	sh_info	
	sh_addralign	
	sh_entsize	
	e	

□

Data Descriptor	Member	Notes
	d_buf	Only when <code>ELF_F_LAYOUT</code> asserted
	d_type	
	d_size	
	d_off	
	d_align	
	d_version	

Note the program is responsible for two particularly important members (among others) in the ELF header. The `e_version` member controls the version of data structures written to the file. If the version is `EV_NONE`, the library uses its own internal version. The `e_ident[EI_DATA]` entry controls the data encoding used in the file. As a special case, the value may be `ELFDATANONE` to request the native data encoding for the host machine. An error occurs in this case if the native encoding doesn't match a file encoding known by the library.

Further note that the program is responsible for the `sh_entsize` section header member. Although the library sets it for sections with known types, it cannot reliably know the correct value for all sections.

Consequently, the library relies on the program to provide the values for unknown section type. If the entry size is unknown or not applicable, the value should be set to zero.

When deciding how to build the output file, `elf_update()` obeys the alignments of individual data buffers to create output sections. A section's most strictly aligned data buffer controls the section's alignment. The library also inserts padding between buffers, as necessary, to ensure the proper alignment of each buffer.

NOTES

As mentioned above, the `ELF_C_WRITE` command translates data as necessary, before writing them to the file. This translation is *not* always transparent to the application program. If a program has obtained pointers to data associated with a file [for example, see `elf_getehdr(3E)` and `elf_getdata(3E)`], the program should reestablish the pointers after calling `elf_update()`.

As `elf_begin(3E)` describes, a program may "update" a COFF file to make the image consistent for ELF. The `ELF_C_NULL` command updates only the memory image; one can use the `ELF_C_WRITE` command to modify the file as well. Absolute executable files (`a.out` files) require special alignment, which cannot normally be preserved between COFF and ELF. Consequently, one may not update an executable COFF file with the `ELF_C_WRITE` command (though `ELF_C_NULL` is allowed).

SEE ALSO

`elf(3E)`, `elf_begin(3E)`, `elf_flag(3E)`, `elf_fsize(3E)`, `elf_getdata(3E)`,
`elf_getehdr(3E)`, `elf_getshdr(3E)`, `elf_xlate(3E)`.

elf_version(3E)**NAME**

elf_version - coordinate ELF versions

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
unsigned elf_version(unsigned ver);
```

DESCRIPTION

As [elf\(3E\)](#) explains, the program, the library, and an object file have independent notions of the "latest" ELF version. `elf_version()` lets a program determine the ELF library's *internal version*. It further lets the program specify what memory types it uses by giving its own *working version*, *ver*, to the library. Every program that uses the ELF library must coordinate versions as described below.

The header file `libelf.h` supplies the version to the program with the macro `EV_CURRENT`. If the library's internal version (the highest version known to the library) is lower than that known by the program itself, the library may lack semantic knowledge assumed by the program. Accordingly, `elf_version()` will not accept a working version unknown to the library.

Passing *ver* equal to `EV_NONE` causes `elf_version()` to return the library's internal version, without altering the working version. If *ver* is a version known to the library, `elf_version()` returns the previous (or initial) working version number. Otherwise, the working version remains unchanged and `elf_version()` returns `EV_NONE`.

EXAMPLE

The following excerpt from an application program protects itself from using an older library.

```
if (elf_version(EV_CURRENT) == EV_NONE)
{
    /* library out of date */
    /* recover from error */
}
```

NOTES

The working version should be the same for all operations on a particular elf descriptor. Changing the version between operations on a descriptor will probably not give the expected results.

SEE ALSO

[elf\(3E\)](#), [elf_begin\(3E\)](#), [elf_xlate\(3E\)](#).

NAME

elf_xlate:
elf32_xlatetof, elf32_xlatetom - class-dependent data translation

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]  
#include <libelf.h>  
  
Elf_Data *elf32_xlatetof(Elf_Data *dst, const Elf_Data *src,   
    unsigned encode);  
  
Elf_Data *elf32_xlatetom(Elf_Data *dst, const Elf_Data *src,   
    unsigned encode);
```

DESCRIPTION

elf32_xlatetom() translates various data structures from their 32-bit class file representations to their memory representations; elf32_xlatetof() provides the inverse. This conversion is particularly important for cross development environments. *src* is a pointer to the source buffer that holds the original data; *dst* is a pointer to a destination buffer that will hold the translated copy. *encode* gives the byte encoding in which the file objects are (to be) represented and must have one of the encoding values defined for the ELF header's `e_ident[EI_DATA]` entry [see [elf_getident\(3E\)](#)]. If the data can be translated, the functions return *dst*. Otherwise, they return null because an error occurred, such as incompatible types, destination buffer overflow, etc.

[elf_getdata\(3E\)](#) describes the `Elf_Data` descriptor, which the translation routines use as follows.

`d_buf` Both the source and destination must have valid buffer pointers.

`d_type`
This member's value specifies the type of the data to which `d_buf` points and the type of data to be created in the destination. The program supplies a `d_type` value in the source; the library sets the destination's `d_type` to the same value. These values are summarized below.

`d_size`
This member holds the total size, in bytes, of the memory occupied by the source data and the size allocated for the destination data. If the destination buffer is not large enough, the routines do not change its original contents. The translation routines reset the destination's `d_size` member to the actual size required, after the translation occurs. The source and destination sizes may differ.

`d_version`
This member holds version number of the objects (desired) in the buffer. The source and destination versions are independent.

Translation routines allow the source and destination buffers to coincide. That is, `dst->d_buf` may equal `src->d_buf`. Other cases where the source and destination buffers overlap give undefined behavior.

Elf_Type	32-Bit Memory Type
ELF_T_ADDR	Elf32_Addr
ELF_T_BYTE	unsigned char
ELF_T_DYN	Elf32_Dyn
ELF_T_EHDR	Elf32_Ehdr
ELF_T_HALF	Elf32_Half
ELF_T_OFF	Elf32_Off
ELF_T_PHDR	Elf32_Phdr

ELF_T_REL	Elf32_Rel
ELF_T_RELA	Elf32_Rela
ELF_T_SHDR	Elf32_Shdr
ELF_T_SWORD	Elf32_Sword
ELF_T_SYM	Elf32_Sym
ELF_T_WORD	Elf32_Word

Translating buffers of type `ELF_T_BYTE` does not change the byte order.

SEE ALSO

`elf(3E)`, `elf_fsize(3E)`, `elf_getdata(3E)`, `elf_getident(3E)`.

nlist(3E)**NAME**

nlist - get entries from name list

SYNOPSIS

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <nlist.h>
int nlist (const char *filename, struct nlist *nl);
```

DESCRIPTION

nlist() examines the name list in the executable file whose name is pointed to by *filename*, and selectively extracts a list of values and puts them in the array of **nlist()** structures pointed to by *nl*. The name list *nl* consists of an array of structures containing names of variables, types, and values. The list is terminated with a null name, i.e. by a structure containing a null string in the name position. Each variable name is looked up in the name list of the file. If the name is found, the type, value, storage class, and section number of the name are inserted in the other fields. The type field may be set to 0 if the file was not compiled with the `-g` option to `cc(1)`. **nlist()** will always return the information for an external symbol of a given name if the name exists in the file. If an external symbol does not exist, and there is more than one symbol with the specified name in the file (such as `static` symbols defined in separate files), the values returned will be for the last occurrence of that name in the file. If the name is not found, all fields in the structure except `n_name` are set to 0.

This function is useful for examining the system name list kept in the file `/stand/unix`. In this way programs can obtain system addresses that are up to date.

RESULT

All value entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

nlist() returns 0 on success, -1 on error.

SEE ALSO

[a.out\(4\)](#).

General functions from the library libgen(3G)**bgets(3G)****NAME**

bgets - read stream up to next delimiter

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>
char *bgets(char *buffer, size_t *count, FILE *stream,
            const char *breakstring);
```

DESCRIPTION

bgets() reads characters from *stream* into *buffer* until either *count* is exhausted or one of the characters in *breakstring* is encountered in the stream. The read data is terminated with a null byte (`\0`) and a pointer to the trailing null is returned. If a *breakstring* character is encountered, the last non-null is the delimiter character that

terminated the scan.

Note that, except for the fact that the returned value points to the end of the read string rather than to the beginning, the following calls are identical:

```
bgets (buffer, sizeof buffer, stream, "\n")
fgets (buffer, sizeof buffer, stream);
```

There is always enough room reserved in the buffer for the trailing null.

If *breakstring* is a null pointer, the value of *breakstring* from the previous call is used. If *breakstring* is null at the first call, no characters will be used to delimit the string.

EXAMPLES

```
#include <libgen.h>
char buffer[8];
/* read in first user name from /etc/passwd */
fp = fopen("/etc/passwd", "r");
bgets(buffer, 8, fp, ":");
```

RESULT

NULL is returned on error or end-of-file. If characters have been read but not yet returned, the error condition is not reported until the next call.

SEE ALSO

[gets\(3S\)](#).

NAME

bufsplit - split buffer into fields

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>
size_t bsplit(char *buf, size_t n, char **a);
```

DESCRIPTION

bufsplit() examines the buffer, *buf*, and assigns values to the pointer array, *a*, so that the pointers point to the first *n* fields in *buf* that are delimited by tabs or new-lines.

To change the characters used to separate fields, call bsplit() with *buf* pointing to the string of characters, and *n* and *a* set to zero. For example, to use :, ., and , as separators along with tab and new-line, enter the following:

```
bufsplit (":.,\t\n", 0, (char**)0);
```

EXAMPLES

```
/*
 * set a[0] = "This", a[1] = "is", a[2] = "a",
 * a[3] = "test"
 */
bufsplit("This\tis\ta\ttest\n", 4, a);
```

NOTES

bufsplit() changes the delimiters to null bytes in *buf*.

RESULT

The number of fields assigned in the array *a* is returned. If *buf* is NULL, the return value is zero and the array is unchanged. Otherwise the value is at least 1. The remainder of the elements in the array are assigned the address of the null byte at the end of the buffer.

NAME

`copylist` - copy a file into memory

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>
char *copylist(const char *filenm, off_t *szptr);
```

DESCRIPTION

`copylist()` copies a list of items from a file into freshly allocated memory, replacing new-lines with null characters. It expects two arguments: a pointer *filenm* to the name of the file to be copied, and a pointer *szptr* to a variable where the size of the file will be stored.

Upon success, `copylist()` returns a pointer to the memory allocated. Otherwise it returns `NULL` if it has trouble finding the file, calling `malloc()`, or opening the file.

EXAMPLES

```
/* read "file" into buf */
off_t size;
char *buf;
buf = copylist("file", &size);
for (i = 0; i < size; i++)
    if (buf[i])
        putchar(buf[i]);
    else
        putchar('\n')
```

SEE ALSO

[malloc\(3C\)](#).

NAME

dirname - output parent directory name

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#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 `"."`.

This interface need not be reentrant.

`dirname()` and `basename()` together yield a complete path name. `dirname(path)` is the directory where `basename(path)` is found.

RETURN VALUE

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.

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

ERRORS

No errors are defined.

EXAMPLES

A simple filename and the strings `"."` and `".."` all have `"."` as their return value.

Input string	Output string
<code>"/usr/lib"</code>	<code>"/usr"</code>
<code>"/usr/"</code>	<code>"/"</code>
<code>"usr"</code>	<code>"."</code>
<code>"/"</code>	<code>"/"</code>
<code>"."</code>	<code>"."</code>
<code>".."</code>	<code>"."</code>

The following code reads a path name, changes directory to the appropriate directory [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);
```

SEE ALSO

```
basename(1), chdir(2), basename(3C), libgen(5).
```

NAME

`gmatch` - shell global pattern matching

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]  
#include <libgen.h>  
int gmatch(const char *str, const char *pattern);
```

DESCRIPTION

`gmatch()` checks whether the null-terminated string *str* matches the null-terminated pattern string *pattern*. `gmatch()` returns non-zero if the pattern matches the string, zero if the pattern doesn't. A backslash `\` is used as an escape character in pattern strings.

EXAMPLE

```
char *s;  
gmatch (s, "[a\-" );  
  
gmatch() returns non-zero for all strings with a or – as their last character.
```

SEE ALSO

[sh\(1\)](#).

NAME

`isencrypt` - determine whether a character buffer is encrypted

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]  
#include <libgen.h>  
int isencrypt(const char *fbuf, size_t ninbuf);
```

DESCRIPTION

`isencrypt()` can be used to determine whether a buffer of characters is encrypted.

`isencrypt()` assumes that the file is not encrypted if all the characters in the first block are ASCII characters. If there are non-ASCII characters in the first *ninbuf* characters, and if the `setlocale LC_CTYPE` category is set to `C` or `ascii`, the buffer is not encrypted.

If the `LC_CTYPE` category is set to a value other than `C` or `ascii`, then `isencrypt()` uses a heuristic to determine if the buffer is encrypted. If the buffer has at least 64 characters, a chi-square test is used to determine if the bytes in the buffer have a uniform distribution; and `isencrypt()` assumes the buffer is encrypted if it does. If the buffer has less than 64 characters, a check is made for null characters and a terminating new-line to determine whether the buffer is encrypted.

RESULT

If the buffer is encrypted, 1 is returned; otherwise 0 is returned.

SEE ALSO

`setlocale(3C)`.

NAME

mkdirp, rmdirp - create/remove directories in a path

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>
int mkdirp(const char *path, mode_t mode);
int rmdirp(char *path1, char *path1);
```

DESCRIPTION

mkdirp() creates all the missing directories in the given *path* with the given *mode* (see [chmod\(2\)](#) for the values of *mode*).

rmdirp() removes directories in path *path1*. This removal starts at the end of the path and moves back toward the root as far as possible. If an error occurs, the remaining path is stored in *path2*. rmdirp() returns a 0 only if it is able to remove every directory in the path.

EXAMPLES

```
/* create scratch directories */
if(mkdirp("/tmp/sub1/sub2/sub3", 0755) == -1) {
    fprintf(stderr, "cannot create directory");
    exit(1);
}
chdir("/tmp/sub1/sub2/sub3");
/* cleanup */
chdir("/tmp");
rmdirp("sub1/sub2/sub3");
```

RESULT

If a needed directory cannot be created, mkdirp() returns -1 and sets `errno` to one of the `mkdir()` error numbers. If all the directories are created, or existed to begin with, it returns zero.

NOTES

mkdirp() uses `malloc()` to allocate temporary space for the string.

rmdirp() returns -2 if a "." or ".." is in the path and -3 if an attempt is made to remove the current directory. If an error occurs other than one of the above, -1 is returned.

SEE ALSO

[mkdir\(2\)](#), [rmdir\(2\)](#).

NAME

`p2open`, `p2close` - open/close pipe for a command

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>
int p2open(const char *cmd, FILE *fp[2]);
int p2close(FILE *fp[2]); □
```

DESCRIPTION

`p2open()` creates and starts a shell procedure to execute the command line pointed to by *cmd*. On return, `fp[0]` points to a `FILE` pointer to write to the command's standard input and `fp[1]` points to a `FILE` pointer to read from the command's standard output. In this way the program has control over the input and output of the command.

The function returns 0 if successful; otherwise it returns -1.

`p2close()` is used to close the file pointers that `p2open()` opened. It waits for the process to terminate and returns the process status. It returns 0 if successful; otherwise it returns -1.

EXAMPLES

```
#include <stdio.h>
#include <libgen.h>
main(argc,argv)
int argc;
char **argv;
{
    FILE *fp[2];
    pid_t pid;
    char buf[16];
    pid=p2open("/usr/bin/cat", fp);
    if ( pid == 0 ) {
        fprintf(stderr, "p2open failed\n");
        exit(1);
    }
    write(fileno(fp[0]), "This is a test\n", 16);
    if(read(fileno(fp[1]), buf, 16) <=0)
        fprintf(stderr, "p2open failed\n");
    else
        write(1, buf, 16);
    (void)p2close(fp);
}
```

RESULT

Often there are too few file descriptors. `p2close()` returns -1 if the two file pointers are not from the same `p2open()`.

NOTES

Buffered writes on `fp[0]` can make it appear that the command is not being executed. Problems can also occur when commands which are linked to a pipe use an unbuffered output. To avoid these problems, use `fflush()` calls or an unbuffered `fp[0]` [see also [fclose\(3S\)](#)].

SEE ALSO

`fclose(3S)`, `popen(3S)`, `setbuf(3S)`.

NAME

pathfind - search for named file in named directories

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

char *pathfind(const char *path, const char *name,
               const char *mode);
```

DESCRIPTION

pathfind() searches the directories named in *path* for the file *name*. The directories named in *path* are separated by semicolons. *mode* is a string of option with the following meanings:

Letter	Meaning
r	readable
w	writable
x	executable
f	normal file
b	block special
c	character special
d	directory
p	FIFO (pipe)
u	set user ID bit
g	set group ID bit
k	sticky bit
s	size nonzero

Options read, write, and execute are checked relative to the real user ID and group ID of the current process.

If the file *name*, with all the characteristics specified by *mode*, is found in any of the directories specified by *path*, then pathfind() returns a pointer to a string containing the member of *path*, followed by a slash character /, followed by *name*.

If *name* begins with a slash, it is treated as an absolute path name, and *path* is ignored.

An empty *path* member is treated as the current directory. ./ is not prepended at the occurrence of the first match; rather, the unadorned *name* is returned.

EXAMPLES

To find the `ls` command using the `PATH` environment variable:

```
pathfind (getenv ("PATH"), "ls", "rx")
```

RESULT

If no match is found, pathfind() returns a null pointer, ((char *)0).

NOTES

The string pointed to by the returned pointer is stored in a static area that is reused on subsequent calls to pathfind().

SEE ALSO

`sh(1)`, `test(1)`, `access(2)`, `mknod(2)`, `stat(2)`, `getenv(3C)`.

NAME

regcmp, regex - compile and execute regular expression

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

char *regcmp(const char *string1, ... /*, (char *)0 */);
char *regex(const char *re, const char *subject, ... );
extern char *__loc1;
```

DESCRIPTION

regcmp() compiles a regular expression consisting of the concatenated arguments, and returns a pointer to the compiled form. malloc(3C) is used to create memory 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.

regex() searches for a compiled pattern against the subject string. Additional arguments are passed in order to receive values back. regex() 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. regcmp() and regex() 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:

[] * . ^

These symbols retain their meaning in ed(1).

- \$ Matches the end of the string; the character sequence \n within a string, matches a newline.
- Within brackets the minus means *through*. For example, [az] 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 (i.e., {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. regex() makes its assignments unconditionally.
- (...) Parentheses are used for grouping. An operator, e.g., *, +, {}, 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 characters. They must, therefore, be escaped with a backslash \ to be used as themselves.

EXAMPLES

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 character 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);
```

ERRORS

The `regcmp()` function may fail if:

`ENOMEM`

Insufficient storage space was available.

NOTES

The user program may be unable to provide any more memory if `regcmp()` is called iteratively without freeing the vectors no longer required.

If you use one of these functions, you must link the `libgen` library at compilation (`cc -lgen`).

SEE ALSO

[ed\(1\)](#), [regcmp\(1\)](#), [malloc\(3C\)](#), [regcomp\(3C\)](#), [regex\(3C\)](#), [regexpr\(3G\)](#), [expressions\(5\)](#), [libgen\(5\)](#), [re_comp\(5\)](#), [regex\(5\)](#), [regexp\(5\)](#).

NAME

regexpr:
 compile, step, advance, loc1, loc2, locs, nbra, regerrno, reglength, braslist,
 braelist - regular expression compile and match routines

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <regexpr.h>

char *compile(const char *instring, char *expbuf, char *endbuf);
int step(const char *string, char *expbuf);
int advance(const char *string, char *expbuf);
extern char *loc1, *loc2, *locs;
extern int nbra, regerrno, reglength;
extern char *braslist[ ], *braelist[ ];
```

DESCRIPTION

These routines are used to compile regular expressions and match the compiled expressions against lines. The regular expressions compiled are in the form used by `ed`.

`compile()` is used as follows:

The parameter *instring* is a null-terminated string representing the regular expression.

The parameter *expbuf* points to the memory location where the compiled regular expression is to be placed. If *expbuf* is `NULL`, `compile()` uses `malloc()` to allocate the space for the compiled regular expression. If an error occurs, this space is freed. It is the user's responsibility to free unneeded space after the compiled regular expression is no longer needed.

The parameter *endbuf* points to a memory location immediately beyond the highest address where the compiled regular expression may be placed. This argument is ignored if *expbuf* is `NULL`. If the compiled expression cannot fit in (*endbuf* - *expbuf*) bytes, `compile()` returns `NULL` and `regerrno` (see below) is set to 50.

If `compile()` succeeds, it returns a non-`NULL` pointer whose value depends on *expbuf*. If *expbuf* is non-`NULL`, `compile()` returns a pointer to the byte after the last byte in the compiled regular expression. The length of the compiled regular expression is stored in `reglength`. Otherwise, `compile()` returns a pointer to the memory space allocated by `malloc()`.

If an error is detected when compiling the regular expression, a `NULL` pointer is returned from `compile()` and `regerrno` is set to one of the non-zero error numbers indicated below:

ERROR

	MEANING
11	Range endpoint too large
16	Bad number
25	<code>\digit</code> out of range
36	Illegal or missing delimiter
41	No remembered search string
42	<code>\(\)</code> imbalance
43	Too many <code>\(</code>
44	More than 2 numbers given in <code>\{ \}</code>
45	<code>}</code> expected after <code>\</code>

- 46 First number exceeds second in `\{ \}`
- 49 `[]` imbalance
- 50 Regular expression overflow

`step()` is used as follows: The first parameter to `step()` is a pointer to a string of characters to be checked for a match. This string should be null-terminated.

The parameter *expbuf* is the compiled regular expression obtained by a call of the function `compile()`.

The function `step()` returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to `step()`. The variable set in `step()` is `loc1`. `loc1` is a pointer to the first character that matched the regular expression. The variable `loc2` points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, `loc1` points to the first character of *string* and `loc2` points to the null at the end of *string*.

The purpose of `step()` is to step through the *string* argument until a match is found or until the end of *string* is reached. If the regular expression begins with `^`, `step()` tries to match the regular expression at the beginning of the string only.

The function `advance()` has the same arguments and side effects as `step()`, but it always restricts matches to the beginning of the string.

If one is looking for successive matches in the same string of characters, `locs` should be set equal to `loc2`, and `step()` should be called with *string* equal to `loc2`. `locs` is used by commands like `ed` and `sed` so that global substitutions like `s/y*/g` do not loop forever, and is `NULL` by default.

The external variable `nbra` is used to determine the number of subexpressions in the compiled regular expression. `braslist` and `braelist` are arrays of character pointers that point to the start and end of the `nbra` subexpressions in the matched string. For example, after calling `step()` or `advance()` with *string* `sabcdefg` and regular expression `\(abcdef\)`, `braslist[0]` will point at `a` and `braelist[0]` will point at `g`. These arrays are used by commands like `ed` and `sed` for substitute replacement patterns that contain the `\n` notation for subexpressions.

Note that it is not necessary to use the external variables `regerrno`, `nbra`, `loc1`, `loc2`, `locs`, `braelist`, and `braslist` if one is only checking whether or not a string matches a regular expression.

EXAMPLES

The following is similar to the regular expression code from `grep`:

```
#include <regexpr.h>

. . .
if(compile(*argv, (char *)0, (char *)0) == (char *)0)
    regerr(regerrno);

. . .
if (step(linebuf, expbuf))
    succeed();
```

NOTES

If you use one of these functions, you must link the `libgen` library at compilation (`cc -lgen`).

SEE ALSO

`ed(1)`, `grep(1)`, `regcmp(1)`, `sed(1)`, `regcomp(3C)`, `regex(3C)`, `regcmp(3G)`, `expressions(5)`, `regex(5)`, `regexp(5)`.

NAME

`str:`
`strfind`, `strrspn`, `strtrns` - amend strings

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>
int strfind(const char *as1, const char *as2);
char *strrspn(const char *string, const char *tc);
char * strtrns(const char *str, const char *old, const char *new,
               □□□□□□□□□□□□□□ char *result);□
```

DESCRIPTION

`strfind()` returns the offset of the second string, *as2*, if it is a substring of string *as1*.

`strrspn()` returns a pointer to the first character in the string to be trimmed (all characters from the first character to the end of *string* are in *tc*).

`strtrns()` transforms *str* and copies it into *result*. Any character that appears in *old* is replaced with the character in the same position in *new*. *new* is returned as the result.

EXAMPLES

```
/* find pointer to substring "hello" in as1 */
i = strfind(as1, "hello");
/* trim junk from end of string */
s2 = strrspn(s1, ".*#$%");
*s2 = '\0'
/* transform lower case to upper case */
a1[] = "abcdefghijklmnopqrstuvwxyz";
a2[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
s2 = strtrns(s1, a1, a2, s2);
```

RESULT

If the second string is not a substring of the first string `strfind()` returns `-1`.

SEE ALSO

[string\(3C\)](#).

strccpy(3G)**NAME**

strccpy, streadd, strcadd, strecpy - copystrings

SYNOPSIS

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>
char *strccpy(char *output, const char *input);
char *strcadd(char *output, const char *input);
\{\
char *strecpy(char *output, const char *input,
const char *exceptions);
char *streadd(char *output, const char *input,
const char *exceptions);
char *strecpy(char *output, const char *input, const char *exceptions);
char *streadd(char *output, const char *input, const char *exceptions);
```

DESCRIPTION

`strccpy()` copies the *input* string, up to a null byte, to the *output* string, compressing the C language escape sequences (for example, `\n`, `\001`) to the equivalent character. A null byte is appended to the output. The *output* argument must point to a memory space big enough to accommodate the result. If it is as big as the memory space pointed to by *input* it is guaranteed to be big enough. `strccpy()` returns the *output* argument.

`strcadd()` is identical to `strccpy()`, except that it returns the pointer to the null byte that terminates the output.

`strecpy()` copies the *input* string, up to a null byte, to the *output* string, expanding non-graphic characters to their equivalent C language escape sequences (for example, `\n`, `\001`). The *output* argument must point to a memory space big enough to accommodate the result; four times the memory space pointed to by *input* is guaranteed to be big enough (each character could become `\` and 3 digits). Characters in the *exceptions* string are not expanded. The *exceptions* argument may be `NULL`, meaning all non-graphic characters are expanded. `strecpy()` returns the *output* argument

`streadd()` is identical to `strecpy()`, except that it returns the pointer to the null byte that terminates the output.

EXAMPLES

```
/* expand all but newline and tab */
strecpy( output, input, "\n");
/* concatenate and compress several strings */
cp = strcadd( output, input1 );
cp = strcadd( cp, input2 );
cp = strcadd( cp, input3 );
```

SEE ALSO

[string\(3C\)](#), [str\(3G\)](#).

Mathematical funtions from the library libm(3M)

NAME

bessel:
 j0, j1, jn, y0, y1, yn - Bessel functions

SYNOPSIS

```
cc [flag ...] file ... -lm [library ...]
#include <math.h>
double j0(double x);
double j1(double x);
double jn(int n, double x);
double y0(double x);
double y1(double x);
double yn(int n, double x);□
```

DESCRIPTION

j0(), j1(), and jn() compute Bessel functions of x of the first kind of orders 0, 1 and n , respectively.
 y0(), y1(), and yn() compute Bessel functions of x of the second kind of orders 0, 1 and n , respectively.
 The value of x must be positive.

An application wishing to check for error situations should set `errno` to 0 before calling the function. If `errno` is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE**Functions j0(), j1() and jn()**

Upon successful completion, j0(), j1() and jn() return the relevant Bessel value of x of the first kind.
 If the x argument is too large in magnitude, 0 is returned and `errno` may be set to `ERANGE`.
 If x is NaN, NaN is returned and `errno` may be set to `EDOM`.
 If the correct result would cause underflow, 0 is returned and `errno` may be set to `ERANGE`.

Functions y0(), y1() and yn()

Upon successful completion, y0(), y1() and yn() will return the relevant Bessel value of x of the second kind.
 If x is NaN, NaN is returned and `errno` may be set to `EDOM`.
 If the x argument to y0(), y1() or yn() is negative, `-HUGE_VAL` or NaN is returned, and `errno` may be set to `EDOM`.
 If x is 0.0, `-HUGE_VAL` is returned and `errno` may be set to `ERANGE` or `EDOM`.
 If the correct result would cause underflow, 0.0 is returned and `errno` may be set to `ERANGE`.
 If the correct result would cause overflow, `-HUGE_VAL` or 0.0 is returned and `errno` may be set to `ERANGE`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Functions j0(), j1() and jn()

The j0(), j1() and jn() functions may fail if:

`EDOM` The value of x is NaN.

`ERANGE`

The value of x was too large in magnitude, or underflow occurred.

Functions y0(), y1() and yn()

The `y0()`, `y1()` and `yn()` functions may fail if:

EDOM The value of x is negative or NaN.

ERANGE

The value of x is too large in magnitude, or x is 0.0, or the correct result would cause overflow or underflow.

No other errors will occur.

NOTES

In the K&R mode, (`cc -kcc`), these error-handling procedures may be changed with the function `matherr()`. In the ANSI mode, (`cc -kansi`), `HUGE_VAL` is returned instead of `HUGE` and no error messages are printed.

SEE ALSO

`isnan(3C)`, `matherr(3M)`, `math(5)`.

NAME

`erf`, `erfc` - error function and complementary error function

SYNOPSIS

```
cc [ flag ... ] file ... -lm [ library ... ]
#include <math.h>
double erf(double x),
□
double erfc(double x);□
```

DESCRIPTION

`erf()` computes the error function of x , defined as:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

`erfc()` which computes $1.0 - \text{erf}(x)$, is provided because of the extreme loss of relative accuracy which results if `erf(x)` is called for large x and the result subtracted from 1.0 (e.g., for $x = 5$, 12 places are lost).

An application wishing to check for error situations should set `errno` to 0 before calling `erf()`. If `errno` is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

Upon successful completion, `erf()` and `erfc()` return the value of the error function and complementary error function, respectively.

If x is NaN, NaN is returned and `errno` may be set to `EDOM`.

If the correct value would cause underflow, 0 is returned and `errno` may be set to `ERANGE`.

ERRORS

The `erf()` and `erfc()` functions may fail if:

`EDOM` The value of x is NaN.

`ERANGE`

The result underflows.

No other errors will occur.

SEE ALSO

`isnan(3C)`, `exp(3M)`, `math(5)`.

NAME

`exp`, `expf`, `expm1`, `cbrt`, `log`, `logf`, `log10`, `log10f`, `log1p`, `pow`, `powf`, `sqrt`, `sqrtf` -
exponential, logarithm, power, square root functions

SYNOPSIS

```
cc [ flag ... ] file ... -lm [ library ... ]
cc -O -Ksd [ flag ... ] file ... -J sfm
[library ...]  
#include <math.h>  
  
double exp(double x);  
float expf(float x);  
double expm1(double x);  
double cbrt(double x);  
double log(double x);  
float logf(float x);  
double log10(double x);  
float log10f(float x);  
double log1p(double x);  
double pow(double x, double y);  
float powf(float x, float y);  
double sqrt(double x);  
float sqrtf(float x);
```

DESCRIPTION

`exp()` computes e^x .

`expf()` is the single-precision version of `exp()`.

`expm1()` computes $e^x - 1.0$.

`cbrt()` computes the cube root of x .

`log()` computes the natural logarithm of x . The value of x must be positive.

`logf()` is the single-precision version of `log()`.

`log10()` computes the base ten logarithm of x . The value of x must be positive.

`log10f()` is the single-precision version of `log10()`.

`log1p()` computes $\log_e(1.0 + x)$. The value of x must be greater than -1.0 .

`pow()` computes x^y . If x is 0, y must be positive. If x is negative, y must be an integer.

`powf()` is the single-precision version of `pow()`.

`sqrt()` computes the non-negative square root of x . The value of x may not be negative.

`sqrtf()` is the single-precision version of `sqrt()`.

An application wishing to check for error situations should set `errno` to 0, then call the function. If `errno` is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

Functions exp() and expf()

Upon successful completion, `exp()` and `expf()` return the exponential value of x .

If the correct value would cause overflow, `exp()` and `expf()` return `HUGE_VAL` and set `errno` to `ERANGE`.

If the correct value would cause underflow, `exp()` and `expf()` return 0 and may set `errno` to `ERANGE`.

If x is NaN, NaN is returned and `errno` may be set to `EDOM`.

If x is NaN, then `expm1()` returns NaN and `errno` may be set to `EDOM`.

Function expm1()

If x is positive infinity, `expm1()` returns positive infinity.

If x is negative infinity, `expm1()` returns -1.0 .

If the value overflows, `expm1()` returns `HUGE_VAL` and may set `errno` to `ERANGE`.

Function cbrt()

On successful completion, `cbrt()` returns the cube root of x . If x is NaN, `cbrt()` returns NaN and `errno` may be set to `EDOM`.

Functions log() and logf()

Upon successful completion, `log()` and `logf()` return the natural logarithm of x .

If x is NaN, NaN is returned and `errno` may be set to `EDOM`.

If x is less than 0, `-HUGE_VAL` or NaN is returned, and `errno` is set to `EDOM`.

If x is 0, `-HUGE_VAL` is returned and `errno` may be set to `ERANGE`.

Functions log10() and log10f()

Upon successful completion, `log10()` and `log10f()` return the base 10 logarithm of x .

If x is NaN, NaN is returned and `errno` may be set to `EDOM`.

If x is less than 0, `-HUGE_VAL` or NaN is returned, and `errno` is set to `EDOM`.

If x is 0, `-HUGE_VAL` is returned and `errno` may be set to `ERANGE`.

Function log1p()

Upon successful completion, `log1p()` returns the natural logarithm of $1.0 + x$.

If x is NaN, `log1p()` returns NaN and may set `errno` to `EDOM`.

If x is less than -1.0 , `log1p()` returns `-HUGE_VAL` or NaN and sets `errno` to `EDOM`.

If x is -1.0 , `log1p()` returns `-HUGE_VAL` and may set `errno` to `ERANGE`.

Functions pow() and powf()

Upon successful completion, `pow()` and `powf()` return the value of x raised to the power y .

If x is 0 and y is 0, 1.0 is returned.

If y is NaN, or y is non-zero and x is NaN, NaN is returned and `errno` may be set to `EDOM`. If y is 0.0 and x is NaN, either 1.0 is returned, or NaN is returned and `errno` may be set to `EDOM`.

If x is 0.0 and y is negative, `-HUGE_VAL` is returned and `errno` may be set to `EDOM` or `ERANGE`.

If the correct value would cause overflow, `±HUGE_VAL` is returned, and `errno` is set to `ERANGE`.

If the correct value would cause underflow, 0 is returned and `errno` may be set to `ERANGE`.

Functions sqrt() and sqrtf()

Upon successful completion, `sqrt()` and `sqrtf()` return the square root of x .

If x is NaN, NaN is returned and `errno` may be set to `EDOM`.

If x is negative, 0.0 or NaN is returned and `errno` is set to `EDOM`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Functions `exp()` and `expf()`

The `exp()` and `expf()` function will fail if:

ERANGE

The result overflows.

The `exp()` or `expf()` function may fail if:

EDOM The value of x is NaN.

ERANGE

The result underflows.

Function `expm1()`

The `expm1()` function may fail if:

EDOM The value of x is NaN.

ERANGE

The result overflows.

Function `cbrt()`

The `cbrt()` function may fail if:

EDOM The value of x is NaN.

Functions `log()` and `logf()`

The `log()` and `logf()` function will fail if:

EDOM The value of x is negative.

The `log()` or `logf()` function may fail if:

EDOM The value of x is NaN.

ERANGE

The value of x is 0.

Functions `log10()` and `log10f()`

The `log10()` and `log10f()` function will fail if:

EDOM The value of x is negative.

The `log10()` or `log10f()` function may fail if:

EDOM The value of x is NaN.

ERANGE

The value of x is 0.

Function `log1p()`

The `log1p()` function will fail if:

EDOM The value of x is less than -1.0 .

The `log1p()` function may fail and set `errno` to:

EDOM The value of x is NaN.

ERANGE

The value of x is -1.0 .

Functions `pow()` and `powf()`

The `pow()` and `powf()` function will fail if:

EDOM The value of x is negative and y is non-integral.

ERANGE

The value to be returned would have caused overflow.

The `pow()` and `powf()` function may fail if:

EDOM The value of x is 0.0 and y is negative, or y is NaN.

ERANGE

The correct value would cause underflow.

Functions `sqrt()` and `sqrtf()`

The `sqrt()` and `sqrtf()` function will fail if:

EDOM The value of x is negative.

The `sqrt()` and `sqrtf()` function may fail if:

EDOM The value of x is NaN.

No other errors will occur.

APPLICATION USAGE

The value of `expm1( $x$ )` may be more accurate than `exp( $x$ )–1.0` for small values of x .

The `expm1()` and `log1p()` functions are useful for financial calculations of $((1+x)^n-1)/x$, namely:

`expm1( $n * \log1p(x)$ )/ $x$`

when x is very small (for example, when calculating small daily interest rates). These functions also simplify writing accurate inverse hyperbolic functions.

NOTES

The functions `sinhf()`, `coshf()`, and `tanhf()` can be used in ANSI mode only.

SEE ALSO

`ilogb(3C)`, `isnan(3C)`, `hypot(3M)`, `matherr(3M)`, `sinh(3M)`, `math(5)`.

floatingpoint(3M)
BSD Compatibility Package**NAME**

floatingpoint - IEEE floating point definitions

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <sys/ieeefp.h>
#include <fp.h>
```

DESCRIPTION

This file defines constants, types, variables, and functions used to implement standard floating point according to ANSI/IEEE Standard 754-1985. The variables and functions are implemented in `libucb.a`. The included file `<sys/ieeefp.h>` defines certain types of interest to the kernel.

IEEE Rounding Modes:

`fp_direction_type`

The type of the IEEE rounding direction mode. The order of enumeration varies according to hardware.

`fp_direction`

The IEEE rounding direction mode currently in force. This is a global variable that is intended to reflect the hardware state, so it should only be written indirectly through a function that also sets the hardware state.

`fp_precision_type`

The type of the IEEE rounding precision mode, which only applies on systems that support extended precision.

`fp_precision`

The IEEE rounding precision mode currently in force. This is a global variable that is intended to reflect the hardware state on systems with extended precision, so it should only be written indirectly.

SIGFPE handling:

`sigfpe_code_type`

The type of a SIGFPE code.

`sigfpe_handler_type`

The type of a user-definable SIGFPE exception handler called to handle a particular SIGFPE code.

`SIGFPE_DEFAULT`

A macro indicating the default SIGFPE exception handling, namely to perform the exception handling specified by calls to `ieee_handler`, if any, and otherwise to dump core using `abort`.

`SIGFPE_IGNORE`

A macro indicating an alternate SIGFPE exception handling, namely to ignore and continue execution.

`SIGFPE_ABORT`

A macro indicating an alternate SIGFPE exception handling, namely to abort with a core dump.

IEEE Exception Handling:

`N_IEEE_EXCEPTION`

The number of distinct IEEE floating-point exceptions.

`fp_exception_type`

The type of the `N_IEEE_EXCEPTION` exceptions. Each exception is given a bit number.

`fp_exception_field_type`

The type intended to hold at least `N_IEEE_EXCEPTION` bits corresponding to the IEEE exceptions numbered by `fp_exception_type`. Thus `fp_inexact` corresponds to the least significant bit and `fp_invalid` to the fifth least significant bit. Note: Some operations may set more than one exception.

`fp_accrued_exceptions`

The IEEE exceptions between the time this global variable was last cleared, and the last time a function was called to update the variable by obtaining the hardware state.

`ieee_handlers`

An array of user-specifiable signal handlers for use by the standard `SIGFPE` handler for IEEE arithmetic-related `SIGFPE` codes. Since IEEE trapping modes correspond to hardware modes, elements of this array should only be modified with a function like `ieee_handler` that performs the appropriate hardware mode update. If no `sigfpe_handler` has been declared for a particular IEEE-related `SIGFPE` code, then the related `ieee_handlers` will be invoked.

IEEE Formats and Classification:

single; extended

Definitions of IEEE formats.

`fp_class_type`

An enumeration of the various classes of IEEE values and symbols.

FILES

`/usr/include/sys/ieeefp.h`

`/usr/include/fp.h`

`/usr/ucblib/libucb.a`

SEE ALSO

`sigfpe(3)`, `abort(3C)`, `strtod(3C)`, `ieee_handler(3M)`.

floor(3M)**NAME**

`floor`, `floorf`, `ceil`, `ceilf`, `fmod`, `fmodf`, `fabs`, `fabsf`, `rint`, `remainder`, `copysign` -
 floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
cc [ flag ... ] file ... -lm [ library ... ]
#include <math.h>
double floor(double x);
float floorf(float x);
double ceil(double x);
float ceilf(float x);
double fmod(double x, double y);
float fmodf(float x, float y);
double fabs(double x);
float fabsf(float x);
double rint(double x);
double remainder(double x, double y);
double copysign(double x, double y);
```

DESCRIPTION

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

`floorf()` is the single-precision version of `floor()`.

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

`ceilf()` is the single-precision version of `ceil()`.

The `fabs()` function computes the absolute value of x , $|x|$.

`fabsf()` is the single-precision version of `fabs()`.

The `fmod()` function returns the floating-point remainder of the division of x by y .

`fmodf()` is the single-precision version of `fmod()`.

The `rint()` function returns the integral value (represented as a double) nearest x in the direction of the current rounding mode. The current rounding mode is implementation-dependent.

If the current rounding mode rounds toward negative infinity, then `rint()` is identical to `floor()`. If the current rounding mode rounds toward positive infinity, then `rint()` is identical to `ceil()`.

The `remainder()` function returns the floating-point remainder $r = x - ny$ when y is non-zero. The value n is the integral value nearest the exact value x/y . When $|n - x/y| = 1/2$, the value n is chosen to be even.

The behaviour of `remainder()` is independent of the rounding mode.

`copysign()` returns x but with the sign of y .

An application wishing to check for error situations should set `errno` to 0, then call the function. If `errno` is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE**Functions `floor()` and `floorf()`**

Upon successful completion, `floor()` and `floorf()` return the largest integral value not greater than x , expressed as a double or float respectively.

If x is NaN, NaN is returned and `errno` may be set to EDOM.

If the correct value would cause overflow, `-HUGE_VAL` is returned and `errno` is set to ERANGE.

If x is $\pm\text{Inf}$ or ± 0 , the value of x is returned.

Functions `ceil()` and `ceilf()`

Upon successful completion, `ceil()` and `ceilf()` return the smallest integral value not less than x , expressed as a type `double` or `float` respectively.

If x is NaN, NaN is returned and `errno` may be set to EDOM.

If the correct value would cause overflow, `HUGE_VAL` is returned and `errno` is set to ERANGE. If x is $\pm\text{Inf}$ or ± 0 , the value of x is returned.

Functions `fabs()` and `fabsf()`

Upon successful completion, `fabs()` and `fabsf()` return the absolute value of x .

If x is NaN, NaN is returned and `errno` may be set to EDOM.

If the result underflows, 0 is returned and `errno` may be set to ERANGE.

Functions `fmod()` and `fmodf()`

The `fmod()` and `fmodf()` function return the value $x - i * y$, for some integer i such that, if y is non-zero, the result has the same sign as x and magnitude less than the magnitude of y .

If x or y is NaN, NaN is returned and `errno` may be set to EDOM.

If y is 0, NaN is returned and `errno` is set to EDOM, or 0 is returned and `errno` may be set to EDOM.

If x is $\pm\text{Inf}$, either 0 is returned and `errno` is set to EDOM, or NaN is returned and `errno` may be set to EDOM.

If y is non-zero, `fmod( $\pm 0, y$ )` returns the value of x . If x is not $\pm\text{Inf}$, `fmod( $x, \pm\text{Inf}$ )` returns the value of x .

If the result underflows, 0 is returned and `errno` may be set to ERANGE.

Function `rint()`

Upon successful completion, the `rint()` function returns the integer (represented as a double precision number) nearest x in the direction of the current rounding mode.

When x is $\pm\text{Inf}$, `rint()` returns x .

If the value of x is NaN, NaN is returned and `errno` may be set to EDOM.

Function `remainder()`

The `remainder()` function returns the floating-point remainder $r = x - ny$ when y is non-zero.

When y is 0, `remainder()` returns (NaN or equivalent if available) and sets `errno` to EDOM.

If the value of x is $\pm\text{Inf}$, `remainder()` returns NaN and sets `errno` to EDOM.

If x or y is NaN, then the function returns NaN and `errno` may be set to EDOM.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Functions `floor()` and `floorf()`

The `floor()` and `floorf()` function will fail if:

ERANGE

The result would cause an overflow.

The `floor()` and `floorf()` function may fail if:

EDOM The value of x is NaN.

Functions `ceil()` and `ceilf()`

The `ceil()` and `ceilf()` function will fail if:

ERANGE

The result overflows.

The `ceil()` and `ceilf()` function may fail if:

EDOM The value of x is NaN.

Functions `fabs()` and `fabsf()`

The `fabs()` and `fabsf()` function may fail if:

EDOM The value of x is NaN.

ERANGE

The result underflows.

Functions `fmod()` and `fmodf()`

The `fmod()` and `fmodf()` function may fail if:

EDOM One or both of the arguments is NaN, or y is 0, or x is $\pm\text{Inf}$.

ERANGE

The result underflows.

Function `rint()`

The `rint()` function may fail if:

EDOM The x argument is NaN.

Function `remainder()`

The `remainder()` function will fail if:

EDOM The y argument is 0 or the x argument is positive or negative infinity.

The `remainder()` function may fail if:

EDOM The x or y argument is NaN.

No other errors will occur.

NOTES

The functions `floorf()`, `ceilf()`, `fmodf()`, and `fabsf()` can be used in ANSI mode only.

SEE ALSO

[abs\(3C\)](#), [isnan\(3C\)](#), [matherr\(3M\)](#), [math\(5\)](#).

NAME

`frexp`, `ldexp`, `logb`, `ilogb`, `nextafter`, `scalb`, `modf`, `modff` - manipulate parts of floating-point numbers

SYNOPSIS

```
cc [ flag ... ] file ... -lm [ library ... ]
#include <math.h>

double frexp(double num, int *exp);
double ldexp(double x, int exp);
double logb(double x);
int ilogb(double x);
double nextafter(double x, double y);
double scalb(double x, double n);
double modf(double x, double *iptr);
float modff(float value, float *iptr);□
```

DESCRIPTION

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

`ldexp()` computes the quantity

`logb()` computes the exponent of x , which is the integral part of $\log_r|x|$, as a signed floating-point value, for non-zero x , where r is the radix of the machine's floating-point arithmetic.

`ilogb()` computes the exponent part of x . Formally, the return value is the integral part of $\log_r|x|$ as a signed integral value, for non-zero x , where r is the radix of the machine's floating-point arithmetic.

The call `ilogb(x)` is equivalent to `(int)logb(x)`.

`nextafter()` computes the next representable double-precision floating-point value following x in the direction of y . Thus, if y is less than x , `nextafter()` returns the largest representable floating-point number less than x .

`scalb()` computes where r is the radix of the machine's floating point arithmetic. When r is 2, `scalb()` is equivalent to `ldexp()`.

`modf()` breaks the argument x into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a double in the object pointed to by `iptr`.

`modff()` is the single-precision version of `modf()`.

An application wishing to check for error situations should set `errno` to 0, then call the function. If `errno` is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE**Function frexp()**

The `frexp()` function returns the value x , such that x is a double with magnitude in the interval $[\frac{1}{2}, 1)$ or 0, and `num` equals x times 2 raised to the power `*exp`.

If `num` is 0, both parts of the result are 0.

If `num` is NaN, NaN is returned, `errno` may be set to EDOM and the value of `*exp` is unspecified.

If `num` is $\pm\text{Inf}$, `num` is returned, `errno` may be set to EDOM and the value of `*exp` is unspecified.

Function ldexp()

Upon successful completion, `ldexp()` returns a double representing the value x multiplied by 2 raised to the power exp .

If the value of x is NaN, NaN is returned and `errno` may be set to EDOM.

If `ldexp()` would cause overflow, `±HUGE_VAL` is returned (according to the sign of x), and `errno` is set to ERANGE.

If `ldexp()` would cause underflow, 0 is returned and `errno` may be set to ERANGE.

Function logb()

Upon successful completion, `logb()` returns the exponent of x .

If x is 0.0, `logb()` returns `-HUGE_VAL` and sets `errno` to EDOM.

If x is $±\text{Inf}$, `logb()` returns $+\text{Inf}$.

If x is NaN, `logb()` returns NaN and may set `errno` to EDOM.

Function ilogb()

Upon successful completion, `ilogb()` returns the exponent part of x .

If x is 0 or NaN, then `ilogb()` returns `INT_MIN`.

If x is $±\text{Inf}$, then `ilogb()` returns `INT_MAX`.

Function nextafter()

The `nextafter()` function returns the next representable double-precision floating-point value following x in the direction of y .

If x or y is NaN, then `nextafter()` returns NaN and may set `errno` to EDOM.

If x is finite and the correct function value would overflow, `HUGE_VAL` is returned and `errno` is set to ERANGE.

Function scalb()

Upon successful completion, the `scalb()` function returns

If the correct value would overflow, `scalb()` returns `±HUGE_VAL` (according to the sign of x) and sets `errno` to ERANGE.

If the correct value would underflow, `scalb()` returns 0 and sets `errno` to ERANGE.

The `scalb()` function returns x when x is $±\text{Inf}$. If x or n is NaN, then `scalb()` returns NaN and may set `errno` to EDOM.

Functions modf() and modff()

Upon successful completion, `modf()` returns the signed fractional part of x .

If x is NaN, NaN is returned, `errno` may be set to EDOM and `*iptr` is set to NaN.

If the correct value would cause underflow, 0 is returned and `errno` may be set to ERANGE.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Function frexp()

The `frexp()` function may fail if:

EDOM The value of num is NaN or $±\text{Inf}$.

Function ldexp()

The `ldexp()` function will fail if:

ERANGE

The value to be returned would have caused overflow.

The `ldexp()` function may fail if:

EDOM The argument x is NaN.

ERANGE

The value to be returned would have caused underflow.

Function `logb()`

The `logb()` function will fail if:

EDOM The x argument is 0.0.

The `logb()` function may fail if:

EDOM The x argument is NaN.

Function `ilogb()`

No errors are defined.

Function `nextafter()`

The `nextafter()` function will fail if:

ERANGE

The correct value would overflow.

The `nextafter()` function may fail if:

EDOM The x or y argument is NaN.

Function `scalb()`

The `scalb()` function will fail if:

ERANGE

The correct value would overflow or underflow.

The `scalb()` function may fail if:

EDOM The x or n argument is NaN.

Function `modf()` and `modff()`

The `modf()` and `modff()` functions may fail if:

EDOM The value of x is NaN.

ERANGE

The result underflows.

No other errors will occur.

NOTES

The function `modff()` can be used in ANSI mode only.

SEE ALSO

`math(5)`.

NAME

gamma, lgamma, signgam - log gamma function

SYNOPSIS

```
cc [ flag ... ] file ... -lm [ library ... ]
#include <math.h>

double gamma(double x);
double lgamma(double x);
extern int signgam;
```

DESCRIPTION

gamma() and lgamma() compute $\log_e |\Gamma(x)|$

where $\Gamma(x)$ is defined as

$$\int_0^{\infty} e^{-t} t^{x-1} dt$$

The sign of $\Gamma(x)$ is returned in the external integer signgam. The argument x need not be a non-positive integer [$\Gamma(x)$ is defined over the reals, except the non-positive integers].

An application wishing to check for error situations should set `errno` to 0, then call the function. If `errno` is non-zero on return, or the return value is NaN, an error has occurred.

The gamma() interface need not be thread-safe.

The following C program fragment might be used to calculate Γ :

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
    error();
y = signgam * exp(y);
```

where `LN_MAXDOUBLE` is the least value that causes `exp()` to return a range error, and is defined in the `<values.h>` header file.

RETURN VALUE

Upon successful completion, gamma() and lgamma() return the logarithmic gamma of x .

If x is NaN, NaN is returned and `errno` may be set to `EDOM`.

If x is a non-positive integer, either `HUGE_VAL` or NaN is returned and `errno` may be set to `EDOM`.

If the correct value would cause overflow, gamma() and lgamma() return `HUGE_VAL` and may set `errno` to `ERANGE`.

If the correct value would cause underflow, gamma() and lgamma() returns 0 and may set `errno` to `ERANGE`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The lgamma() function may fail if:

EDOM The value of x is a non-positive integer or NaN.

ERANGE

The value to be returned would have caused overflow or underflow.

No other errors will occur.

SEE ALSO

`isnan(3C)`, `exp(3M)`, `matherr(3M)`, `math(5)`, `values(5)`.

hypot(3M)**NAME**

`hypot` - Euclidean distance function

SYNOPSIS

```
cc [ flag ... ] file ... -lm [ library ... ]
#include <math.h>

double hypot (double x, double y);
```

DESCRIPTION

`hypot()` computes the length of the hypotenuse of a right-angled triangle:

$$\sqrt{x^2 + y^2}$$

An application wishing to check for error situations should set `errno` to 0 before calling `hypot()`. If `errno` is non-zero on return, or the return value is `HUGE_VAL` or `NaN`, an error has occurred.

RETURN VALUE

Upon successful completion, `hypot()` returns the length of the hypotenuse of a right angled triangle with sides of length `x` and `y`.

If the result would cause overflow, `HUGE_VAL` is returned and `errno` may be set to `ERANGE`.

If `x` or `y` is `NaN`, `NaN` is returned. and `errno` may be set to `EDOM`.

If the correct result would cause underflow, 0 is returned and `errno` may be set to `ERANGE`.

ERRORS

The `hypot` function may fail if:

`EDOM` The value of `x` or `y` is `NaN`.

`ERANGE`

The result overflows or underflows.

No other errors will occur.

APPLICATION USAGE

The `hypot()` function takes precautions against overflow during intermediate steps of the computation. If the calculated result would still overflow a double, then `hypot()` returns `HUGE_VAL`.

SEE ALSO

`isnan(3C)`, `matherr(3M)`, `sqrt(3M)`, `math(5)`.

ieee_functions(3M)
BSD Compatibility Package**NAME**

ieee_functions, fp_class, isnan, copysign, scalbn - miscellaneous functions for IEEE arithmetic

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <fp.h>
#include <math.h>
#include <stdio.h>
enum fp_class_type fp_class(x)
double x;
int isnan(x)
double x;
double copysign(x,y)
double x, y;
double scalbn(x,n)
double x; int n;
```

DESCRIPTION

Most of these functions provide capabilities required by ANSI/IEEE Standard 754-1985 or suggested in its appendix.

`fp_class(x)` corresponds to the IEEE's `class()` and classifies `x` as zero, subnormal, normal, ∞ , or quiet or signaling *NaN*; `/usr/ucbinclude/sys/ieeefp.h` defines `enum fp_class_type`. The following function returns 0 if the indicated condition is not satisfied:

`isnan(x)`
returns 1 if `x` is *NaN*.

`copysign(x,y)`
returns `x` with `y`'s sign bit.

`scalbn(x,n)` returns $x \cdot 2^n$ computed by exponent manipulation rather than by actually performing an exponentiation or a multiplication. Thus

$1 \leq \text{scalbn}(\text{fabs}(x), -\text{ilogb}(x)) < 2$

for every `x` except 0, ∞ , and *NaN*.

FILES

```
/usr/ucbinclude/sys/ieeefp.h
/usr/ucbinclude/math.h
/usr/include/values.h
```

NAME

`ieee_handler` - IEEE exception trap handler function

SYNOPSIS

```

/usr/ucb/cc [ flag ... ] file ... -lucb
#include <fp.h>
int ieee_handler(action, exception, hdl)
char action[], exception[];
sigfpe_handler_type hdl;

```

DESCRIPTION

This function provides easy exception handling to exploit ANSI/IEEE Standard 754-1985 arithmetic in a C program. All arguments are pointers to strings. Results arising from invalid arguments and invalid combinations are undefined for efficiency.

There are three types of *action*: "get", "set", and "clear". There are five types of *exception*:

"inexact"

"division"

... division by zero exception

"underflow"

"overflow"

"invalid"

"all"

... all five exceptions above

"common"

... invalid, overflow, and division exceptions

The exceptions "all" and "common" only make sense with "set" or "clear".

`hdl` contains the address of a signal-handling routine. `<fp.h>` defines `sigfpe_handler_type`.

"get" will find the location of the current handler routine for *exception* in `hdl`. "set" will set the routine pointed at by `hdl` to be the handler routine and at the same time enable the trap on *exception*, except when `hdl == SIGFPE_DEFAULT` or `SIGFPE_IGNORE`; then `ieee_handler` will disable the trap on *exception*. When `hdl == SIGFPE_ABORT`, any trap on *exception* will dump core using `abort(3C)`. "clear" "all" disables trapping on all five exceptions.

Two steps are required to intercept an IEEE-related `SIGFPE` code with `ieee_handler`:

- 1) Set up a handler with `ieee_handler`.
- 2) Perform a floating-point operation that generates the intended IEEE exception.

Unlike `sigfpe`, `ieee_handler` also adjusts floating-point hardware mode bits affecting IEEE trapping. The hardware trap is disabled when these actions are performed: "clear", "set" `SIGFPE_DEFAULT` or "set" `SIGFPE_IGNORE`.

The hardware trap is enabled when any other "set" is used.

`SIGFPE` signals can be handled using `sigvec(3)`, `signal(2)`, `signal(3-ucb)`, `sigfpe(3)`, or `ieee_handler(3M)`. In a particular program, to avoid confusion, use only one of these interfaces to handle `SIGFPE` signals.

RETURN VALUE

`ieee_handler` normally returns 0. In the case of set, 1 will be returned if the action is not available (for

instance, not supported in hardware).

EXAMPLES

A user-specified signal handler might look like this:

```
void sample_handler( sig, code, scp, addr)
int sig ;                /* sig == SIGFPE always */
int code ;
struct sigcontext *scp ;
char *addr ;
{
    /*
     * Sample user-written sigfpe code handler.
     * Prints a message and continues.
     * struct sigcontext is defined in <signal.h>.
     */
    printf("ieee exception code %x occurred at pc %X \n",
           code, scp->sc_pc);
}
```

and it might be set up like this:

```
extern void sample_handler;
main
{
    sigfpe_handler_type hdl, old_handler1, old_handler2;
    /*
     * save current overflow and invalid handlers
     */
    ieee_handler("get", "overflow", old_handler1);
    ieee_handler("get", "invalid", old_handler2);
    /*
     * set new overflow handler to sample_handler and set new
     * invalid handler to SIGFPE_ABORT (abort on invalid)
     */
    hdl = (sigfpe_handler_type) sample_handler;
    if(ieee_handler("set", "overflow", hdl) != 0)
        printf("ieee_handler can't set overflow \n");
    if(ieee_handler("set", "invalid", SIGFPE_ABORT) != 0)
        printf("ieee_handler can't set invalid \n");
    ...
    /*
     * restore old overflow and invalid handlers
     */
    ieee_handler("set", "overflow", old_handler1);
    ieee_handler("set", "invalid", old_handler2);
}
```

FILES

```
/usr/include/fp.h
/usr/include/signal.h
```

SEE ALSO

[signal\(2\)](#), [sigfpe\(3\)](#), [sigvec\(3\)](#), [signal\(3-ucb\)](#), [abort\(3C\)](#), [floatingpoint\(3M\)](#).

NAME

matherr - error-handling function

SYNOPSIS

```
cc [ flag ... ] file ... -lm [ library ... ]
#include <math.h>

int matherr(struct exception *x);
```

DESCRIPTION

matherr() is invoked by functions in the math libraries when errors are detected. Note that matherr() is not invoked when the -kansi compilation option is used. Users may define their own procedures for handling errors, by including a function named matherr() in their programs. matherr() must be of the form described above. When an error occurs, a pointer to the exception structure x will be passed to the user-supplied matherr() function. This structure, which is defined in the math.h header file, is as follows:

```
struct exception {
    int type;
    char *name;
    double arg1, arg2, retval;
};
```

The element type is a number of type int describing the type of error that has occurred, from the following list of constants (defined in the header file):

- DOMAIN argument domain error
- SING argument singularity
- OVERFLOW overflow range error
- UNDERFLOW underflow range error
- TLOSS total loss of significance
- PLOSS partial loss of significance

The element name points to a string containing the name of the function that incurred the error. The variables arg1 and arg2 are the arguments with which the function was invoked. retval is set to the default value that will be returned by the function unless matherr() sets it to a different value.

If the matherr() function returns non-zero, no error message will be printed, and errno will not be set. If matherr() is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, errno is set to EDOM or ERANGE and the program continues.

Default Error Handling Procedures						
	Types of Errors					
type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
		M				
BESSEL:	-	-	-	-	M, 0	-
y0, y1, yn (arg £0)	M, -H	-	-	-	-	-

EXP, EXPF:	–	–	H	0	–	–
LOG, LOG10: LOGF, LOG10F: (arg < 0) (arg = 0)	M, –H M, –H	– –	– –	– –	– –	– –
POW, POWF: neg ** non-int 0 ** non-pos	– M, 0 M, 0	– – –	H – –	0 – –	– – –	– – –
SQRT, SQRTF:	M, 0	–	–	–	–	–
FMOD, FMODF: (arg2 = 0)	M, X	–	–	–	–	–
REMAINDER: (arg2 = 0)	M, N	–	–	–	–	–
GAMMA, LGAMMA:	–	M, H	H	–	–	–
HYPOT:	–	–	H	–	–	–
SINH, SINHF:	–	–	H	–	–	–
COSH, COSHF:	–	–	H	–	–	–
ASIN, ACOS, ATAN2: ASINF, ACOSF, ATAN2F:	M, 0	–	–	–	–	–
ACOSH:	M, N	–	–	–	–	–
ATANH: (arg > 1) (arg = 1)	M, N –	– M, N	– –	– –	– –	– –

Abbreviations

- M** Message is printed (not in the ANSI mode).
H HUGE is returned (HUGE_VAL in the ANSI mode).
–H –HUGE is returned (–HUGE_VAL in the ANSI mode).
H HUGE or –HUGE is returned (HUGE_VAL or –HUGE_VAL in the ANSI mode).
0 0 is returned.
X arg1 is returned.
N NaN is returned.

EXAMPLE

```

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```

int
matherr(register struct exception *x);
{
    switch (x->type) {
    case DOMAIN:
        /* change sqrt to return sqrt(-arg1), not 0 */
        if (!strcmp(x->name, "sqrt")) {
            x->retval = sqrt(-x->arg1);
            return (0); /* print message and set errno */
        }
    case SING:
        /* all other domain or sing errors, print message */
        /* and abort */
        fprintf(stderr, "domain error in %s\n", x->name);
        abort( );
    case PLOSS:
        /* print detailed
error message */
        fprintf(stderr, "loss of significance in %s(%g)=%g\n"
            x->name, x->arg1, x->retval);
        return (1); /* take no other action */
    }
    return (0); /* all other errors, execute default procedure */
}

```

SEE ALSO

[math\(5\)](#).

NAME

`sinh`, `sinhf`, `cosh`, `coshf`, `tanh`, `tanhf`, `asinh`, `acosh`, `atanh` - hyperbolic functions

SYNOPSIS

```
cc [ flag ... ] file ... -lm [ library ... ]
#include <math.h>

double sinh(double x);
float  sinhf(float x);
double cosh(double x);
float  coshf(float x);
double tanh(double x);
float  tanhf(float x);
double asinh(double x);
double acosh(double x);
double atanh(double x);□
```

DESCRIPTION

The `sinh()` function computes the hyperbolic sine of x .

`sinhf()` is the single-precision version of `sinh()`.

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

`coshf()` is the single-precision version of `cosh()`.

The `tanh()` function computes the hyperbolic tangent of x .

`tanhf()` is the single-precision version of `tanh()`.

The `acosh()`, `asinh()` and `atanh()` functions compute the inverse hyperbolic cosine, sine, and tangent of their argument, respectively.

An application wishing to check for error situations should set `errno` to 0, then call the function. If `errno` is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE**Functions `sinh()` and `sinhf()`**

Upon successful completion, `sinh()` returns the hyperbolic sine of x .

If the result would cause an overflow, `±HUGE_VAL` is returned and `errno` is set to `ERANGE`.

If the result would cause underflow, 0.0 is returned and `errno` may be set to `ERANGE`.

If x is NaN, NaN is returned and `errno` may be set to `EDOM`.

Functions `cosh()` and `coshf()`

Upon successful completion, `cosh()` and `coshf()` return the hyperbolic cosine of x .

If the result would cause an overflow, `HUGE_VAL` is returned and `errno` is set to `ERANGE`.

If x is NaN, NaN is returned and `errno` may be set to `EDOM`.

Functions `tanh()` and `tanhf()`

Upon successful completion, `tanh()` and `tanhf()` return the hyperbolic tangent of x .

If x is NaN, NaN is returned and `errno` may be set to `EDOM`.

If the correct value would cause underflow, 0.0 is returned and `errno` may be set to `ERANGE`.

Functions `acosh()`, `asinh()` and `atanh()`

The `acosh()`, `asinh()` and `atanh()` functions return the inverse hyperbolic cosine, sine, and tangent of their argument, respectively.

The `acosh()` function returns an implementation-dependent value (NaN or equivalent if available) and sets `errno` to `EDOM` when its argument is less than 1.0.

The `atanh()` function returns an implementation-dependent value (NaN or equivalent if available) and sets `errno` to `EDOM` when its argument has absolute value greater than 1.0.

If x is NaN, the `asinh()`, `acosh()` and `atanh()` functions return NaN and may set `errno` to `EDOM`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Functions `sinh()` and `sinhf()`

The `sinh()` and `sinhf()` function will fail if:

`ERANGE`

The result would cause overflow.

The `sinh()` and `sinhf()` function may fail if:

`EDOM` The value of x is NaN.

`ERANGE`

The result would cause underflow.

Functions `cosh()` and `coshf()`

The `cosh()` and `coshf()` function will fail if:

`ERANGE`

The result would cause an overflow.

The `cosh()` and `coshf()` function may fail if:

`EDOM` The value of x is NaN.

Functions `tanh()` and `tanhf()`

The `tanh()` and `tanhf()` function may fail if:

`EDOM` The value of x is NaN.

`ERANGE`

The correct result would cause underflow.

Functions `acosh()`, `asinh()` and `atanh()`

The `acosh()` function will fail if:

`EDOM` The x argument is less than 1.0.

The `atanh()` function will fail if:

`EDOM` The x argument has an absolute value greater than 1.0.

The `atanh()` function will fail if:

`ERANGE`

The x argument has an absolute value equal to 1.0

The `asinh()`, `acosh()` and `atanh()` functions may fail if:

`EDOM` The value of x is NaN.

No other errors will occur.

NOTES

The functions `sinhf()`, `coshf()`, and `tanhf()` can be used in ANSI mode only.

SEE ALSO

`isnan(3C)`, `matherr(3M)`, `math(5)`.

NAME

trig:
 acos, acosf, asin, asinf, atan, atanf, atan2, atan2f, cos, cosf, sin, sinf, tan,
 tanf - trigonometric functions

SYNOPSIS

```
cc [ flag ... ] file ... -lm [ library ... ]
cc -O -Ksd [ flag ... ] file ... -J sfm
[library ...]
#include <math.h>
double acos(double x);
float acosf(float x);
double asin(double x);
float asinf(float x);
double atan(double x);
float atanf(float x);
double atan2(double y, double x);
float atan2f(float y, float x);
double cos(double x);
float cosf(float x);
double sin(double x);
float sinf(float x);
double tan(double x);
float tanf(float x);
```

DESCRIPTION

The `acos()` function computes the principal value of the arc cosine of x . The value of x should be in the range $[-1,1]$.

`acosf()` is the single-precision version of `acos()`.

The `asin()` function computes the principal value of the arc sine of x . The value of x should be in the range $[-1,1]$.

`asinf()` is the single-precision version of `asin()`.

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

`atanf()` is the single-precision version of `atan()`.

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

`atan2f()` is the single-precision version of `atan2()`.

The `cos()` function computes the cosine of x , measured in radians.

`cosf()` is the single-precision version of `cos()`.

The `sin()` function computes the sine of its argument x , measured in radians.

The `sin()` function may lose accuracy when its argument is far from 0.0.

`sinf()` is the single-precision version of `sin()`.

The `tan()` function computes the tangent of its argument x , measured in radians.

The `tan()` function may lose accuracy when its argument is far from 0.0.

`tanf()` is the single-precision version of `tan()`.

An application wishing to check for error situations should set `errno` to 0, then call the function. If `errno` is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

Functions `acos()` and `acosf()`

Upon successful completion, `acos()` and `acosf()` return the arc cosine of x , in the range $[0, \pi]$ radians. If the value of x is not in the range $[-1, 1]$, and is not $\pm\text{Inf}$ or NaN, either 0.0 or NaN is returned and `errno` is set to EDOM.

If x is NaN, NaN is returned and `errno` may be set to EDOM. If x is $\pm\text{Inf}$, either 0.0 is returned and `errno` is set to EDOM, or NaN is returned and `errno` may be set to EDOM.

Functions `asin()` and `asinf()`

Upon successful completion, `asin()` and `asinf()` return the arc sine of x , in the range $[-\pi/2, \pi/2]$ radians. If the value of x is not in the range $[-1, 1]$, and is not $\pm\text{Inf}$ or NaN, either 0.0 or NaN is returned and `errno` is set to EDOM.

If x is NaN, NaN is returned and `errno` may be set to EDOM.

If x is $\pm\text{Inf}$, either 0.0 is returned and `errno` is set to EDOM or NaN is returned and `errno` may be set to EDOM.

If the result underflows, 0.0 is returned and `errno` may be set to ERANGE.

Functions `atan()` and `atanf()`

Upon successful completion, `atan()` and `atanf()` return the arc tangent of x in the range $[-\pi/2, \pi/2]$ radians.

If x is NaN, NaN is returned and `errno` may be set to EDOM.

If the result underflows, 0.0 is returned and `errno` may be set to ERANGE.

Functions `atan2()` and `atan2f()`

Upon successful completion, `atan2()` and `atan2f()` return the arc tangent of y/x in the range $[-\pi/2, \pi/2]$ radians. If both arguments are 0.0, an implementation-dependent value is returned and `errno` may be set to EDOM.

If x or y is NaN, NaN is returned and `errno` may be set to EDOM.

If the result underflows, 0.0 is returned and `errno` may be set to ERANGE.

Functions `cos()` and `cosf()`

Upon successful completion, `cos()` and `cosf()` return the cosine of x .

If x is NaN, NaN is returned and `errno` may be set to EDOM.

If x is $\pm\text{Inf}$, either 0 is returned and `errno` is set to EDOM, or NaN is returned and `errno` may be set to EDOM.

If the result underflows, 0 is returned and `errno` may be set to ERANGE.

Functions `sin()` and `sinf()`

Upon successful completion, `sin()` and `sinf()` return the sine of x .

If x is NaN, NaN is returned and `errno` may be set to EDOM.

If x is $\pm\text{Inf}$, either 0.0 is returned and `errno` is set to EDOM, or NaN is returned and `errno` may be set to EDOM.

If the correct result would cause underflow, 0.0 is returned and `errno` may be set to ERANGE.

Functions `tan()` and `tanf()`

Upon successful completion, `tan()` and `tanf()` return the tangent of x .

If x is NaN, NaN is returned and `errno` may be set to `EDOM`.

If x is $\pm\text{Inf}$, either 0.0 is returned and `errno` is set to `EDOM`, or NaN is returned and `errno` may be set to `EDOM`.

If the correct value would cause overflow, `±HUGE_VAL` is returned and `errno` is set to `ERANGE`.

If the correct value would cause underflow, 0.0 is returned and `errno` may be set to `ERANGE`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Functions `acos()` and `acosf()`

The `acos()` or `acosf()` function will fail if:

`EDOM` The value x is not $\pm\text{Inf}$ or NaN and is not in the range $[-1, 1]$.

The `acos()` or `acosf()` function may fail if:

`EDOM` The value x is $\pm\text{Inf}$ or NaN.

Functions `asin()` and `asinf()`

The `asin()` or `asinf()` function will fail if:

`EDOM` The value x is not $\pm\text{Inf}$ or NaN and is not in the range $[-1, 1]$.

The `asin()` or `asinf()` function may fail if:

`EDOM` The value of x is $\pm\text{Inf}$ or NaN.

`ERANGE`

The result underflows.

Functions `atan()` and `atanf()`

The `atan()` or `atanf()` function may fail if:

`EDOM` The value of x is NaN.

`ERANGE`

The result underflows.

Functions `atan2()` and `atan2f()`

The `atan2()` or `atan2f()` function may fail if:

`EDOM` Both arguments are 0.0 or one or more of the arguments is NaN.

`ERANGE`

The result underflows.

Functions `cos()` and `cosf()`

The `cos()` or `cosf()` function may fail if:

`EDOM` The value of x is NaN or x is $\pm\text{Inf}$.

`ERANGE`

The result underflows.

Functions `sin()` and `sinf()`

The `sin()` or `sinf()` function may fail if:

`EDOM` The value of x is NaN, or x is $\pm\text{Inf}$.

`ERANGE`

The result underflows.

Functions `tan()` and `tanf()`

The `tan()` or `tanf()` function will fail if:

ERANGE

The value to be returned would cause overflow.

The `tan()` or `tanf()` function may fail if:

EDOM The value x is NaN or $\pm\text{Inf}$.

ERANGE

The value to be returned would cause underflow.

No other errors will occur.

NOTES

The functions `sinf()`, `cosf()`, `tanf()`, `asinf()`, `acosf()`, `atanf()`, and `atan2f()` can be used in ANSI mode only.

SEE ALSO

[isnan\(3C\)](#), [matherr\(3M\)](#), [math\(5\)](#).

Standard input/output functions from the standard library libc(3S)

NAME

`ctermid` - generate filename for terminal

SYNOPSIS

```
#include <stdio.h>
□
□
char *ctermid(char *s); □
```

DESCRIPTION

The `ctermid()` function generates a string that, when used as a pathname, refers to the current controlling terminal for the current process. If `ctermid()` returns a pathname, access to the file is not guaranteed.

If the application uses any of the `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS` interfaces, the `ctermid()` function must be called with a non-NULL parameter.

RETURN VALUE

If s is a null pointer, the string is generated in an area that may be static (and therefore may be overwritten by each call), the address of which is returned. Otherwise s is assumed to point to a character array of at least `L_ctermid` bytes; the string is placed in this array and the value of s is returned. The symbolic constant `L_ctermid` is defined in `<stdio.h>`, and will have a value greater than 0.

The `ctermid()` function will return an empty string if the pathname that would refer to the controlling terminal cannot be determined, or if the function is unsuccessful.

ERRORS

No errors are defined.

NOTES

The difference between `ctermid()` and [ttyname\(3C\)](#) is as follows: `ttyname()` expects to receive 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 filename. Thus `ttyname()` is useful only if the process already has at least one file open to a terminal.

SEE ALSO

`ttyname(3C)`, `stdio(5)`.

NAME

`cuserid` - generate string for user's name

SYNOPSIS

```
#include <stdio.h>
□
□
char *cuserid(char *s); □
```

DESCRIPTION

`cuserid()` generates a character-string representation of the name associated with the effective user ID of the current process. If `s` is a null pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, `s` is assumed to point to an array of at least `L_cuserid` characters; the representation is left in this array. The constant `L_cuserid` is defined in the `stdio.h` header file.

RESULT

If the login name cannot be found, `cuserid()` returns a `NULL` pointer; if `s` is not a `NULL` pointer, a null character `\0` will be placed at `s[0]`. If `s` is a `NULL` pointer and the login name can be found, the address of a (static) buffer containing the login name is returned.

SEE ALSO

[getlogin\(3C\)](#), [getpwent\(3C\)](#).

fclose(3S)**NAME**

`fclose`, `fflush` - close or flush a stream

SYNOPSIS

```
#include <stdio.h>
□
□
int fclose(FILE *stream) ;
int fflush(FILE *stream) ; □
```

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. It marks for update the `st_ctime` and `st_mtime` fields of the underlying file, if the stream was writable, and if buffered data had not been written to the file yet. The `fclose()` function will perform a `close()` 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.

If *stream* points to an output stream or an update stream on which the most recent operation was not input, `fflush()` causes any buffered data for the named *stream* to be written to that file. The `st_ctime` and `st_mtime` fields of the underlying file are marked for update.

When calling `fflush()`, if *stream* is a null pointer, all files open for writing are flushed.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `fclose()` or `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' file size limit.

EFBIG

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 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. 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 process.

The `fclose()` or `fflush()` function may fail if:

`ENXIO`

A request was made of a non-existent device, or the request was outside the device.

RESULT

Upon successful completion these functions return a value of zero. Otherwise `EOF` is returned and `errno` is set to indicate the error.

SEE ALSO

`close(2)`, `exit(2)`, `fopen(3S)`, `setbuf(3S)`, `stdio(3S)`, `lfs(5)`, `stdio(5)`.

NAME

`ferror`, `feof`, `clearerr`, `fileno` - **streamstatus inquiries**

SYNOPSIS

```
#include <stdio.h>
□
□
int ferror(FILE *Stream);
int feof(FILE *Stream);
void clearerr(FILE *Stream);
int fileno(FILE *Stream); □
```

DESCRIPTION

`ferror()` returns non-zero when an error has previously occurred reading from or writing to the named *Stream*, otherwise zero.

`feof()` returns non-zero when EOF has previously been detected reading the named input *Stream*, otherwise zero.

`clearerr()` resets the error indicator and EOF indicator to zero on the named *Stream*.

`fileno()` returns the integer file descriptor associated with the named *Stream* [see [open\(2\)](#)].

SEE ALSO

[open\(2\)](#), [fopen\(3S\)](#), [stdio\(3S\)](#).

fgetwc(3S)**NAME**

`fgetwc`, `getwc`, `getwchar` - get a wide character code from a stream

SYNOPSIS

```
cc [ flag ... ] file ... -lw [ library ... ]
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
```

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 `ferror()` or `feof()` functions must be used to distinguish between an error condition and an end-of-file condition.

The `getwc()` function is equivalent to `fgetwc()`, except that if it is implemented as a macro it can evaluate *stream* more than once, so the argument should never be an expression with side-effects. In particular, `getwc(*f++)` may not work as expected. Therefore, use of this function is not recommended; the `fgetwc()` function should be used instead.

The function call `getwchar()` is equivalent to `getwc(stdin)`. If the 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.

The 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()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` or `scanf()` using *stream* that returns data not supplied by a prior call to `ungetc()` or `ungetwc()`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

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 the 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 `fgetwc()` function will 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.

RESULT

Upon successful completion the `fgetwc()` function returns the wide character code of the character read from the input stream pointed to by *stream* of type `wint_t`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `fgetwc()` returns `WEOF`. If a read error occurs, the error indicator for the stream is set, `fgetwc()` returns a null pointer and sets `errno` to indicate the error.

SEE ALSO

`feof(3S)`, `ferror(3S)`, `fopen(3S)`, `lfs(5)`, `stdio(5)`, `wchar(5)`.

NAME

`fgetws` - get wide character string from a stream

SYNOPSIS

```
cc [ flag ... ] file ... -lw [ library ... ]
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t *ws, int n, FILE *stream);
```

DESCRIPTION

The `fgetws()` function reads characters from the *stream*, converts these to the corresponding wide-character codes, places them in the `wchar_t` array pointed to by *ws*, until *n*-1 characters are read, or 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()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `getc()`, `getchar()`, `fgets()` or `scanf()` using *stream* that returns data not supplied by a prior call to `ungetc()` or `ungetwc()`.

ERRORS

The `fgetws()` 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

The process is in a background process group attempting to read from the 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 `fgetws()` function will 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.

RESULT

Upon successful completion the `fgetws()` function returns *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. If a read error occurs, the error indicator for the stream is set, `fgetws()` returns a null pointer and sets `errno` to indicate the error.

SEE ALSO

`fopen(3S)`, `fread(3S)`, `lfs(5)`, `wchar(5)`.

NAME

`fopen`, `freopen`, `fdopen`, `fopen64`, `freopen64` - open a stream

SYNOPSIS

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *type);
FILE *freopen(const char *filename, const char *type, FILE *stream);
FILE *fdopen(int fildes, const char *type);
FILE *fopen64(const char *filename, const char *mode);
FILE *freopen64(const char *filename, const char *mode, FILE *stream);
```

DESCRIPTION

`fopen()` opens the file named by *filename* and returns a pointer to the `FILE` structure associated with the *filename*. *filename* must point to a string containing the name of the file to be opened. The type of access required is specified in the *type* string, which can have one of the following values.

`r` or `rb`

Open file for reading

`w` or `wb`

Truncate file to zero length for writing, or create

`a` or `ab`

Append; open for writing at end of file, or create for writing

`r+`, `r+b` or `rb+`

Open file for update (reading and writing)

`w+`, `w+b` or `wb+`

Truncate or create for update

`a+`, `a+b` or `ab+`

Append; open or create for update at end-of-file

The `b` is ignored in the above types. It exists to distinguish binary files from text files. However, there is no distinction between these types of files on a Reliant UNIX system.

`freopen()` substitutes the opened file designated by *stream* with the file whose pathname is pointed to by *filename*. A flush is first attempted, and then the original *stream* is closed, regardless of whether the open ultimately succeeds. Failure to flush or close *stream* successfully is ignored. `freopen()` returns a pointer to the `FILE` structure associated with *stream*.

After a successful call to the `freopen()` function, the orientation of the stream is cleared and the associated `mbstate_t` object is set to describe an initial conversion state.

`freopen()` is typically used to attach the preopened *streams* associated with `stdin`, `stdout`, and `stderr` to other files. `stderr` is by default unbuffered, but the use of `freopen()` will cause it to become buffered or line-buffered.

`fdopen()` associates a stream with a file descriptor. File descriptors are obtained from `open()`, `dup()`, `creat()`, or `pipe()`, which open files but do not return pointers to a `FILE` structure *stream*. Streams are necessary input for almost all of the Section 3S library routines. The stream type must agree with the mode of the open file. The file position indicator associated with the stream is set to the position indicated by the file offset associated with *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.

The `fdopen()` function will preserve the offset maximum previously set for the open file description

corresponding to *fildest*.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening `fflush()`, `fseek()`, `fsetpos()`, or `rewind()`, and input may not be directly followed by output without an intervening `fseek()`, `fsetpos()`, or `rewind()`, or an input operation that encounters end-of-file.

When a file is opened for append (i.e., when *type* is `a`, `ab`, `a+`, or `ab+`), it is impossible to overwrite information already in the file. `fseek()` may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

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 are cleared for the stream when it is opened.

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.

There is no functional difference between `fopen()/freopen()` and `fopen64()/freopen64()`, except for the interpretation of `off64_t` [see [lfs\(5\)](#)].

RETURN VALUE

The functions `fopen()` and `freopen()` return a null pointer if *filename* cannot be accessed, or if *type* is invalid, or if the file cannot be opened.

The function `fdopen()` returns a null pointer if *fildest* is not an open file descriptor, or if *type* is invalid, or if the file cannot be opened.

The functions `fopen()` or `fdopen()` will fail and not set `errno` if there are no free [stdio\(3S\)](#) streams.

File descriptors used by `fdopen()` must be less than 255.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

Function `fopen()`

`fopen()` will fail if:

EACCES

Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *type* 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 `fopen()` function.

EISDIR

The named file is a directory and *type* requires write access.

ELOOP

Too many symbolic links were encountered in resolving *path*.

EMFILE

`OPEN_MAX` file descriptors are 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 on the system.

ENOENT

The named file does not exist or the *filename* argument points to an empty string.

ENOSPC

The file does not exist and the directory that would contain the new file cannot be expanded.

ENOTDIR

A component of the path prefix is not a directory.

ENXIO

The file is a special file and cannot be accessed.

EOVERFLOW

The named file is a regular file and the size of the file cannot be represented correctly in an object of type `off_t`.

EROFS

The file resides on a read-only file system, and *type* requires write access.

The `fopen()` function may fail if:

EINVAL

The value of the *type* argument is not valid.

EMFILE

`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 *type* requires write access.

Function `freopen()`

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.

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

`OPEN_MAX` file descriptors are 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 named file is a regular file and the size of the file 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.

Function `fdopen()`

The `fdopen()` function may fail if:

EBADF

The *fildev* 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.

EMFILE

`STREAM_MAX` streams are currently open in the calling process.

ENOMEM

Insufficient space to allocate a buffer.

APPLICATION USAGE

`STREAM_MAX` is the number of streams that one process can have open at one time. If defined, it has the same value as `FOPEN_MAX`.

SEE ALSO

`close(2)`, `creat(2)`, `dup(2)`, `open(2)`, `pipe(2)`, `write(2)`, `fclose(3S)`, `fseek(3S)`, `setbuf(3S)`, `stdio(3S)`, `lfs(5)`, `stdio(5)`.

fopen(3S-ucb)
BSD Compatibility Package**NAME**

`fopen`, `freopen` - open a stream

SYNOPSIS

```

/usr/ucb/cc [ flag ... ] file ... -lucb
#include <stdio.h>
FILE *fopen(filename, type)
char *filename, *type;
FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;

```

DESCRIPTION

`fopen` opens the file named by *filename* and associates a stream with it. If the open succeeds, `fopen` returns a pointer to be used to identify the stream in subsequent operations.

filename points to a character string that contains the name of the file to be opened.

type is a character string having one of the following values:

- `r` Open for reading.
- `w` Truncate or create for writing.
- `a` Append: open for writing at end of file, or create for writing.
- `r+` Open for update (reading and writing).
- `w+` Truncate or create for update.
- `a+` Append; open or create for update at EOF.

`freopen` opens the file named by *filename* and associates the stream pointed to by *stream* with it. The *type* argument is used just as in `fopen`. The original stream is closed, regardless of whether the open ultimately succeeds. If the open succeeds, `freopen` returns the original value of *stream*.

`freopen` is typically used to attach the preopened streams associated with `stdin`, `stdout`, and `stderr` to other files.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening `fseek` or `rewind`, and input may not be directly followed by output without an intervening `fseek`, `rewind`, or an input operation which encounters EOF.

RETURN VALUE

`fopen` and `freopen` return a NULL pointer on failure.

NOTES

`fopen` differs from the library routine of the same name in the base system only in interface.

In order to support the same number of open files that the system does, `fopen` must allocate additional memory for data structures using `calloc` after 64 files have been opened. This confuses some programs which use their own memory allocators.

SEE ALSO

`open(2)`, `pipe(2)`, `malloc(3C)`, `fclose(3S)`, `fopen(3S)`, `fseek(3S)`, `stdio(5)`.

NAME

fputc, putwc, putwchar - put wide character code on astream

SYNOPSIS

```
cc [ flag ... ] file ... -lw [ library ... ]
#include <stdio.h>
#include <wchar.h>
wint_t fputc(wchar_t wc, FILE *stream);
wint_t putwc(wchar_t wc, FILE *stream);
wint_t putwchar(wchar_t wc); □
```

DESCRIPTION

The `fputc()` 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.

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()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

The `putwc()` function is equivalent to `fpuwc()`, 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. Because it may be implemented as a macro, `putwc()` may treat a stream argument with side-effects incorrectly. In particular, `putwc(wc, *f++)` may not work correctly. Therefore, use of this function is not recommended; the `fputc()` function should be used instead.

The function call `putwchar(wc)` is equivalent to `putwc(wc, stdout)`.

RETURN VALUE

Upon successful completion, `fputc()` returns `wc`. Otherwise it returns `WEOF`, the error indicator for the stream is set, and `errno` is set to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `fputc()` function will fail if either the stream is unbuffered or data in the `streams` 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 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 [see [ulimit\(2\)](#)].

EFBIG

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 `SIGTOU` 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 a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the process.

The `fputwc()` 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 wide character code `wc` does not correspond to a valid character.

SEE ALSO

`ferror(3S)`, `fopen(3S)`, `setbuf(3S)`, `lfs(5)`, `stdio(5)`, `wchar(5)`.

NAME

fputws - put wide character string on a stream

SYNOPSIS

```
cc [ flag ... ] file ... -lw [ library ... ]
#include <stdio.h>
#include <wchar.h>
int fputws(const wchar_t *ws, FILE *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.

The `fputws()` function does not append a newline character.

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()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `fputws()` function will fail if either the stream is unbuffered or data in the `streams` 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 file size limit [see [ulimit\(2\)](#)].

EFBIG

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

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 a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the process.

The `fgetws()` function will 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.

RESULT

Upon successful completion, the `fputws()` function returns a non-negative number. Otherwise it returns `-1`, sets an error indicator for the stream and `errno` is set to indicate the error.

SEE ALSO

`fopen(3S)`, `lfs(5)`, `stdio(5)`, `wchar(5)`.

fread(3S)**NAME**

`fread`, `fwrite` - binary input/output

SYNOPSIS

```
#include <stdio.h>
□
□
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream); □
```

DESCRIPTION

`fread()` reads into an array pointed to by *ptr* up to *nitems* items of data from *stream*, where an item of data is a sequence of bytes of length *size*. The sequence is not necessarily terminated by a null byte. `fread()` stops reading bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. `fread()` increments the data pointer in *stream* to point to the byte following the last byte read if there is one. `fread()` does not change the contents of *stream*. `fread()` returns the number of items read. The return value is less than *nitems* only if a read error or an EOF occurs.

`fwrite()` writes to the named output *stream* at most *nitems* items of data from the array pointed to by *ptr*, where an item of data is a sequence of bytes of length *size*. This sequence is not necessarily terminated by a null byte. `fwrite()` stops writing when it has written *nitems* items of data or if an error condition is encountered on *stream*. `fwrite()` does not change the contents of the array pointed to by *ptr*. `fwrite()` increments the data-pointer in *stream* by the number of bytes written. `fwrite()` returns the number of items written.

If *size* or *nitems* is zero, then `fread()` and `fwrite()` return a value of 0. The contents of the array pointed to by *ptr* and the state of *stream* remain unchanged.

The `ferror()` or `feof()` routines must be used to distinguish between an error condition and end-of-file condition.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The `fread()` function will fail if:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor underlying *stream*, and the process is delayed.

EBADF

The file descriptor underlying *stream* is not a valid file descriptor opened for reading.

EINTR

The read operation was terminated due to the receipt of a signal; 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 `fread()` 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.

The `fwrite()` 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 a valid 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 [see `ulimit(2)`].

EFBIG

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 `SIGTOU` 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 a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the process.

The `fwrite()` 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.

RESULT

If an error occurs, the error indicator for *stream* is set.

SEE ALSO

`exit(2)`, `lseek(2)`, `read(2)`, `write(2)`, `abort(3C)`, `fclose(3S)`, `fopen(3S)`, `getc(3S)`, `gets(3S)`, `printf(3S)`, `putc(3S)`, `puts(3S)`, `scanf(3S)`, `stdio(3S)`, `lfs(5)`, `stdio(5)`.

NAME

`fseek`, `rewind`, `ftell`, `fseeko`, `fseeko64`, `ftello`, `ftello64` - reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek(FILE *stream, long int offset, int whence);
void rewind(FILE *stream);
long int ftell(FILE *stream);
int fseeko(FILE *stream, off_t offset, int whence);
int fseeko64(FILE *stream, off64_t offset, int whence);
off_t ftello(FILE *stream);
off64_t ftello64(FILE *stream); □
```

DESCRIPTION

`fseek()` sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according to a *whence* value of `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, 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 zero or a value returned by an earlier `ftell()` call on the same *stream*, and *prname* must be `SEEK_SET`.

`fseek()` allows the file position indicator to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return zero until data is actually written into the gap.

`rewind(stream)` is equivalent to:

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

except that `rewind()` also clears the error indicator on *stream*.

`fseek()` and `rewind()` clear the EOF indicator and undo any effects of `ungetc()` on *stream*.

After `fseek()` or `rewind()`, the next operation on a file opened for update may be either input or output.

If *stream* is writable and buffered data has not been written to the underlying file, `fseek()` and `rewind()` cause the unwritten data to be written to the file.

`ftell()` returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

The `fseeko()` and `ftello()` functions are identical to the modified `fseek()` and `ftell()`, except that the *offset* argument is of type `off_t` and the `EOVERFLOW` error has changed.

`fseeko()` and `ftello()` are needed because `fseek()` and `ftell()` are limited by the `long` offset type required by ISO C.

There is no functional difference between `fseeko()/ftello()` and `fseeko64()/ftello64()`, except for the interpretation of `off64_t` [see [lfs\(5\)](#)].

RETURN VALUE

Function `fseek()` and `fseeko()`

`fseek()` returns `-1` for improper seeks, otherwise zero. An improper seek can be, for example, an `fseek()` done on a file that has not been opened via `fopen()`; in particular, `fseek()` may not be used on a terminal or on a file opened via `popen(3S)`. After a stream is closed, no further operations are defined on that stream.

Function `rewind()`

The `rewind()` returns no value.

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.

Function `ftell()` and `ftello()`

Upon successful completion, `ftell()` and `ftello()` return the current value of the file-position indicator for the stream measured in bytes from the beginning of the file.

Otherwise, `ftell()` and `ftello()` return `-1`, cast to `long` and `off_t` respectively, and set `errno` to indicate the error.

ERRORS

The following error code descriptions are function-specific. You will find a general description in `intro_prm2(2)` or in `errno(5)`.

Function `fseek()` and `fseeko()`

`fseek()` and `fseeko()` will fail if:

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* 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' file size limit.

EFBIG

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

whence is an invalid argument. The resulting file-position indicator would be set to a negative value.

EIO

A physical input/output error has occurred, or the process is a member of a background process group attempting to perform a `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 is no free space on the device.

EOVERFLOW

For `fseek()`, the resulting file offset would be a value which cannot be represented correctly in an object of type `long`.

EOVERFLOW

For `fseeko()`, the current file offset cannot be represented correctly in an object of type `off_t`.

EPIPE

The file descriptor underlying the *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 be sent to the process.

ENXIO

The device does not exist or cannot be accessed.

Function rewind()

Refer to `fseek()` with the exception of `EINVAL` which does not apply.

Function ftell() and ftello()

`ftell()` and `ftello()` will fail if:

EBADF

The file descriptor underlying *stream* is not an open file descriptor.

EOVERFLOW

For `ftell()`, the current file offset cannot be represented correctly in an object of type `long`.

EOVERFLOW

For `ftello()`, the current file offset cannot be represented correctly in an object of type `off_t`.

ESPIPE

The file descriptor underlying *stream* is associated with a pipe or FIFO.

NOTES

Although on the Reliant UNIX system an offset returned by `ftell()` is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non Reliant UNIX systems requires that a direct offset is used by `fseek()`. Arithmetic operations cannot always be meaningfully performed on a non Reliant UNIX offset, which is not necessarily measured in bytes.

Applications which use the `fseeko()` and `ftello()` interfaces should define `_LARGEFILE_SOURCE` to be 1, then include `<unistd.h>` and then test that `_LFS_LARGEFILE` is 1 to determine if the additional functionality is indeed available. This additional functionality may be available even when `_LARGEFILE_SOURCE` is not defined, but it will not be available to strictly conforming X/Open programs.

SEE ALSO

`lseek(2)`, `write(2)`, `fopen(3S)`, `popen(3S)`, `stdio(3S)`, `ungetc(3S)`, `unistd(4)`, `lfs(5)`, `stdio(5)`.

fprintf(3S)**NAME**

`fprintf`, `wprintf`, `swprintf`, `vfprintf`, `vwprintf`, `vswprintf` - formatted wide-character output

SYNOPSIS

```
cc [ flag ... ] file ... -lw [ library ... ]
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int fprintf(FILE *stream, const wchar_t *format, ...);
int wprintf(const wchar_t *format, ...);
int swprintf(wchar_t *s, size_t n, const wchar_t *format, ...);
int vfprintf(FILE *stream, const wchar_t *format, va_list arg);
int vwprintf(const wchar_t *format, va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);
```

DESCRIPTION

The `fprintf()` function writes output to the stream pointed to by *stream*, under control of the wide string pointed to by *format* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The `fprintf()` function returns when the end of the format string is encountered.

The `wprintf()` function is equivalent to `fprintf()` with the argument `stdout` interposed before the arguments to `wprintf()`.

The `swprintf()` function is equivalent to `fprintf()`, except that the argument *s* specifies an array of wide-characters into which the generated output is to be written, rather than written to a stream. No more than *n* wide-characters are written, including a terminating null wide-character, which is always added (unless *n* is zero).

The `vfprintf()` function is equivalent to `fprintf()`, with the variable argument list replaced by *arg*, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfprintf()` function does not invoke the `va_end` macro.

Note:

As the functions `vfprintf()`, `vswprintf()`, and `vwprintf()` invoke the `va_arg` macro, the value of *arg* after the return is unspecified.

The `vwprintf()` function is equivalent to `swprintf()`, with the variable argument list replaced by *arg*, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vwprintf()` function does not invoke the `va_end` macro.

The `vswprintf()` function is equivalent to `swprintf()`, with the variable argument list replaced by *arg*, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vswprintf()` function does not invoke the `va_end` macro.

Format

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, `INTL_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 decimal-point character. The decimal-point character is defined by the program's locale (`LC_NUMERIC` [see `setlocale(3C)`]). In the C locale, or in a locale where the decimal-point character is not defined, the decimal-point 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) that 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 is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk * (described later) or a decimal integer. (Note that 0 is taken as a flag, not as the beginning of a field width.)
- 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 decimal-point characters for the `e`, `E` and `f` conversions, the maximum number of significant digits for the `g` and `G` conversion or the maximum number of wide-characters to be written from a string in `s` conversion. The precision takes the form of a period (.) followed either by an asterisk * (described later) or by an optional decimal integer, if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- A wide-character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk *. In this case, an `int` argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a – flag followed by a positive field width. A negative precision argument 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, `INTL_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.

Flags

The wide-character flags and their meanings are:

- ' The integer portion of the result of a decimal conversion (%*i*, %*d*, %*u*, %*f*, %*g* or %*G*) will be formatted with thousands' grouping wide-characters. For other conversions the behaviour is undefined. The non-monetary grouping wide-character is used.

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)
 - + The result of a signed conversion always begins with a plus or minus sign. (It begins 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 is prefixed to the result. If the *space* and + flags both appear, the *space* flag is ignored.
- # The result is to be converted to an "alternate form". For o conversion, it increases the precision to force the first digit of the result to be a zero, if necessary. For x (or X) conversion, a nonzero result has 0x (or 0X) prefixed to it. For e, E, f, g, and G conversions, the result always contains a decimal-point wide-character, even if no digits follow it. (Normally, a decimal-point wide-character appears in the result of these conversions only if a digit follows its.) For g and G conversions, trailing zeros are *not* to be removed from the result. For other conversion, the behavior is undefined.
- 0 For d, i, o, u, x, X, e, E, 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 is ignored. For d, i, o, u, x, and X conversion, if a precision is specified, the 0 flag is ignored. For other conversion, the behavior is undefined.

Conversion specifiers

The conversion specifiers and their meanings are:

c, lc

An argument of type `int` is converted to an `wchar_t`, and the resulting character is printed.

If no `l` qualifier is present, the `int` argument is converted to a wide-character as if by calling `btowc()` and the resulting wide-character is written. Otherwise, the `wint_t` argument is converted to `wchar_t` and written.

C Same as `lc`.

d, i, o, u, x, X

An `int` type argument is converted to signed decimal (`d` or `i`), to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` and `X`). The `x` conversion uses the letters `abcdef` and the `X` conversion uses the letters `ABCDEF`. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide-characters.

e, E The `double` argument is converted in the style `[-]d.ddde±dd`, where there is one digit before the decimal-point wide-character (which is nonzero if the argument is nonzero) 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 zero and the `#` flag is not specified, no decimal-point wide-character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

The `fwprintf()` family of functions may make available character string representations for infinity and NaN.

f The `double` argument is converted to decimal notation in the style `[-]ddd.ddd`, where the number of digits after the decimal-point wide-character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point wide-character appears. If a decimal-point wide-character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

The `fwprintf()` family of functions may make available character string representations for infinity and NaN.

g, G The double argument is converted in style `f` or `e` (or in style `E` in the `G` conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style `e` (or `E`) is 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 decimal-point wide-character appears only if it is followed by a digit.

The `fwprintf()` family of functions may make available character string representations for infinity and NaN.

hd, hi, ho, hu, hx, hX

Specifies that the corresponding conversion specifier applies to a short int or unsigned short int argument. The argument will be supported according to the C rules for value conversions, and its value converted to short int or unsigned short int before printing.

hn Specifies that the corresponding conversion specifier applies to a pointer to a short int argument.

ld, li, lo, lu, lx, lX

Specifies that the corresponding conversion specifier applies to a long int or unsigned long int argument.

lld, llx

Specifies that the corresponding conversion specifier applies to a long long int or unsigned long long int argument.

ln Specifies that the corresponding conversion specifier applies to a pointer to long int argument.

Le, LE, Lf, Lg, LG

Specifies that the corresponding conversion specifier applies to a long double argument.

Note:

If an `h`, `l`, or `L` modifier appears before a conversion specifiers other than those listed above (with the exception of `ls`; see below), the behavior is undefined.

n The argument shall be a pointer to an integer into which is written the number of wide-characters written to the output stream so far by this call to one of the `fwprintf()` functions. No argument is converted.

p The argument shall be a pointer to `void`. The value of the pointer is converted to a sequence of printable wide-characters, in an implementation-defined manner.

s, ls

If no `l` qualifier is present, the argument shall be a pointer to a character array containing a multibyte sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the `mbrtowc()` function, with the conversion state described by an `mstate_t` object initialized to zero before the first multibyte 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 converted array, the converted array shall contain a null wide-character.

If an `l` qualifier is present, the argument shall be a pointer to an array of `wchar_t` type. Wide-characters from the array are written up to (but not including) a 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 shall contain a null wide-character.

S Same as `ls`.

% A % wide-character is written. No argument is converted. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field. If the result of a conversion is wider than the field width, the field is expanded to contain the conversion result. Characters generated by `fwprintf()` and `wprintf()` are printed as if `fputwc()` 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()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

RETURN VALUES

The `fwprintf()`, `wprintf()`, `vfwprintf()` and `vwprintf()` functions return the number of wide-characters transmitted, or a negative value if an output error occurred.

The `swprintf()` and `vswprintf()` functions return the number of wide-characters written in the array, not counting the terminating null wide-character, or a negative value if *n* or more wide-characters were requested to be written.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`fwprintf()` and `wprintf()` will fail [cf. [fputwc\(3S\)](#)] 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 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 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. 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 process.

`fwprintf()` and `wprintf()` may fail [cf. [fputwc\(3S\)](#)] 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.

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.

EXAMPLES

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 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 string:

```
L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the message:

```
Sonntag, 3. Juli, 10:02
```

To print π to 5 decimal places:

```
fwprintf(stdout, l"pi = %.5f\n", 4 * atan(1.0));
```

NOTES

If the application calling `printf()` has any objects of type `wint_t` or `wchar_t`, it must also include the header `<wchar.h>` to have these objects defined.

The minimum value for the maximum number of wide-characters produced by any single conversion shall be 509.

SEE ALSO

`btowc(3C)`, `mbrtowc(3C)`, `printf(3S)`, `vprintf(3S)`, `stdarg(5)`, `stdio(5)`, `wchar(5)`.

fwscanf(3S)**NAME**

`fwscanf`, `wscanf`, `swscanf` - formatted wide-character input

SYNOPSIS

```
cc [ flag ... ] file ... -lw [ library ... ]
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE *stream, const wchar_t *format, ...);
int wscanf(const wchar_t *format, ...);
int swscanf(const wchar_t *s, const wchar_t *format, ...);
```

DESCRIPTION

The `fwscanf()` function reads input from the stream pointed to by *stream*, under control of the wide string pointed to by *format* that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The `wscanf()` function is equivalent to `fwscanf()` with the argument `stdin` interposed before the arguments to `wscanf()`.

The `swscanf()` function is equivalent to `fwscanf()`, except that the argument *s* specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for `fwscanf()` function.

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` [see [setlocale\(3C\)](#)]). In the C locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

Format

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 wide-character *.
- An optional nonzero decimal integer that specifies the maximum field width (in wide-character).
- A wide-character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The `fwscanf()` function executes each directive of the format in turn. If a directive fails, as detailed below, the `fwscanf()` function returns. Failures are described as input failures (if an encoding error occurs or due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space wide-character(s) is executed by reading input up to the first non-white-space wide-character (which remains unread), or until no more wide-characters can be read.

A directive that is an ordinary wide-character is executed by reading the next wide-character of the stream. If the wide-character differs from the directive, 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 specifier. A conversion specification is executed in the following steps.

Input white-space wide-characters (as specified by the `iswspace()` function) are skipped, unless the specification includes a `c` or `n` specifier. These white-space wide-characters are not counted against a specified field width.

An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest sequence of input wide-characters, not exceeding any specified field width, which is, or is a prefix of a matching sequence. The first wide-character, if any, after input item remains unread. If the length of the input item is zero, the execution of the directive 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 `%` specifier, the input item (or, in the case of a `%n` directive, the count of input wide-characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive 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.

Conversion specifiers

The following conversion specifiers are valid:

`c, lc`

Matches a sequence of wide-characters of the number specified by the field width (1 if no field width is present in the directive). If no `l` qualifier is present, characters from the input field are converted as if by repeated calls to the `wcrtomb()` function, with the shift state described by an `mbstate_t` object initialized to zero before the first wide-character is converted. The corresponding argument shall be a pointer to a character array large enough to accept the sequence. No null character is added.

If an `l` qualifier is present, the corresponding argument shall be a pointer to the initial element of an array of `wchar_t` type large enough to accept the sequence. No null wide-character is added.

`C` Same as `lc`.

`d, i, o, u, x, X`

An `int` type argument is converted to signed decimal (`d` or `i`), to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` and `X`). The `x` conversion uses the letters `abcdef` and the `X` conversion uses the letters `ABCDEF`. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide-characters.

`e, E, f, g, G`

Matches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of the `wcstol()` function. The corresponding argument shall be a pointer to floating.

If the `fwprintf()` family of functions generates character string representations for infinity and NaN (a 7858 symbolic entity encoded in floating-point format) to support the ANSI/IEEE Std 754:1985 standard, the `fwscanf()` family of functions will recognise them as input.

hd, hi, ho, hu, hx, hX

Specifies that the corresponding conversion specifier applies to a `short int` or unsigned `short int` argument. The argument will be supported according to the C rules for value conversions, and its value converted to `short int` or unsigned `short int` before printing.

ld, li, lo, lu, lx, lX

Specifies that the corresponding conversion specifier applies to a `long int` or unsigned `long int` argument.

Le, LE, Lf, Lg, LG

Specifies that the corresponding conversion specifier applies to a `long double` argument.

Note:

If an `h`, `l`, or `L` modifier appears before a conversion specifiers other than those listed above (with the exception of `ls`; see below), the behavior is undefined.

n No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of wide-characters read from the input stream so far by this call to the `fwscanf()` function. Execution of an `%n` directive does not affect the assignment count returned at the completion of execution of the `fwscanf()` function.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the `%p` conversion of the `fwprintf()` function. The corresponding argument shall be a pointer to a pointer to void. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the `%p` conversion is undefined.

s, ls

Matches a sequence of non-white-space wide-characters. If no `l` qualifier is present, 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 shall be a pointer to a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

Otherwise, the corresponding argument shall be a pointer to the initial element of an array of `wchar_t` type large enough to accept the sequence and a terminating null wide-character, which will be added automatically.

S Same as `ls`.

[

Matches a nonempty sequence of wide-characters from a set of expected characters (the *scanset*). If no `l` qualifier is present, 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 shall be a pointer to a character array large enough to accept the sequence and a terminating null character, which will be added automatically. If an `l` qualifier is present, the corresponding argument shall be a pointer to the initial element of an array of `wchar_t` type large enough to accept the sequence and a terminating null wide-character, which will be added automatically.

The conversion specifier includes all subsequent wide-characters in the format string, up to and including the matching right bracket wide-character (`]`). The wide-characters between the brackets (the *scanlist*) comprise the *scanset*, unless the wide-character after the left 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 bracket. If the conversion specifier begins with `[ ]` or `[^]`, the right bracket wide-character is in the *scanlist* and the next right bracket wide-character is the matching right bracket that ends the specification; otherwise the first right bracket wide-character is the one that ends the specification. If a `-` wide-character is in the *scanlist* and is not the first, nor the second where the first wide-character is a circumflex, nor the last character, the behavior is implementation-defined.

% Matches a single `%`: no conversion or assignment occurs. The complete conversion specification shall

be %%.

If a conversion specification is invalid, the behavior is undefined.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide-characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (other than %n, 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()`, `fgetwc()`, `fgets()`, `fgetws()`, `fread()`, `getc()`, `getwc()`, `getchar()`, `getwchar()`, `gets()`, `fscanf()` or `fwscanf()` using *stream* that returns data not supplied by a prior call to `ungetc()`.

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

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

These 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 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.

These 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

Input byte sequence does not form a valid character.

EINVAL

There are insufficient arguments.

EXAMPLES

1) The call:

```
int n, i; float x; wchar_t name[50];  
n = fwscanf(stdin, L"%d%f%ls", &i, &x, name);
```

with the input line:

```
25 54.32E-1 Thompson
```

will assign to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and to `name` the sequence `Thompson\0`.

2) The call:

```
int i; float x; double y;  
fwscanf(stdin, L"%2d%f*d %lf", &i, &x, &y);
```

with input:

```
56789 0123 56a72
```

will assign to `i` the value 56 and to `x` the value 789.0, will skip past 0123, and will assign to `y` the value 56.0. The next wide-character read from the input stream will be `a`.

SEE ALSO

[setlocale\(3C\)](#), [wrtomb\(3C\)](#), [wcstod\(3C\)](#), [wcstol\(3C\)](#), [wcstoul\(3C\)](#), [fgetwc\(3S\)](#), [fwprintf\(3S\)](#), [stdio\(5\)](#), [wchar\(5\)](#).

NAME

getc, getchar, fgetc, getw - get character or word from a stream

SYNOPSIS

```
#include <stdio.h>
□
□
int getc(FILE *stream);
int getchar(void);
int fgetc(FILE *stream);
int getw(FILE *stream); □
```

DESCRIPTION

getc() returns the next character (i.e., byte) from the named input *stream* as an unsigned char converted to an int. It also moves the file pointer, if defined, ahead one character in *stream*.

getchar() is defined as getc(stdin). getc() and getchar() are macros.

fgetc() behaves like getc(), but is a function rather than a macro. fgetc() runs more slowly than getc(), but it takes less space per invocation and its name can be passed as an argument to a function.

getw() returns the next word (i.e., integer) from the named input *stream*. getw() increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an int type number, and varies from machine to machine. getw() assumes no special alignment in the file.

The getw() interface need not be thread-safe.

These 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(), fgets(), fgetwc(), fgetws(), fread(), getc(), getchar(), gets(), fscanf() or scanf() using *stream* that returns data not supplied by a prior call to ungetc() or ungetwc().

RETURN VALUE

These functions return the constant EOF at end-of-file or upon an error and set the EOF or error indicator of *stream*, respectively. Because EOF is a valid integer, ferror() or feof() should be used to detect errors.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The functions fail if:

EAGAIN

The O_NONBLOCK flag is set for the file descriptor underlying *stream*, and the process is delayed.

EBADF

The file descriptor underlying *stream* is not a valid file descriptor opened for reading.

EINTR

The read operation was terminated due to the receipt of a signal; 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 SIGTIN 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 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.

NOTES

If the integer value returned by `getc()`, `getchar()`, or `fgetc()` is stored into a `char` type variable and then compared against the integer constant `EOF`, the comparison may never succeed, because sign-extension in transferring from `char` to `int` is implementation dependent.

The macro version of `getc()` evaluates a *stream* argument more than once and may treat side effects incorrectly. In particular, `getc(*f++)` does not work sensibly. Use `fgetc()` instead.

Because of possible differences in word length and byte ordering, files written using `putw()` are implementation dependent, and may not be read using `getw()` on a different processor.

Functions exist for all the above-defined macros. To get the function form, the macro name must be undefined (e.g., `#undef getc()`).

SEE ALSO

`fclose(3S)`, `ferror(3S)`, `fopen(3S)`, `fread(3S)`, `gets(3S)`, `putc(3S)`, `scanf(3S)`, `stdio(3S)`, `ungetc(3S)`, `lfs(5)`, `stdio(5)`.

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

`gets()` reads bytes from standard input `stdin` into the array pointed to by `s`, until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

`fgets()` reads 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 character.

When using `gets()`, if the length of an input line exceeds the size of `s`, indeterminate behavior may result. `fgets()` should therefore be used instead of `gets()`.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The functions will fail if:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the 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 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.

RESULT

If end-of-file is encountered and no bytes have been read, no bytes are transferred to `s` and a null pointer is returned. 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.

SEE ALSO

`lseek(2)`, `read(2)`, `ferror(3S)`, `fopen(3S)`, `fread(3S)`, `getc(3S)`, `scanf(3S)`, `stdio(3S)`,
`ungetc(3S)`, `lfs(5)`, `stdio(5)`.

popen(3S)**NAME**

`popen`, `pclose` - open/close a pipe to a command

SYNOPSIS

```
#include <stdio.h>
□
□
FILE *popen(const char *command, const char *mode);
int pclose(FILE *stream); □
```

DESCRIPTION

`popen()` creates a pipe between the calling program and the command to be executed. The arguments to `popen()` are pointers to null-terminated strings. *command* consists of a shell command line. *mode* is an I/O mode, either *r* for reading or *w* for writing. The value returned is a stream pointer such that in the I/O mode *w* one can write to the standard input of the command, by writing to the file *stream*; and one can read from the standard output of the command, if the I/O mode is *r*, by reading from the file *stream*.

A pipe opened by `popen()` should be closed by `pclose()`, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type *r* command may be used as an input filter and a type *w* as an output filter.

The behavior of `popen()` is specified only for values of *mode* of *r* and *w*. Other values such as *rb* or *wb* might be supported by specific implementations, but these would not be portable features. Note that historical implementations of `popen()` only check to see if the first character of *mode* is *r*. Thus a *mode* of "robert the robot" would be treated as read mode *r*, and a *mode* of "anything else" as write mode *w*.

EXAMPLE

```
#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);
    return 0;
}
```

This program will print on the standard output [see [stdio\(3S\)](#)] all the filenames in the current directory that have a *.c* suffix.

RESULT

`popen()` returns a null pointer if files or processes cannot be created.

`pclose()` returns `-1` if *stream* is not associated with a `popen()` command.

NOTES

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, e.g., with `fflush` [see [fclose\(3S\)](#)].

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"`.

SEE ALSO

`pipe(2)`, `wait(2)`, `fclose(3S)`, `fopen(3S)`, `stdio(3S)`, `system(3S)`.

NAME

printf, fprintf, sprintf, snprintf - printformatted output

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, .../* args */);
int fprintf(FILE *stream, const char *format, .../* args */);
int sprintf(char *s, const char *format, .../* args */);
int snprintf(char *s, size_t n, const char *format, .../* args */);□
```

DESCRIPTION

printf() writes on the standard output stream stdout.

fprintf() writes on the specified 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 space is available.

snprintf() is identical to sprintf() with the addition of the *n* argument, which states the size of the buffer referred to by *s*.

Each function returns the number of characters transmitted (not including the \0 in the case of sprintf()) or a negative value if an output error was encountered.

These functions convert, format, and print their *args* in the specified *format*. The *format* is a character string that can contain three types of objects:

1. Plain characters that are simply copied to the output stream;
2. Escape sequences that represent non-graphic characters;
3. Conversion specifications.

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 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 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 decimal-point character. The decimal-point character is defined by the program's locale (LC_NUMERIC [see [setlocale\(3C\)](#)]). In the C locale, or in a locale where the decimal-point character is not defined, the decimal-point character defaults to a period.

Escape sequences

The following escape sequences produce the associated action on devices capable of the action:

- \a Alert. Outputs a tone (BEL character).
- \b Backspace. Move the printing position to one character before the current position, unless the current position is the start of a line (BS character).
- \f Form feed. Move the printing position to the initial printing position of the next logical page (FF character).

`\n` Newline. Move the printing position to the start of the next line (NL character).
`\r` Return. Move the printing position to the start of the current line (CR character).
`\t` Horizontal tab. Move the printing position to the next horizontal tab position on the current line (TAB character).
`\v` Vertical tab. Move the printing position to the next vertical tab position (VT character).
`\?` Outputs question mark.
`\\` Outputs backslash.
`\'` Outputs apostrophe.
`\"` Outputs inverted commas.
`\octal_code`
 Character entered in octal code is output in ASCII code.
`\xhexadecimal_code`
 Character entered in hexadecimal code is output in ASCII code.
 Example:
`\100` (oct.) or `\x40` (hex.) both give the same @ character in ASCII code.

Conversion specifications

Each conversion specification is introduced by the `%` character or by the 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 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.
- An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, or `X` conversions (the field is padded with leading zeros), the number of digits to appear after the decimal-point character for the `e`, `E`, and `f` conversions, the maximum number of significant digits for the `g` and `G` conversions, or the maximum number of characters to be printed from a string in `s` and `S` conversion. The precision takes the form of a period followed by a decimal number string; a null digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.
- A *conversion character* that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (`*`) or a decimal digit string. In this case, an integer argument supplies the field width or precision. The argument that is actually converted is not fetched until the conversion letter is seen, so any arguments which specify field width or precision must appear before the argument to be converted. If the precision argument is negative, it will be changed to zero. A negative field width argument is taken as a `–` flag, followed by a positive field width.

In format strings containing the `%digits$` form of a conversion specification, a field width or precision may also be indicated by the sequence `*digits$`, where *digits* 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);
```

When numbered argument specifications are used, specifying the *n*th argument requires that all the preceding arguments, from the first to the (*n*–1)th, be specified in the format string.

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.

Flags

- ' The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%g` or `%G`) will be formatted with thousands' grouping character. For other conversions the behavior is undefined.
- The result of the conversion will be left-justified within the field. (It will be right-justified if this flag is not specified.)
- + The result of a signed conversion will always begin with a sign (+ or –).
- <blank> If the first character of a signed conversion is not a sign, a space will be placed before the result. This means that the space will be ignored if it is specified with the + character.
- # 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 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 `e`, `E`, `f`, `g`, and `G` conversions, the result will always contain a decimal-point character, even if no digits follow the point (normally, a decimal point 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`, `e`, `E`, `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. For other conversions, the behavior is undefined.

Conversion characters

Each conversion character 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.

`c`, `lc`

An argument of type `int` is converted to an unsigned `char`, and the resulting character is printed.

If no `l` qualifier is present, the `int` argument is converted to an unassigned `char`, and the resulting character is written. Otherwise, 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 `wchar_t`, the first element containing the `wint_t` argument to the `lc` conversion specification and the second a null wide-character.

`C` Same as `lc`.

`d`, `i`, `o`, `u`, `x`, `X`

An `int` type argument is converted to signed decimal (`d` or `i`), to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` and `X`). The `x` conversion uses the letters `abcdef` and the `X` conversion uses the letters `ABCDEF`. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`e`, `E` An argument of type `double` is converted to decimal notation in style `[-]d.ddde±dd`, where there is one digit before the decimal-point character (which is nonzero if the argument is non-zero) and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The `E` conversion character will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. The value is rounded to the appropriate number of digits.

The `printf()` family of functions may make available character string representations for infinity and NaN.

`f` An argument of type `double` is converted to decimal notation in style `[-]ddd.ddd`, where the number of digits after the decimal-point character [see [setlocale\(3C\)](#)] is equal to the precision specification. If no precision is specified, six digits are output; if the precision is explicitly zero and the `#` flag is not

specified, no decimal-point character appears. If a decimal-point appears, at least 1 digit appears before it. The value is rounded to the appropriate number of digits.

The `printf()` family of functions may make available character string representations for infinity and NaN.

- g, G** An argument of type `double` is printed in style `f` or `e` (or in style `E` in the case of a `G` conversion character), with the precision specifying the number of significant digits. If the precision is zero, a value of 1 is assumed. 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 decimal-point character appears only if it is followed by a digit.

The `printf()` family of functions may make available character string representations for infinity and NaN.

hd, hi, ho, hu, hx, hX

Specifies that the corresponding conversion specifier applies to a `short int` or unsigned `short int` argument. The argument will be supported according to the C rules for value conversions, and its value converted to `short int` or unsigned `short int` before printing.

hn Specifies that the corresponding conversion specifier applies to a pointer to a `short int` argument.

ld, li, lo, lu, lx, lX

Specifies that the corresponding conversion specifier applies to a `long int` or unsigned `long int` argument.

lld, llx

Specifies that the corresponding conversion specifier applies to a `long long int` or unsigned `long long int` argument.

ln Specifies that the corresponding conversion specifier applies to a pointer to `long int` argument.

Le, LE, Lf, Lg, LG

Specifies that the corresponding conversion specifier applies to a `long double` argument.

Note:

If an `h`, `l`, or `L` modifier appears before a conversion specifiers other than those listed above (with the exception of `ls`; see below), the behavior is undefined.

n The argument should be a pointer to an `int`, into which is written the number of characters written to the output standard I/O stream so far by this call to one of the `printf()` functions. No argument is converted.

p The argument is a pointer to `void`. The value of the pointer is converted to an implementation-defined 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()` function.

s, ls

The argument is a character pointer to `char`, and characters from the string are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified, it is taken to be infinite, so all characters up to the first null character are printed. A `NULL` value for an argument will yield undefined results.

If an `l` qualifier is present, the argument shall be a pointer to an array of `wchar_t` type.

Wide-characters from the array are converted to multibyte characters (each as if by a call to the `wcrtomb()` function, with the conversion state describe 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 multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall 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 shall contain a null wide-character if, to equal the multibyte 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 multibyte character written. Redundant shift sequences may result if multibyte characters

have a state-dependent encoding.

S Same as `ls`.

% Print a %; no argument is converted. The entire conversion specification must be `%%`.

If the character after the `%` or `%digits$` sequence is not a valid conversion character, the results of the conversion are 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 `fputc()` routine 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()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

RETURN VALUE

Upon successful completion, these functions return the number of bytes transmitted excluding the terminating null in the case of `sprintf()` or `snprintf()` or a negative value if an output error was encountered.

If the value of *n* is zero on a call to `snprintf()`, an unspecified value less than 1 is returned.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

`printf()` and `fprintf()` will fail [cf. [putc\(3S\)](#)] 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 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 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. 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 process.

`printf()` and `fprintf()` may fail [cf. [putc\(3S\)](#)] 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.

In addition, all forms of `printf()` may fail if:

EILSEQ

A wide-character code that does not correspond to a valid character has been detected.

EINVAL

There are insufficient arguments.

EXAMPLES

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
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

NOTES

If the application calling `printf()` has any objects of type `wint_t` or `wchar_t`, it must also include the header `<wchar.h>` to have these objects defined.

Flag `'` and conversions `%C` and `%S` are recognized from XPG4 mode onwards.

SEE ALSO

`exit(2)`, `lseek(2)`, `write(2)`, `abort(3C)`, `ecvt(3C)`, `setlocale(3C)`, `fwprintf(3S)`, `putc(3S)`, `scanf(3S)`, `sprintf(3S)`, `vprintf(3S)`, `stdio(3S)`, `lfs(5)`, `stdio(5)`.

NAME

putc, putchar, fputc, putw - output character/word on a stream

SYNOPSIS

```
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
int fputc(int c, FILE *stream);
int putw(int w, FILE *stream); □
```

DESCRIPTION

putc() writes the byte *c* (converted to an unsigned char) onto the output *stream* at the position where the file pointer (if defined) is pointing, and advances the file pointer appropriately. If the file cannot support positioning requests, or *stream* was opened with append mode, the byte is written at the end of the output. putchar(*c*) is defined as putc(*c*, □stdout). puts() and putchar() are macros.

fputc() behaves like putc(), but is a function rather than a macro. fputc() runs more slowly than putc(), but it takes less space per invocation and its name can be passed as an argument to a function.

putw() writes the word (int) *w* to the output *stream* (where the file pointer, if defined, is pointing). The size of a word is the size of an int and varies from machine to machine. putw() neither assumes nor causes special alignment in the file.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

These functions will fail if either the stream is unbuffered or the stream's buffer needed to be flushed and:

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 *stream* is not a valid 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 input/output error has occurred, 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.

EPIPE

An attempt was made to write a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal will be sent to the process.

The fputc() 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.

RESULT

On success, these functions (with the exception of `putw()`) each return the value they have written. `putw()` returns `ferror(stream)`. On failure, they return the constant `EOF`. This result will occur, for example, if the file *stream* is not open for writing or if the output file cannot grow.

NOTES

Because it is implemented as a macro, `putc()` can evaluate *stream* more than once. If *stream* is an expression with side-effects, it is not treated correctly by `putc()`. In particular, `putc(c, &*f++)` does not behave as expected. The `fputc()` function should be used instead.

Because of possible differences in word length and byte ordering, files written using `putw()` are machine-dependent, and may not be read using `getw()` on a different processor.

Functions exist for all the above defined macros. To get the function form, the macro name must be undefined (e.g., `#undef putc()`).

SEE ALSO

`exit(2)`, `lseek(2)`, `write(2)`, `abort(3C)`, `fclose(3S)`, `ferror(3S)`, `fopen(3S)`, `fread(3S)`, `printf(3S)`, `puts(3S)`, `setbuf(3S)`, `stdio(3S)`, `lfs(5)`, `stdio(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

`puts()` writes the string pointed to by *s*, followed by a new-line character, to the standard output stream `stdout`.

`fputs()` writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

These functions will fail if either the stream is unbuffered or the stream's buffer needed to be flushed and:

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 *stream* is not a valid descriptor open for writing.

EFBIG

An attempt was made to write to a file that exceeds the maximum file size.

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 input/output error has occurred.

ENOSPC

There was no free space remaining on the device.

EPIPE

An attempt was made to write a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will be sent to the process.

RESULT

On success both routines return the number of characters written; otherwise they return `EOF`.

NOTES

`puts()` appends a new-line character while `fputs()` does not.

SEE ALSO

[exit\(2\)](#), [lseek\(2\)](#), [write\(2\)](#), [abort\(3C\)](#), [fclose\(3S\)](#), [ferror\(3S\)](#), [fopen\(3S\)](#), [fread\(3S\)](#), [printf\(3S\)](#), [putc\(3S\)](#), [stdio\(3S\)](#), [lfs\(5\)](#), [stdio\(5\)](#).

NAME

`scanf`, `fscanf`, `sscanf` - formatted input

SYNOPSIS

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

DESCRIPTION

`scanf()` reads from the standard input stream, `stdin`.

`fscanf()` reads from the specified input stream, `stream`.

`sscanf()` reads from the character string `s`.

Each of these functions reads bytes, interprets them according to a specified format, and stores the results in its arguments. Each expects, as arguments, a control string *format* and a set of pointer arguments indicating where the converted input should be stored. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

A conversion specification in *format* directs the conversion to the next input field; the result is placed in the variable pointed to by the corresponding argument. The assignment can be suppressed with `*` in order to skip a field. An input field is defined as a string of non-spaced characters; it extends to the next inappropriate character or until the maximum field width, if one is specified.

Conversions can be applied to the *n*th argument in the argument list, rather than to the next unused argument. In this case, the conversion character `%` is replaced by the sequence `%n$` where *n* is a decimal integer in the range `[1, INTL_ARGMAX]`. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages. In format strings containing the `%n$` 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 control string can contain either form of a conversion specification, i.e., `%` or `%n$`, although the two forms cannot be mixed within a single control string. Only the forms `%%` and `%*` can be mixed with the form `%n$`.

The control string *format* usually contains conversion specifications, which are used to direct interpretation of input sequences. *format* comprises the following characters:

- White-space characters (blanks, tabs, new-lines or form-feeds).
- Normal characters (not `%` and not white-space characters) cause the next character of the input to be read and compared with the byte from *format*. If they do not match, this and the subsequent bytes remain unread.
- Conversion specifications

These begin with the character `%` or with the string `%n$`. Then the following are specified in the given order:

1. `*` to suppress assignment (optional)
2. a nonzero decimal number, specifying the maximum field size (optional)
3. `h`, `l` or `L` to indicate the size of the expected object (optional)
4. conversion codes to specify the conversion to be performed (see explanation below)

A conversion specification is performed in the following steps: white-space characters (as specified by the `isspace()` function) are ignored, unless the conversion specification contains the conversion code `[%c, %C]` or `%n`. If the specification does not contain `%n`, an input item is read. An input item is defined as the longest

sequence of input bytes (up to a maximum item size, calculated in characters or bytes depending on the conversion code) that is the initial subsequence of a matching sequence. The first byte, if any, after the input item remains unread. If the length of the input item is zero, the execution of the conversion specification fails. The input item is converted to a type appropriate to the conversion code. If the input item is not a matching sequence, the execution of the specification fails. Unless assignment suppression was indicated by ***, the result of the conversion is placed in the variable pointed to by the first argument following *format*. 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.

Conversion characters

The conversion characters are evaluated one after the other. If a character cannot be evaluated, the function returns with an error. The following characters are valid:

c, lc

Reads a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument must be a pointer to the initial byte of an array of type `char`, `signed char` or `unsigned char` large enough to accept the sequence. No null byte is added. The normal skip over white space is suppressed.

If an `l` qualifier is present, the input shall be a sequence of multibyte character conversions in the initial shift state. Each multibyte character 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 multibyte character is converted. The corresponding argument shall 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.

C Reads a sequence of characters of the number specified by the field width (1 if no field width is present in the specification). The sequence is converted to a sequence of wide-character codes in the same manner as the `mbstowcs()` function. The corresponding argument must be a pointer to the initial byte of an array of type `wchar_t` large enough to accept the sequence which is the result of the conversion. No null byte is added.

If the sequence begins with the initial shift state, the conversion is the same as expected for the `mbwstowcs()` function; otherwise the behavior of the conversion is undefined. The normal skip over white-space characters is suppressed in this case.

This conversion code is supported from XPG4 mode onwards.

d Reads an optionally signed decimal integer, which has the same format as the subject sequence of the `strtol()` function with the value 10 for the *base* argument. The corresponding argument should be a pointer to `int`.

e, f, g

Reads an optionally signed floating point number, which has the same format as the subject string of the `strstr()` function. The corresponding argument should be a pointer to `float`.

E In ANSI mode (`cc -kansi`), **E** behaves the same as **e**. In K&R mode (`cc -kcc`), **E** behaves the same as **le**.

G In ANSI mode (`cc -kansi`), **G** behaves the same as **g**. In K&R mode (`cc -kcc`), **G** behaves the same as **lg**.

hd, hi, hn

The corresponding argument is a pointer to `short int` rather than a pointer to `int`.

ho, hu, hx

The corresponding argument is a pointer to `unsigned short int` rather than a pointer to `unsigned int`.

i Reads an optionally signed integer, which has the same format as the subject sequence of the `strtol()` function with the value 0 for the *base* argument. The corresponding argument should be a pointer to `int`.

ld, li, ln

The corresponding argument is a pointer to `long int` rather than a pointer to `int`.

lld, llx

Reads an optionally signed integer, which has the same format as the subject sequence of the `strtoll()` function with the value 0 for the *base* argument. The corresponding argument should be a pointer to `long long int` rather than a pointer to `int`.

le, lf, lg

The corresponding argument should be a pointer to `double` rather than a pointer to `float`.

lo, lu, lx

The corresponding argument should be a pointer to `unsigned long int` rather than a pointer to `unsigned int`.

Le, Lf, Lg

The corresponding argument should be a pointer to `long double`.

n

No input is processed. The corresponding argument should be a pointer to `int` into which is to be written the number of characters read from the input stream so far by the call to the function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the function.

o

Reads an optionally signed octal integer, which has the same format as the subject sequence of the `strtoul()` function with the value 8 for the *base* argument. The corresponding argument should be a pointer to `unsigned int`.

p

Reads a hexadecimal number with the value 16 for the argument *base* that has the same format as the set of sequences that may be produced by the `%p` conversion of the `fprintf()` function. The corresponding argument should be a pointer to `void`. The interpretation of the input item is implementation-defined. 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.

s

Reads a sequence without white-space characters. The corresponding argument must be a pointer to the initial byte of an array of type `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` qualifier is present, the input shall be a sequence of multibyte character conversions in the initial shift state. Each multibyte character 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 multibyte character is converted. The corresponding argument shall 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.

S

Reads a sequence without white-space characters. This sequence is converted to a sequence of wide-character codes in the same manner as the `mbstowcs()` function. The corresponding argument must be a pointer to the initial byte of an array of type `wchar_t` large enough to accept the result of the conversion and a terminating null byte which will be added automatically. If the field width is specified, it denotes the maximum number of characters to accept.

This conversion code is supported from XPG4 mode onwards.

u

Reads an optionally signed decimal integer, which has the same format as the subject sequence of the `strtoul()` function with the value 10 for the *base* argument. The corresponding argument should be a pointer to `unsigned int`.

x

Reads an optionally signed hexadecimal integer, which has the same format as the subject sequence of the `strtoul()` function with the value 16 for the *base* argument. The corresponding argument should be a pointer to `unsigned int`.

X

In ANSI mode (`cc -kansi`), `X` behaves the same as `x`. In K&R mode (`cc -kcc`), `X` behaves the same as `lx`.

[, l[

Reads a nonempty sequence of characters from a set of expected bytes. The corresponding argument should be a pointer to the initial byte of an array large enough to accept the sequence, including a terminating null byte, which will be added automatically. The conversion specification includes all subsequent bytes in the *format* string, up to and including the matching right bracket `]`. The bytes between the brackets comprise the scanset, unless the byte after the left bracket is `^`, in which case the scanset contains all bytes that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket character is in the scanlist and the closing bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is always the one that ends the specification. A range of bytes in the scanset may be represented by the construct *first* – *last*; thus `[0123456789]` may be expressed `[0-9]`. Using this convention, *first* must be lexically less than or equal to *last*, or else – will stand for itself. The character – will also stand for itself whenever it is the first or the last character in the scanlist. To include the right bracket as an element of the scanset, it must appear as the first character (possibly preceded by `^`) of the scanlist and in this case it will not be syntactically interpreted as the closing bracket. At least one character must match for this conversion to be considered successful.

If an `l` qualifier is present, the input shall be a sequence of multibyte character conversions in the initial shift state. Each multibyte character 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 multibyte character is converted. The corresponding argument shall 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.

- % A single % is expected in the input at this point; no assignment or conversion is done. The complete conversion specification must be %%.

The `h`, `l`, or `L` modifier is ignored for all conversion specifiers other than those listed.

If an invalid conversion character follows the %, the results of the operation may not be predictable.

Each function allows for detection of a language-dependent decimal point character in the input string. The decimal point character is defined by the program's locale (`LC_NUMERIC`). In the C locale, or in a locale where the decimal point character is not defined, the decimal point character defaults to a period.

The `scanf()` conversion terminates at end of file, at the end of the control string, or when an input character conflicts with the control string.

If end-of-file is reached before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure. If the end-of-file is reached during execution of one of the subsequent directives, conversion is terminated with an input failure.

Trailing white space (including new-line characters) is left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

EXAMPLES

The call to the function `scanf()`:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and `name` will contain the string `thompson\0`.

The call to the function `scanf()`:

```
int i; float x; char name[50];
(void) scanf("%2d%f%d %[0-9]", &i, &x, name);
```

with the input line:

56789 0123 56a72

will assign 56 to `i`, 789.0 to `x`, skip 0123, and place the characters 56\0 in `name`. The next character read from `stdin` will be `a`.

RETURN VALUE

These routines return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure between an input character and the control string. 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

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

The 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 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 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.

In addition, `fscanf()` may fail if:

EILSEQ

Input byte sequence does not form a valid character.

EINVAL

There are insufficient arguments.

NOTES

If the application that calls `fprintf()` contains objects of type `wchar_t`, the files `sys/types.h` and `stddef.h` must be linked.

In format strings containing the `%` form of conversion specifications, each argument in the argument list is used exactly once.

The conversion codes `c` and `s` are supported from XPG4 mode onwards.

SEE ALSO

`cc(1)`, [strtod\(3C\)](#), [strtoul\(3C\)](#), [fwscanf\(3S\)](#), [printf\(3S\)](#), [lfs\(5\)](#), [stdio\(5\)](#).

setbuf(3S)**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

`setbuf()` may be used after a stream has been 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 buffer size is not limited; however, the constant `BUFSIZ`, defined in the `stdio.h` header file, is typically a good buffer size:

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

`setvbuf()` may be used after a stream has been opened but before it is read or written. *type* determines how *stream* will be buffered. The meanings of the permissible values for *type* (defined in `stdio.h`) are:

`_IOFBF`

Full buffering of the input/output

`_IOLBF`

Output is line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.

`_IONBF`

Input/output is 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. *size* specifies the size of the buffer to be used. If input/output is unbuffered, *buf* and *size* are ignored.

RESULT

If an illegal value for *type* is provided, `setbuf()` returns a non-zero value. Otherwise, it returns zero.

NOTES

A common source of error is allocating buffer space as an `auto` variable in a code block, and then failing to close the stream in the same block.

Parts of *buf* will be used for internal bookkeeping of the stream and, therefore, *buf* will contain less than *size* bytes when full. When using `setbuf()`, the automatically allocated buffer should be used.

SEE ALSO

`malloc(3C)`, `fopen(3S)`, `getc(3S)`, `putc(3S)`, `stdio(3S)`, `stdio(5)`.

setbuffer(3S)
BSD Compatibility Package**NAME**

setbuffer, setlinebuf - assign buffering to a stream

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb
#include <stdio.h>
setbuffer(stream, buf, size)
FILE *stream;
char *buf;
int size;
setlinebuf(stream)
FILE *stream;
```

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a NEWLINE is encountered or input is read from any line buffered input stream. `fflush` may be used to force the block out early. Normally all files are block buffered. A buffer is obtained from `malloc` upon the first `getc` or `putc` on the file.

By default, output to a terminal is line buffered, except for output to the standard stream `stderr` which is unbuffered, and all other input/output is fully buffered.

`setbuffer` can be used after a stream has been opened but before it is read or written. It uses the character array `buf` whose size is determined by the `size` argument instead of an automatically allocated buffer. If `buf` is the `NULL` pointer, input/output will be completely unbuffered. A manifest constant `BUFSIZ`, defined in the `<stdio.h>` header file, tells how big an array is needed:

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

`setlinebuf` 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 file descriptor is active.

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

NOTE

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.

SEE ALSO

`malloc(3C)`, `fclose(3S)`, `getc(3S)`, `fopen(3S)`, `fread(3S)`, `printf(3S)`, `putc(3S)`, `puts(3S)`.

sprintf(3S)
BSD Compatibility Package**NAME**`sprintf, vsprintf` - formatted output conversion**SYNOPSIS**

```

/usr/ucb/cc [flag ...] file ... -lucb
#include <stdio.h>
char sprintf(char *s, *format [,arg] ...);
char *vsprintf(char *s, *format, va_list ap);

```

DESCRIPTION

`sprintf` places output, followed by the `NULL` character (`\0`), in consecutive bytes starting at `s`; it is the user's responsibility to ensure that enough storage is available.

`vsprintf` is the same as `sprintf`, except that instead of being called with a variable number of arguments, it is called with an argument list as defined by `varargs`.

Each of these functions converts, formats, and prints its arguments under control of the *format*. The *format* is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing 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 simply ignored.

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

- Zero or more flags, which modify the meaning of the conversion specification.
- An optional decimal digit string specifying a minimum "*field width*". If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag `-`, described below, has been given) to the field width. The padding is with blanks unless the field width digit string starts with a zero, in which case the padding is with zeros.
- A *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, or `X` conversions, the number of digits to appear after the decimal point for the `e`, `E`, and `f` conversions, the maximum number of significant digits for the `g` and `G` conversion, or the maximum number of characters to be printed from a string in `s` conversion. 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 `l` (ell) specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion character applies to a long integer argument. An `l` before any other conversion character is ignored.
- A character that indicates the type of conversion to be applied.

A field width or precision or both may be indicated by an asterisk (`*`) instead of a digit string. In this case, an integer argument supplies the field width or precision. The argument that is actually converted is not fetched until the conversion letter is seen, so the arguments specifying field width or precision must appear *before* the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. If the precision argument is negative, it will be changed to zero.

The flag characters and their meanings are:

- The result of the conversion will be left-justified within the field.
- + The result of a signed conversion will always begin with a sign (`+` or `-`).

blank

If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and `+` flags both appear, the blank flag will be ignored.

- # This flag specifies that the value is to be converted to an alternate form. For `c`, `d`, `i`, `s`, and `u` conversions, the flag has no effect. For `o` conversion, it increases the precision 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` prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For `g` and `G` conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

`d`, `i`, `o`, `u`, `x`, `X`

The integer argument is converted to signed decimal (`d` or `i`), unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` and `X`), respectively; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. 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 zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a `NULL` string.

`f` The float or double argument is converted to decimal notation in the style `[-]ddd.ddd` where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

`e`, `E` The float or double argument is converted in the style `[-]d.ddde±ddd`, where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The `E` format code will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits.

`g`, `G` The float or double argument is printed in style `f` or `e` (or in style `E` in the case of a `G` format code), with the precision specifying the number of significant digits. 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 the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

The `e`, `E`, `f`, `g`, and `G` formats print IEEE indeterminate values (infinity or not-a-number) as *Infinity* or *NaN* respectively.

`c` The character argument is printed.

`s` The argument is taken to be a string (character pointer) and characters from the string are printed until a `NULL` character (`\0`) is encountered or until the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first `NULL` character are printed. A `NULL` value for argument will yield undefined results.

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

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. Padding takes place only if the specified field width exceeds the actual width.

RETURN VALUE

Upon success, `sprintf` and `vsprintf` always return `s`.

EXAMPLES

The content of `buf` is a date and time in the form `Sunday, July 3, 10:02`, where `weekday` and `month` are pointers to `NULL`-terminated strings:

```
sprintf(buf, "%s, %s %i, %d:%.2d", weekday, month, day, hour, min);
```

The following routine `π` has the effect that data is stored in `buf` with a precision of five decimal places:

```
sprintf(buf, "pi = %.5f", 4 * atan(1. 0));
```

NOTES

Very wide fields (>128 characters) fail.

SEE ALSO

`putc(3S)`, `scanf(3S)`, `vprintf(3S)`, `varargs(5)`.

NAME

stdio:
 stdin, stdout, stderr - standard buffered input/output streams

SYNOPSIS

```
#include <stdio.h>
FILE *stdin, *stdout, *stderr;
```

DESCRIPTION

The functions described in section 3S 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()` and `putchar()`, and the higher-level routines `fgetc()`, `fgets()`, `fprintf()`, `fputc()`, `fputs()`, `fread()`, `fscanf()`, `fwrite()`, `gets()`, `getw()`, `printf()`, `puts()`, `putw()`, and `scanf()` 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* and is declared to be a pointer to a defined type `FILE`. `fopen()` creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the `stdio.h` header file and associated with the standard open files:

```
stdin
    Standard input file
stdout
    Standard output file
stderr
    Standard error output 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, stdin (0).
STDOUT_FILENO
    Standard output value, stdout (1).
STDERR_FILENO
    Standard error value, stderr (2).
```

A constant `NULL` designates a null pointer.

An integer-constant `EOF` (-1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant `BUFSIZ` specifies the size of the buffers used by the particular implementation.

An integer constant `FILENAME_MAX` specifies the size needed for an array of `char` large enough to hold the longest path name string that the implementation guarantees can be opened.

An 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 via `fopen`, and only file descriptors 0 through 255 are valid.

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

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

The functions and constants marked (3S) in this manual are declared in that header file 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()`, `feof()`, `clearerr()`, and `fileno()`. 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\(3S\)](#)] 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). `setbuf()` or `setbuffer()` [both described in [setbuf\(3S\)](#)] can change the stream's buffering strategy.

RESULT

Invalid file 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\(3S\)](#), [cuserid\(3S\)](#), [fclose\(3S\)](#), [ferror\(3S\)](#), [fopen\(3S\)](#), [fread\(3S\)](#), [fseek\(3S\)](#), [getc\(3S\)](#), [gets\(3S\)](#), [popen\(3S\)](#), [printf\(3S\)](#), [putc\(3S\)](#), [puts\(3S\)](#), [scanf\(3S\)](#), [setbuf\(3S\)](#), [system\(3S\)](#), [tmpfile\(3S\)](#), [tmpnam\(3S\)](#), [ungetc\(3S\)](#), [unistd\(4\)](#), [stdio\(5\)](#).

NAME

`system` - issue a shell command

SYNOPSIS

```
#include <stdlib.h>

int system(const char *command);
```

DESCRIPTION

`system()` causes the string *command* to be given to the shell [see [sh\(1\)](#)] as input, as if the string had been typed as a command at a terminal. The current process waits until the shell command has completed, then returns the exit status of the shell in the format specified by `waitpid()`.

If *command* is a null pointer, `system()` checks if `/sbin/sh` exists and is executable. If `/sbin/sh` is available, `system()` returns nonzero; otherwise it returns zero.

`system()` fails if:

EINTR

`system()` was interrupted by a signal [XPG3].

EAGAIN

Insufficient memory on the system to start another process, or the maximum number of processes executable on the system level or for one user (`CHILD_MAX`) has been reached.

RESULT

`system()` forks to create a child process that in turn execs `/sbin/sh` in order to execute *command*. If the `fork()` or `exec()` fails, `system()` returns a negative value and sets `errno`.

NOTES

If the return value of `system()` is not -1, its value can be decoded through the use of the macros described in `sys/wait.h`. For convenience, these macros are also provided in `stdlib.h`.

Note that, while `system()` must ignore `SIGINT` and `SIGQUIT` and block `SIGCHLD` while waiting for the child to terminate, the handling of signals in the executed command is as specified by `fork()` and `exec()`. For example, if `SIGINT` is being caught or is set to `SIG_DFL` when `system()` is called, then the child will be started with `SIGINT` handling set to `SIG_DFL`.

Ignoring `SIGINT` and `SIGQUIT` in the parent process in XPG3 mode prevents coordination problems (two processes reading from the same terminal, for example) when the executed command ignores or catches one of the signals. It is also usually the correct action when the user has given a command to the application to be executed synchronously (as in the `!"` command in many interactive applications). In either case, the signal should be delivered only to the child process, not to the application itself. There is one situation where ignoring the signals might have less than the desired effect. This is when the application uses `system()` to perform some task invisible to the user. If the user typed the interrupt character (`^C` for example) while `system()` is being used in this way, one would expect the application to be killed, but only the executed command will be killed. Applications that use `system()` in this way should carefully check the return status from `system()` to see if the executed command was successful, and should take appropriate action when the command fails.

Blocking `SIGCHLD` while waiting for the child to terminate prevents the application from catching the signal and obtaining status from `system()`'s child process before `system()` can get the status itself.

The context in which the utility is ultimately executed may differ from that in which the `system()` function was called. For example, file descriptors that have the `FD_CLOEXEC` flag set will be closed, and the process ID and parent process ID will be different. Also, if the executed utility changes its environment variables or its current working directory, that change will not be reflected in the caller's context.

There is no defined way for an application to find the specific path for the shell. However, `confstr()` can provide a value for `PATH` that is guaranteed to find the `sh` utility.

The shell applies restrictions to the use of "s bits": If real (uid/gid) and effective (euid/egid) user/group numbers are different and (euid/egid) is less than 100 and egid is not set to "1", (euid/egid) are assigned the same numbers as (uid/gid).

SEE ALSO

`sh(1)`, `exec(2)`, `wait(2)`.

NAME

tmpfile, tmpfile64 - create a temporary file

SYNOPSIS

```
#include <stdio.h>
FILE *tmpfile(void);
FILE *tmpfile64(void);
```

DESCRIPTION

tmpfile() creates a temporary file using a name generated by the tmpnam(3S) routine and returns a corresponding FILE pointer. If the file cannot be opened, perror() is used to issue an error message, and a null pointer is returned. The file is automatically deleted when the process using it terminates or when the file is closed. The file is opened 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.

If the process is killed in the period between file creation and unlinking, a permanent file may be left behind.

An error message may be written to standard error if the stream cannot be opened.

There is no functional difference between tmpfile() and tmpfile64(), except for the interpretation of off64_t [see lfs(5)].

RETURN VALUE

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 following error code descriptions are function-specific. You will find a general description in intro_prm2(2) or in errno(5).

The tmpfile() function will fail if:

EINTR

A signal was caught during the tmpfile() function.

EMFILE

The number of open files exceeds OPEN_MAX.

ENFILE

The system file table is full.

ENOSPC

No new file can be opened in the directory or file system.

EOVERFLOW

The file is a regular file and the size of the file cannot be represented correctly in an object of type off_t.

The tmpfile() function may fail if:

EMFILE

FOPEN_MAX streams are currently open in the calling process.

ENOMEM

Insufficient storage space is available.

SEE ALSO

```
creat(2), open(2), unlink(2), mktemp(3C), perror(3C), fopen(3S), stdio(3S), tmpnam(3S)  
, lfs(5), stdio(5).
```


tmpnam(3S)

NAME

tmpnam, tmpnam - create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>
char *tmpnam(char *s);
char *tmpnam(const char *dir, const char *pfx);
```

DESCRIPTION

The `tmpnam()` function generates a string that is a valid filename and that is not the same as the name of an existing file.

`tmpnam()` always generates a filename using the path-prefix defined as `P_tmpdir` in the `stdio.h` header file. If `s` is `NULL`, `tmpnam()` leaves its result in an internal static area and returns a pointer to that area. The next call to `tmpnam()` will destroy the contents of the area. If `s` is not `NULL`, it is assumed to be the address of an array of at least `L_tmpnam` bytes, where `L_tmpnam` is a constant defined in `stdio.h`; `tmpnam()` places its result in that array and returns `s`.

The `tmpnam()` function generates a different string each time it is called from the same process, up to `TMP_MAX` times. If it is called more than `TMP_MAX` times, the behaviour is implementation-dependent.

If the application uses any of the interfaces guaranteed to be available if either `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS` is defined, the `tmpnam()` function must be called with a non-`NULL` parameter.

The `tmpnam()` function generates a pathname that may be used for a temporary file.

`tmpnam()` allows the choice of a directory to be controlled. 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 file is used. If that directory is not accessible via `access(2)`, `/tmp` will be used as a last resort. This entire sequence can be improved by providing an environment variable `TMPDIR`, whose value is the name of the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the `pfx` argument for this. This argument may be `NULL` or point to a string of up to five bytes to be used as the first few bytes of the temporary filename.

`tmpnam()` uses `malloc()` to get space for the constructed filename, and returns a pointer to this area. Thus, any pointer value returned from `tmpnam()` may serve as an argument to `free()` [see [malloc\(3C\)](#)]. If `tmpnam()` cannot return the expected result for any reason – e.g. `malloc()` failed, or no appropriate directory could be found – a `NULL` pointer will be returned.

RETURN VALUE

Upon successful completion, `tmpnam()` returns a pointer to a string.

If the argument `s` is a null pointer, `tmpnam()` leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to `tmpnam()` may modify the same object. If the argument `s` is not a null pointer, it is presumed to point to an array of at least `L_tmpnam` bytes; `tmpnam()` writes its result in that array and returns the argument as its value.

Upon successful completion, `tmpnam()` allocates 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()`. Otherwise it returns a null pointer and sets `errno` to indicate the error.

ERRORS

No errors are defined for `tmpnam()`.

The `tmpnam()` function will fail if:

`ENOMEM`

Insufficient storage space is available.

NOTES

These functions only create filenames. It is the application's responsibility to create and remove the files.

Files created using these functions and either `fopen()` or `creat()` 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, names which have been used previously will be reused.

Between the time a filename 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()` and the filenames are chosen to render duplication by other means unlikely.

SEE ALSO

`access(2)`, `creat(2)`, `unlink(2)`, `malloc(3C)`, `mktemp(3C)`, `fopen(3S)`, `tmpfile(3S)`, `stdio(5)`.

NAME

ungetc, ungetwc - push character back onto input stream

SYNOPSIS

```
#include <stdio.h>
□
□
□
int ungetc(int c, FILE *stream);□
cc [ flag ... ] file ... -lw [ library ... ]
#include <stdio.h>
□
#include <wchar.h>□
□
□
wint_t ungetwc(wint_t wc, FILE *stream);□
```

DESCRIPTION

ungetc() pushes back the character specified by *c* (converted to an unsigned char) into the buffer associated with the input stream pointed to by *stream*.

ungetwc() pushes back a wide character *wc* in the buffer. These characters are then returned again by the next [getc\(3S\)](#) or [getwc\(3S\)](#) call for *stream*. ungetc() returns *c*; ungetwc() returns *wc*; the file associated with *stream* remains unchanged.

A successful call to ungetc() or ungetwc() clears the EOF indicator of the input stream.

For ungetc() storage space is guaranteed for a pushed-back character, for ungetwc() a returned wide character is guaranteed.

The value of the file position indicator for *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 zero before a call, its value is indeterminate after the call.

If *c* equals EOF, ungetc() does nothing to the buffer and returns EOF. If the value of *wc* equals that of the macro WEOF, no wide character is pushed back to the buffer.

fseek() and rewind() [both described under [fseek\(3S\)](#)], and setpos() erase the memory of inserted characters for the stream on which they are applied.

If ungetc() or ungetwc() is called too many times on the same stream without an intervening read or file-positioning operation, the function will fail.

RESULT

ungetc() returns EOF if it cannot push back the character.

ungetwc() returns WEOF if it cannot push back the wide character.

If the characters are pushed back, the characters themselves are returned after conversion.

SEE ALSO

[read\(2\)](#), [fsetpos\(3C\)](#), [fgetwc\(3S\)](#), [fseek\(3S\)](#), [getc\(3S\)](#), [rewind\(3S\)](#), [setbuf\(3S\)](#), [stdio\(3S\)](#), [wchar\(5\)](#).

vprintf(3S)**NAME**

`vprintf`, `vfprintf`, `vsprintf`, `vsprintf`, `vwprintf`, `vfwprintf`, `vswprintf` - print formatted output of a variable argument list

SYNOPSIS

```
#include <stdio.h>
#include <stdarg.h>
#include <wchar.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 vwprintf(const wchar_t *format, va_list ap);
int vfwprintf(FILE *stream, const wchar_t *format, va_list ap);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list ap);
```

DESCRIPTION

`vprintf()`, `vfprintf()`, `vsprintf()` and `vsnprintf()` are the same as `printf()`, `fprintf()`, `sprintf()` and `snprintf()`, 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 file.

These functions do not invoke the `va_end` macro. As these functions invoke the `va_arg` macro, the value of `ap` after the return is indeterminate.

The functions `vwprintf()`, `vfwprintf()`, and `vswprintf()` are equivalent to the functions `swprintf()`, `fwprintf()`, and `swprintf()` respectively, with the variable argument list replaced by `ap`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). These functions do not invoke the `va_end` macro.

As the functions `vfwprintf()`, `vswprintf()`, and `vwprintf()` invoke the `va_arg` macro, the value of `ap` after the return is unspecified.

The `stdarg.h` header file 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` 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\(5\)](#)].

EXAMPLES

The following code fragment illustrates the use of `vfprintf()` to write an error routine:

```
#include <stdio.h>
#include <stdarg.h>
/*
 * 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);
    va_arg(ap, char*);
    /* print out remainder of message */
}
```

```

    (void) vfprintf(stderr, format, ap);
    va_end(ap);
    (void) abort;
}

```

The following show the use of the `vfwprintf()` function in a general error-reporting routine.

```

#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
void error(char *function_name, wchar_t *format, ...)
{
    va_list ap;
    va_start(ap, format);
    /* print out name of function causing error */
    fprintf(stderr, L"ERROR in %s: ", function_name);
    /* print out remainder of message */
    vfwprintf(stderr, format, ap);
    va_end(ap);
}

```

RETURN VALUES

`vprintf()` and `fprintf()` return the number of characters transmitted, or return `-1` if an error was encountered.

The `vwprintf()` and the `vfwprintf()` functions return the number of wide-characters transmitted, or a negative value if an output error occurred.

The `vswprintf()` function returns the number of wide-characters written in the array, not counting the terminating null wide-character, or a negative value if `n` or more wide-characters were requested to be generated.

ERRORS

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

These functions will fail [cf. [putc\(3S\)](#)] 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 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 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. 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 process.

These functions may fail [cf. `putc(3S)`] if:

`EILSEQ`

A wide-character code that does not correspond to a valid character has been detected.

`EINVAL`

There are insufficient arguments.

`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.

NOTES

After using these functions you should call the `va_end(ap)` macro to reset the pointer `ap` to a defined value so that any subsequent calls to these functions have correct initial values.

SEE ALSO

`fwprintf(3S)`, `printf(3S)`, `lfs(5)`, `stdarg(5)`, `stdio(5)`, `wchar(5)`.

Functions from Thread libraries(3t)

NAME

`flockfile`, `ftrylockfile`, `funlockfile` - standard input/output locking functions

SYNOPSIS

```
#include <stdio.h>
void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

DESCRIPTION

The `flockfile()`, `ftrylockfile()` and `funlockfile()` functions provide for explicit application-level locking of (`FILE*`) objects. These functions can be used by a thread to delineate a sequence of I/O statements that are to be executed as a unit.

The `flockfile()` function is used by a thread to acquire ownership of a (`FILE*`) object.

The `ftrylockfile()` function is used by a thread to acquire ownership of a (`FILE*`) object if the object is available; `ftrylockfile()` is a non-blocking version of `flockfile()`.

The `funlockfile()` function is used to relinquish the ownership granted to the thread. The behaviour is undefined if a thread other than the current owner calls the `funlockfile()` function.

Logically, there is a lock count associated with each (`FILE*`) object. This count is implicitly initialised to zero when the (`FILE*`) object is created. The (`FILE*`) object is unlocked when the count is zero. When the count is positive, a single thread owns the (`FILE*`) object. When the `flockfile()` function is called, if the count is zero or if the count is positive and the caller owns the (`FILE*`) object, the count is incremented. Otherwise, the calling thread is suspended, waiting for the count to return to zero. Each call to `funlockfile()` decrements the count. This allows matching calls to `flockfile()` [or successful calls to `ftrylockfile()`] and `funlockfile()` to be nested.

All functions that reference (`FILE*`) objects behave as if they use `flockfile()` and `funlockfile()` internally to obtain ownership of these (`FILE*`) objects.

RETURN VALUE

None for `flockfile()` and `funlockfile()`.

The function `ftrylock()` returns zero for success and non-zero to indicate that the lock cannot be acquired.

APPLICATION USAGE

Realtime applications may encounter priority inversion when using `FILE` locks. The problem occurs when a high priority thread "locks" a `FILE` that is about to be "unlocked" 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 `FILE` locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

ERRORS

No errors are defined.

SEE ALSO

`getc_unlocked(3T)`, `pthread_intro(3T)`, `stdio(5)`.

getc_unlocked(3T)**NAME**

`getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` - standard input/output with explicit client locking

SYNOPSIS

```
#include <stdio.h>
int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

DESCRIPTION

Versions of the functions `getc()`, `getchar()`, `putc()`, and `putchar()` respectively named `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` are provided which are functionally identical to the original versions with the exception that they are not required to be implemented in a thread-safe manner. They may only safely be used within a scope protected by `flockfile()` [or `ftrylockfile()`] and `funlockfile()`. These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (`FILE*`) object, as is the case after a successful call of the `flockfile()` or `ftrylockfile()` functions.

RETURN VALUE

See `getc()`, `getchar()` [both in [getc\(3S\)](#)], `putc()`, and `putchar()` [both in [putc\(3S\)](#)].

ERRORS

No errors are defined.

SEE ALSO

[getc\(3S\)](#), [putc\(3S\)](#), [flockfile\(3T\)](#), [pthread_intro\(3T\)](#), [stdio\(5\)](#).

pthread_atfork(3T)**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 before and after `fork()`, in the context of the thread that called `fork()`. The *prepare* fork handler is called before `fork()` processing commences. The *parent* fork handle is called after `fork()` processing completes in the parent process. The *child* fork handler is called after `fork()` processing completes in the child process. If no handling is desired at one or more of these three points, the corresponding fork handler address(es) may be set to `NULL`.

The order of calls to `pthread_atfork()` is significant. The *parent* and *child* fork handlers are called in the order in which they were established by calls to `pthread_atfork()`. The *prepare* fork handlers are called in the opposite order.

RETURN VALUE

Upon successful completion, `pthread_atfork()` returns a value of zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_atfork()` function will fail if:

`ENOMEM`

Insufficient table space exists to record the fork handler addresses.

The `pthread_atfork()` function will not return an error code of `EINTR`.

SEE ALSO

`fork(2)`, `atexit(3C)`, `pthread_intro(3T)`, `unistd(4)`, `types(5)`.

pthread_attr_getguardsize(3T)**NAME**

pthread_attr_getguardsize, pthread_attr_setguardsize - get or set the thread *guardsize* attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *attr,
                             □□□ size_t *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 zero, a guard area will not be provided for threads created with *attr*. If *guardsize* is greater than zero, 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` [see `<sys/mman.h>` in `mman(5)`]. 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 VALUE

If successful, the `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions return zero. 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.

SEE ALSO

`pthread_intro(3T)`, `pthread(5)`.

pthread_attr_init(3T)**NAME**

pthread_attr_init, pthread_attr_destroy - initialise and destroy threads attribute object

SYNOPSIS

```
#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()` initialises 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_destroy()` function is used to destroy a thread attributes object. An implementation may cause `pthread_attr_destroy()` to set *attr* to an implementation-dependent invalid value. The behaviour of using the attribute after it has been destroyed is undefined.

RETURN VALUE

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 initialise the thread attributes object.

These functions will not return an error code of `EINTR`.

SEE ALSO

`pthread_attr_setdetachstate(3T)`, `pthread_attr_setstackaddr(3T)`,
`pthread_attr_setstacksize(3T)`, `pthread_create(3T)`, `pthread_intro(3T)`, `pthread(5)`.

pthread_attr_setdetachstate(3T)**NAME**

pthread_attr_setdetachstate, pthread_attr_getdetachstate - set and get *detachstate* attribute

SYNOPSIS

```
#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 VALUE

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()` function will fail if:

`EINVAL`

The value of *detachstate* was not valid

These functions will not return an error code of `EINTR`.

SEE ALSO

`pthread_attr_init(3T)`, `pthread_attr_setstackaddr(3T)`, `pthread_attr_setstacksize(3T)`, `pthread_create(3T)`, `pthread_intro(3T)`, `pthread(5)`.

pthread_attr_setinheritsched(3-thr)**NAME**

`pthread_attr_setinheritsched` - changes the inherit scheduling attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setinheritsched(
    pthread_attr_t attr,
    int inherit);
```

PARAMETERS

attr Thread attributes object to be modified.

inherit

New value for the inherit scheduling attribute. Valid values are as follows:

`PTHREAD_INHERIT_SCHED`

This is the default value. The created thread inherits the current priority and scheduling policy of the thread calling `pthread_create()`.

`PTHREAD_DEFAULT_SCHED`

The created thread starts execution with the priority and scheduling policy stored in the thread attributes object.

DESCRIPTION

The `pthread_attr_setinheritsched()` routine changes the inherit scheduling attribute of thread creation. The inherit scheduling attribute specifies whether threads created using the specified thread attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the thread attributes object that is passed to `pthread_create()`.

The first thread in an application that is not created by an explicit call to `pthread_create()` has a scheduling policy of `SCHED_OTHER`. (See the `pthread_attr_setprio()` and `pthread_attr_setsched()` routines for more information on valid priority values and valid scheduling policy values, respectively.)

Inheriting scheduling attributes (instead of using the scheduling attributes stored in the attributes object) is useful when a thread is creating several helper threads – threads that are intended to work closely with the creating thread to cooperatively solve the same problem. For example, inherited scheduling attributes ensure that helper threads created in a sort routine execute with the same priority as the calling thread.

RETURN VALUES

If the function fails, `-1` is returned, and `errno` may be set to one of the following values:

`EINVAL`

The value specified by *attr* is invalid.

`EINVAL`

The value specified by *inherit* is invalid.

SEE ALSO

`pthread_attr_create(3-thr)`, `pthread_attr_getinheritsched(3-thr)`,
`pthread_attr_setprio(3-thr)`, `pthread_attr_setsched(3-thr)`, `pthread_create(3-thr)`.

pthread_attr_setschedparam(3T)**NAME**

pthread_attr_setschedparam, pthread_attr_getschedparam - **set and get schedparam attribute**

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setschedparam(pthread_attr_t *attr,
    const struct sched_param *param);

int pthread_attr_getschedparam(const pthread_attr_t *attr,
    struct sched_param *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>`. For the `SCHED_FIFO` and `SCHED_RR` policies, the only required member of *param* is `sched_priority`.

RETURN VALUE

If successful, the `pthread_attr_setschedparam()` and `pthread_attr_getschedparam()` functions return zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_attr_setschedparam()` function may fail if:

`EINVAL`

The value of the attribute being set is not valid.

`ENOTSUP`

An attempt was made to set the attribute to an unsupported value.

The `pthread_attr_setschedparam()` and `pthread_attr_getschedparam()` functions will not return an error code of `EINTR`.

APPLICATION 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.

SEE ALSO

`pthread_attr_init(3T)`, `pthread_attr_setinheritsched(3T)`,
`pthread_attr_setschedpolicy(3T)`, `pthread_attr_setscope(3T)`, `pthread_create(3T)`,
`pthread_intro(3T)`, `pthread_setsched_param(3T)`, `pthread(5)`, `sched(5)`.

pthread_attr_setschedpolicy(3T)**NAME**

pthread_attr_setschedpolicy, pthread_attr_getschedpolicy - set and get schedpolicy attribute (**REALTIME THREADS**)

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                                int *policy);
```

DESCRIPTION

ENOSYS is returned every time this interface is called, since **REALTIME THREADS** are not supported.

SEE ALSO

pthread_attr_init(3T), pthread_attr_setinheritsched(3T),
pthread_attr_setschedparam(3T), pthread_attr_setscope(3T), pthread_create(3T),
pthread_intro(3T), pthread_setsched_param(3T), pthread(5), sched(5).

pthread_attr_setscope(3T)**NAME**

pthread_attr_setscope, pthread_attr_getscope - set and get contentionscope attribute
(**REALTIME THREADS**)

SYNOPSIS

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
int pthread_attr_getscope(const pthread_attr_t *attr,
    int *contentionscope);
```

DESCRIPTION

ENOSYS is returned every time this interface is called, since **REALTIME THREADS** are not supported.

SEE ALSO

pthread_attr_init(3T), pthread_attr_setinheritsched(3T),
pthread_attr_setschedparam(3T), pthread_attr_setschedpolicy(3T), pthread_create(3T),
pthread_intro(3T), pthread_setsched_param(3T), pthread(5), sched(5).

pthread_attr_setstackaddr(3T)**NAME**

pthread_attr_setstackaddr, pthread_attr_getstackaddr - set and get *stackaddr* attribute

SYNOPSIS

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

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.

The `pthread_attr_getstackaddr()` function stores the *stackaddr* attribute value in *stackaddr* if successful.

ERRORS

No errors are defined.

These functions will not return an error code of `EINTR`.

SEE ALSO

[pthread_attr_init\(3T\)](#), [pthread_attr_setdetachstate\(3T\)](#),
[pthread_attr_setstacksize\(3T\)](#), [pthread_create\(3T\)](#), [pthread_intro\(3T\)](#), [limits\(4\)](#),
[pthread\(5\)](#).

pthread_attr_setstacksize(3-thr)**NAME**

`pthread_attr_setstacksize` - changes the stacksize attribute of thread creation

SYNOPSIS

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

PARAMETERS

attr Thread attributes object modified.

stacksize

New value for the stacksize attribute. The *stacksize* parameter specifies the minimum size (in bytes) of the stack needed for a thread.

DESCRIPTION

The `pthread_attr_setstacksize()` routine sets the minimum size (in bytes) of the stack needed for a thread created using the attributes object specified by the *attr* parameter. Use this routine to adjust the size of the writable area of the stack. The default value of the stacksize attribute is machine specific.

A thread's stack is fixed at the time of thread creation. Only the main or initial thread can dynamically extend its stack.

Most compilers do not check for stack overflow. Ensure that your thread stack is large enough for anything that you call from the thread.

RETURN VALUES

If the function fails, `errno` may be set to one of the following values:

`EINVAL`

The value specified by *attr* is invalid.

`EINVAL`

The value specified by *stacksize* is invalid.

SEE ALSO

`pthread_attr_create(3-thr)`, `pthread_attr_getstacksize(3-thr)`, `pthread_create(3-thr)`.

pthread_cancel(3-thr)**NAME**

`pthread_cancel` - allows a thread to request that it or another thread terminate execution

SYNOPSIS

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

PARAMETERS

thread

Thread that receives a cancel request.

DESCRIPTION

The `pthread_cancel()` routine sends a cancel to the specified thread. A cancel is a mechanism by which a calling thread informs either itself or the called thread to terminate as quickly as possible. Issuing a cancel does not guarantee that the canceled thread receives or handles the cancel. The canceled thread can delay processing the cancel after receiving it. For instance, if a cancel arrives during an important operation, the canceled thread can continue if what it is doing cannot be interrupted at the point where the cancel is requested.

Because of communications delays, the calling thread can only rely on the fact that a cancel eventually becomes pending in the designated thread (provided that the thread does not terminate beforehand). Furthermore, the calling thread has no guarantee that a pending cancel is to be delivered because delivery is controlled by the designated thread.

Termination processing when a cancel is delivered to a thread is similar to `pthread_exit()`. Outstanding cleanup routines are executed in the context of the target thread, and a status of `-1` is made available to any threads joining with the target thread.

This routine is preferred in implementing Ada's `abort` statement and any other language (or software-defined construct) for requesting thread cancellation.

The results of this routine are unpredictable if the value specified in *thread* refers to a thread that does not currently exist.

RETURN VALUES

If the function fails, `errno` may be set to one of the following values:

`EINVAL`

The specified thread is invalid.

`ERSCH`

The specified thread does not refer to a currently existing thread.

SEE ALSO

`pthread_exit(3-thr)`, `pthread_join(3-thr)`, `pthread_setasynccancel(3-thr)`,
`pthread_setcancel(3-thr)`, `pthread_testcancel(3-thr)`.

pthread_cleanup_push(3-thr)**NAME**

`pthread_cleanup_push` - establishes a cleanup handler

SYNOPSIS

```
#include <pthread.h>
void pthread_cleanup_push(□
void routine, □
pthread_addr_t arg) ; □
```

PARAMETERS

routine

Routine executed as the cleanup handler.

arg

Parameter executed with the cleanup routine.

DESCRIPTION

The `pthread_cleanup_push()` routine pushes the specified routine onto the calling thread's cleanup stack. The cleanup routine is popped from the stack and executed with the *arg* parameter when any of the following actions occur:

- The thread calls `pthread_exit()`.
- The thread is canceled.
- The thread calls `pthread_cleanup_pop()` and specifies a nonzero value for the *execute* parameter.

This routine and `pthread_cleanup_pop()` are implemented as macros and must be displayed as statements and in pairs within the same lexical scope. You can think of the `pthread_cleanup_push()` macro as expanding to a string whose first character is a { (left brace) and `pthread_cleanup_pop()` as expanding to a string containing the corresponding } (right brace).

RETURN VALUES

This routine must be used as a statement.

SEE ALSO

`pthread_cancel(3-thr)`, `pthread_cleanup_pop(3-thr)`, `pthread_exit(3-thr)`,
`pthread_testcancel(3-thr)`.

pthread_condattr_getpshared(3T)**NAME**

pthread_condattr_getpshared, pthread_condattr_setpshared - get and set the process-shared condition variable attributes

SYNOPSIS

```
#include <pthread.h>
int pthread_condattr_getpshared(const pthread_condattr_t *attr,
                                int *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 initialised 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 initialised the condition variable; if threads of differing processes attempt to operate on such a condition variable, the behaviour 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 VALUE

If successful, the pthread_condattr_setpshared() function returns zero. Otherwise, an error number is returned to indicate the error.

If successful, the pthread_condattr_getpshared() function returns zero 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 may fail if:

EINVAL

The new value specified for the attribute is outside the range of legal values for that attribute.

These functions will not return an error code of EINTR.

SEE ALSO

pthread_condattr_init(3T), pthread_cond_init(3T), pthread_create(3T), pthread_intro(3T), pthread_mutex_init(3T), pthread(5).

pthread_condattr_init(3T)**NAME**

`pthread_condattr_init`, `pthread_condattr_destroy` - initialise and destroy condition variable attributes object

SYNOPSIS

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

DESCRIPTION

The function `pthread_condattr_init()` initialises a condition variable attributes object *attr* with the default value for all of the attributes defined by the implementation.

Attempting to initialise an already initialised condition variable attributes object results in undefined behaviour.

After a condition variable attributes object has been used to initialise one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialised condition variables.

The `pthread_condattr_destroy()` function destroys a condition variable attributes object; the object becomes, in effect, uninitialised. 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-initialised 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 VALUE

If successful, the `pthread_condattr_init()` and `pthread_condattr_destroy()` functions return zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_condattr_init()` function will fail if:

`ENOMEM`

Insufficient memory exists to initialise the condition variable attributes object.

The `pthread_condattr_destroy()` function may fail if:

`EINVAL`

The value specified by *attr* is invalid.

These functions will not return an error code of `EINTR`.

SEE ALSO

`pthread_condattr_getpshared(3T)`, `pthread_cond_init(3T)`, `pthread_create(3T)`, `pthread_intro(3T)`, `pthread_mutex_init(3T)`, `pthread(5)`.

pthread_cond_init(3-thr)**NAME**

`pthread_cond_init` - creates a condition variable

SYNOPSIS

```
#include <pthread.h>
int pthread_cond_init(
    pthread_cond_t *cond,
    pthread_condattr_t attr);
```

PARAMETERS

cond Condition variable that is created.

attr Condition variable attributes object that defines the characteristics of the condition variable created. If you specify `pthread_condattr_default`, default attributes are used.

DESCRIPTION

The `pthread_cond_init()` routine creates and initializes a condition variable. A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to shared data; a condition variable allows threads to wait for that data to enter a defined state. The state is defined by a Boolean expression called a predicate.

A condition variable is signaled or broadcast to indicate that a predicate might have become true. The broadcast operation indicates that all waiting threads need to resume and reevaluate the predicate. The signal operation is used when any one waiting thread can continue.

If a thread that holds a mutex determines that the shared data is not in the correct state for it to proceed (the associated predicate is not true), it waits on a condition variable associated with the desired state. Waiting on the condition variable automatically releases the mutex so that other threads can modify or examine the shared data. When a thread modifies the state of the shared data so that a predicate might be true, it signals or broadcasts on the appropriate condition variable so that threads waiting for that predicate can continue.

It is important that all threads waiting on a particular condition variable at any time hold the *same* mutex. If they do not, the behavior of the wait operation is unpredictable (an implementation can use the mutex to control internal access to the condition variable object). However, it is legal for a client to store condition variables and mutexes and later reuse them in different combinations. The client must ensure that no threads use the condition variable with the old mutex. At any time, an arbitrary number of condition variables can be associated with a single mutex, each representing a different predicate of the shared data protected by that mutex.

Condition variables are not owned by a particular thread. Any associated storage is not automatically deallocated when the creating thread terminates.

RETURN VALUES

If the function fails, `errno` may be set to one of the following values:

`EAGAIN`

The system lacks the necessary resources to initialize another condition variable.

`EINVAL`

Invalid attributes object.

`ENOMEM`

Insufficient memory exists to initialize the condition variable.

SEE ALSO

`pthread_cond_broadcast(3-thr)`, `pthread_cond_destroy(3-thr)`, `pthread_cond_signal(3-thr)`,


```
pthread_cond_timedwait(3-thr), pthread_cond_wait(3-thr).
```

pthread_cond_signal(3-thr)**NAME**

pthread_cond_signal - wakes one thread that is waiting on a condition variable

SYNOPSIS

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

PARAMETERS

cond Condition variable signaled.

DESCRIPTION

The `pthread_cond_signal()` routine wakes one thread waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so that it is possible for a single waiting thread to proceed. Call this routine when any thread waiting on the specified condition variable might find its predicate true, but only one thread needs to proceed.

The scheduling policy determines which thread is awakened. For policies `SCHED_FIFO` and `SCHED_RR` a blocked thread is chosen in priority order.

Call this routine when the associated mutex is either locked or unlocked.

RETURN VALUES

If the function fails, `errno` may be set to one of the following values:

`EINVAL`

The value specified by *cond* is invalid.

SEE ALSO

`pthread_cond_broadcast(3-thr)`, `pthread_cond_destroy(3-thr)`, `pthread_cond_init(3-thr)`,
`pthread_cond_timedwait(3-thr)`, `pthread_cond_wait(3-thr)`.

pthread_cond_wait(3-thr)**NAME**

`pthread_cond_wait` - causes a thread to wait for a condition variable to be signaled or broadcast

SYNOPSIS

```
#include <pthread.h>
int pthread_cond_wait(
pthread_cond_t *cond,
pthread_mutex_t *mutex);
```

PARAMETERS

cond Condition variable waited on.

mutex

Mutex associated with the condition variable specified in *cond*.

DESCRIPTION

The `pthread_cond_wait()` routine causes a thread to wait for a condition variable to be signaled or broadcast. Each condition corresponds to one or more predicates based on shared data. The calling thread waits for the data to reach a particular state (for the predicate to become true).

Call this routine after you have locked the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

This routine automatically releases the mutex and causes the calling thread to wait on the condition. If the wait is satisfied as a result of some thread calling `pthread_cond_signal()` or `pthread_cond_broadcast()`, the mutex is reacquired and the routine returns.

A thread that changes the state of storage protected by the mutex in such a way that a predicate associated with a condition variable might now be true must call either `pthread_cond_signal()` or `pthread_cond_broadcast()` for that condition variable. If neither call is made, any thread waiting on the condition variable continues to wait.

This routine might (with low probability) return when the condition variable has not been signaled or broadcast. When a spurious wakeup occurs, the mutex is reacquired before the routine returns. (To handle this type of situation, enclose this routine in a loop that checks the predicate.)

RETURN VALUES

If the function fails, `errno` may be set to one of the following values:

`EINVAL`

The value specified by *cond* or *mutex* is invalid.

`EDEADLK`

A deadlock condition is detected.

SEE ALSO

`pthread_cond_broadcast(3-thr)`, `pthread_cond_destroy(3-thr)`, `pthread_cond_init(3-thr)`, `pthread_cond_signal(3-thr)`, `pthread_cond_timedwait(3-thr)`.

pthread_create(3-thr)**NAME**

pthread_create - creates a thread object and thread

SYNOPSIS

```
#include <pthread.h>
int pthread_create(
    pthread_t *thread,
    pthread_attr_t attr,
    pthread_startroutine_t start_routine,
    pthread_addr_t arg);
```

PARAMETERS

thread

Handle to the thread object created.

attr Thread attributes object that defines the characteristics of the thread being created. If you specify `pthread_attr_default`, default attributes are used.

start_routine

Function executed as the new thread's start routine.

arg Address value copied and passed to the thread's start routine.

DESCRIPTION

The `pthread_create()` routine creates a thread object and a thread. A thread is a single, sequential flow of control within a program. It is the active execution of a designated routine, including any nested routine invocations. A thread object defines and controls the executing thread.

Creating a Thread

Calling this routine sets into motion the following actions:

- An internal thread object is created to describe the thread.
- The associated executable thread is created with attributes specified by the *attr* parameter (or with default attributes if `pthread_attr_default` is specified).
- The *thread* parameter receives the new thread.
- The *start_routine* function is called. This may occur before this routine returns successfully.

Thread Execution

The thread is created in the ready state and therefore might immediately begin executing the function specified by the *start_routine* parameter. The newly created thread begins running before `pthread_create()` completes if the new thread follows the `SCHED_RR` or `SCHED_FIFO` scheduling policy or has a priority higher than the creating thread, or both. Otherwise, the new thread begins running at its turn, which with sufficient processors might also be before `pthread_create()` returns.

The *start_routine* parameter is passed a copy of the *arg* parameter. The value of the *arg* parameter is unspecified.

The thread object exists until the `pthread_detach()` routine is called or the thread terminates, whichever occurs last.

The synchronization between the caller of `pthread_create()` and the newly created thread is through the use of the `pthread_join()` routine (or any other mutexes or condition variables they agree to use).

Terminating a Thread

A thread terminates when one of the following events occurs:

- The thread returns from its start routine.
- The thread exits (within a routine) as the result of calling the `pthread_exit()` routine.
- The thread is canceled.

When a Thread Terminates

The following actions are performed when a thread terminates:

- If the thread terminates by returning from its start routine or calling `pthread_exit()`, the return value is copied into the thread object. If the start routine returns normally and the start routine is a procedure that does not return a value, then the result obtained by `pthread_join()` is unpredictable. If the thread has been cancelled, a return value of `-1` is copied into the thread object. The return value can be retrieved by other threads by calling the `pthread_join()` routine.
- A destructor for each thread-specific data point is removed from the list of destructors for this thread and then is called. This step destroys all the thread-specific data associated with the current thread.
- Each cleanup handler that has been declared by `pthread_cleanup_push()` and not yet removed by `pthread_cleanup_pop()` is called. The most recently pushed handler is called first.
- A flag is set in the thread object indicating that the thread has terminated. This flag must be set in order for callers of `pthread_join()` to return from the call.
- A broadcast is made so that all threads currently waiting in a call to `pthread_join()` can return from the call.
- The thread object is marked to indicate that it is no longer needed by the thread itself. A check is made to determine if the thread object is no longer needed by other threads; that is, if `pthread_detach()` has been called. If that routine is called, then the thread object is deallocated.

RETURN VALUES

Upon successful completion, this routine stores the identifier of the created thread at *thread* and returns 0. Otherwise, a value of `-1` is returned and no thread is created, the contents of *thread* are undefined, and `errno` may be set to one of the following values:

EAGAIN

The system lacks the necessary resources to create another thread.

ENOMEM

Insufficient memory exists to create the thread object. This is not a temporary condition.

SEE ALSO

`pthread_attr_create(3-thr)`, `pthread_cancel(3-thr)`, `pthread_detach(3-thr)`,
`pthread_exit(3-thr)`, `pthread_join(3-thr)`.

pthread_detach(3-thr)**NAME**

`pthread_detach` - marks a thread object for deletion

SYNOPSIS

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

PARAMETERS

thread

Thread object marked for deletion.

DESCRIPTION

The `pthread_detach()` routine indicates that storage for the specified thread is reclaimed when the thread terminates. This includes storage for the *thread* parameter's return value. If *thread* has not terminated when this routine is called, this routine does not cause it to terminate.

Call this routine when a thread object is no longer referenced. Additionally, call this routine for every thread that is created to ensure that storage for thread objects does not accumulate.

You cannot join with a thread after the thread has been detached.

The results of this routine are unpredictable if the value of *thread* refers to a thread object that does not exist.

RETURN VALUES

If the function fails, `errno` may be set to one of the following values:

`EINVAL`

The value specified by *thread* is invalid.

`ESRCH`

The value specified by *thread* does not refer to an existing thread.

SEE ALSO

`pthread_cancel(3-thr)`, `pthread_create(3-thr)`, `pthread_exit(3-thr)`, `pthread_join(3-thr)`.

pthread_equal(3-thr)**NAME**

`pthread_equal` - compares one thread identifier to another thread identifier

SYNOPSIS

```
#include <pthread.h>
boolean32 pthread_equal(pthread_t *thread1, pthread_t *thread2);
```

PARAMETERS

thread1

The first thread identifier to be compared.

thread2

The second thread identifier to be compared.

DESCRIPTION

This routine compares one thread identifier to another thread identifier. (This routine does not check whether the objects that correspond to the identifiers currently exist.) If the identifiers have values indicating that they designate the same object, 1 (true) is returned. If the values do not designate the same object, 0 (false) is returned.

This routine is implemented as a C macro.

RETURN VALUES

Possible return values are as follows:

- 0 Values of *thread1* and *thread2* do not designate the same object.
- 1 Values of *thread1* and *thread2* designate the same object.

SEE ALSO

`pthread_create(3-thr)`.

pthread_exit(3-thr)**NAME**

`pthread_exit` - terminates the calling thread

SYNOPSIS

```
#include <pthread.h>
void pthread_exit(pthread_addr_t status);
```

PARAMETERS

status

Address value copied and returned to the caller of `pthread_join()`.

DESCRIPTION

The `pthread_exit()` routine terminates the calling thread and makes a status value available to any thread that calls `pthread_join()` and specifies the terminating thread.

An implicit call to `pthread_exit()` is issued when a thread returns from the start routine that was used to create it. The function's return value serves as the thread's exit status. If the return value is `-1`, an error exit is forced for the thread instead of a normal exit. The process exits when the last running thread calls `pthread_exit()`, with an undefined exit status.

Restrictions

The `pthread_exit()` routine does not work in the main (initial) thread because DCE Threads relies on information at the base of thread stacks; this information does not exist in the main thread.

RETURN VALUES

No value is returned.

SEE ALSO

[pthread_create\(3-thr\)](#), [pthread_detach\(3-thr\)](#), [pthread_join\(3-thr\)](#).

pthread_getconcurrency(3T)**NAME**

pthread_getconcurrency, pthread_setconcurrency - get or set level of concurrency

SYNOPSIS

```
#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 zero, 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 zero 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 VALUE

If successful, the pthread_setconcurrency() function returns zero. 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 zero.

ERRORS

The pthread_setconcurrency() function will fail if:

EAGAIN

The value specified by *new_level* would cause a system resource to be exceeded.

EINVAL

The value specified by *new_level* is negative.

APPLICATION USAGE

Use of these functions changes the state of the underlying concurrency upon which the application depends.

Library developers are advised to not use the pthread_getconcurrency() and pthread_setconcurrency() functions since their use may conflict with an applications use of these functions.

SEE ALSO

`pthread_intro(3T)`, `pthread(5)`.

pthread_getschedparam(3T)**NAME**

pthread_getschedparam, pthread_setschedparam - dynamic thread scheduling parameters access (**REALTIME THREADS**)

SYNOPSIS

```
#include <pthread.h>

int pthread_getschedparam(pthread_t thread, int *policy,
    □ struct sched_param *param);□

int pthread_setschedparam(pthread_t thread, int *policy,
    □ const struct sched_param *param);□
```

DESCRIPTION

ENOSYS is returned every time this interface is called, since **REALTIME THREADS** are not supported.

SEE ALSO

[pthread_intro\(3T\)](#), [pthread\(5\)](#), [sched\(5\)](#).

NAME

pthread_intro - introduction to UNIX 98 threads

DESCRIPTION**About multi-threaded programming**

The choice of multi-threading is really a question of specific application design. Multi-threaded programming, in general, requires a number of practices that are likely to be unfamiliar and unintuitive to many programmers, and errors arising from failure to follow these practices can be obscure, infrequent and difficult to reproduce. There are many potential errors in a multi-threaded program that can result in all kinds of deadlocks, contentious situations, and even data corruption (see the section **Thread-safeness and reentrance**). Cancels and signals introduce a number of specific semantic issues that applications must be aware of when programming in a multi-threaded environment.

Correct use of threads mechanisms means following a set of general rules designed to avoid errors that may or may not occur in specific cases. Rules like this are, in general, not enforced by the thread programming mechanisms, and failure to follow them will not always result in program failures (see the section **Synchronization strategies and deadlock avoidance**).

Application programmers need to be aware that, depending on the threads implementation and the underlying hardware, concurrency may be more apparent than real for many applications. If threads are being timesliced on a *single* processor, non-blocking activities will not go any faster because they are multi-threaded. In fact, given the extra overhead of a threads implementation, they might even be slower. On the other hand, if multiple threads are carrying out activities that may block, then multi-threading will probably be beneficial.

In summary, the developer must consider the following questions in order to decide whether an application will benefit from multi-threading:

- Are the threaded operations likely to block, for example, because they make blocking I/O calls? If so, then multi-threading is likely to be beneficial in any implementation or hardware environment.
- Can the underlying hardware and implementation support threads on more than one processor within a single process? If not, then multi-threading cannot achieve *real* parallelism for *processor intensive* operations.
- Even if the answer to both of the first two questions is yes, will the use of a timeslicing thread scheduling policy enable more equitable distribution of resources among contending clients? If so, then multi-threading may be beneficial.

If programming is to be thread-safe, the code must be structured such that it can be executed simultaneously by multiple threads. For thread-safe programming, you have to use and implement thread-safe functions. To make a function thread-safe you have to do the following:

Use mutexes and condition variables

These work on one appointed thread and are prepared by the application programmer. They are used to synchronize threads on the application level. They are described in the section **Threads synchronization objects**.

Make global data thread specific

This allows a thread to allocate and maintain private static data that needs no protection by mutexes against concurrent access by other threads. This is described in the section **Thread specific data**.

Use thread-safe libraries

The functions in thread-safe libraries are either thread-reentrant as such or they are "jacketed". For some functions the library may provide a thread-safe renamed "_r" function. The programmer can use this function in place of the non-reentrant function. For example you can use `strtok_r(3C)` which is reentrant instead of `strtok(3C)` which is not reentrant. The thread-safe system libraries are

described in the section **Thread-safe system libraries**.

Multi-threaded programming imposes restrictions and specific semantic issues for certain system interfaces [for example, signal handling, `fork(2)` and `exec(2)`] and system libraries. Using thread-safe (system) libraries within a non-threaded program is undefined. Using thread-unsafe (system) libraries within a multi-threaded program is possible if controlled appropriately by the application.

Any executable program (or object or library) built with the compiler's `thread` option or linked with modules of the threaded C/C++ runtime support becomes a multi-threaded program (or object or library) with all side effects on functionality (for example, signal handling, `fork(2)` / `exec(2)` restrictions). On the other hand, a multi-threaded program with exactly one thread *is not equivalent* to a non-threaded program.

UNIX 98 threads

The Single UNIX Specification, Version 2 [UNIX 98], subsequently called UNIX 98, defines interfaces and functionality to support multiple flows of control, called *threads*, within a UNIX process. The UNIX 98 glossary defines a thread as:

"A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy, `errno` value, thread-specific key/value bindings, and the required system resources to support a flow of control. Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via `malloc()`, directly addressable storage obtained through implementation-supplied functions and automatic variables, are accessible to all threads in the same process."

UNIX 98 threads define 70 (plus 14 optional) system interfaces to support the source portability of multi-threaded applications. They are guaranteed to be present on X/Open branded UNIX 98 systems.

UNIX 98 threads are built upon the foundation of IEEE Standard POSIX 1003.1c-1995 [POSIX.1c][POSIX.1], known as *pthreads*. UNIX 98 threads make all of the mandatory plus most of the optional functionality of *pthreads* mandatory and adds some mandatory extension not defined in *pthreads*. These include: setting the level of thread concurrency, read/write locks, additional threads and mutex attribute types, additional scheduling policies. Some optional threads interfaces from *pthreads* are collected into the X/Open Realtime Threads Feature group.

The specific functional areas covered by UNIX 98 threads and their scope includes:

Thread management

including the creation, control, and termination of multiple flows of control (threads) in the same process under the assumption of a common, shared address space.

Synchronization primitives

optimized for tightly coupled operation of multiple flows of control in a common, shared address space.

Harmonization with the existing BASE interfaces

the resolution of the impact of multiple flows of control (threads) onto the system interfaces previously defined under the assumption of one flow of control. This is the issue of thread-safe system libraries [see `pthread_syslib_intro(3T)`].

There are no administrative tasks involved with UNIX 98 threads.

Thread implementation models

There are various thread implementation models. In the *user-space threads* model, the threads in a process are not visible to the operating system kernel. The process is the only entity scheduled by the kernel, its threads are not. The process is scheduled onto a processor according to its scheduling attributes. Within the execution time (slice) of the process, its threads are scheduled by the threads runtime library according to their scheduling attributes. This model offers virtual thread concurrency via I/O overlap. Only one thread per process can execute at a time. If this thread blocks on I/O, then other threads in the process continue to execute or the entire process blocks.

In the *kernel-threads* model, all threads within a process are visible to the operating system kernel. Thus, all threads are kernel scheduled entities, and threads within the same process can really concurrently execute. The threads are scheduled onto processors by the kernel according to their scheduling attributes. Creation and management of the threads entails operating system call overhead and strains operating system

resources.

UNIX 98 threads in Reliant UNIX 5.45B implement the *kernel-thread* model. Each thread is an individual Reliant UNIX process, cloned from its creator process and sharing its address space and other process related resources. Thus *threads within the same process are processes sharing the same address space*.

The programming environment

Feature test macros

The following symbolic constants are always defined in `<unistd.h>` to unspecified values to indicate that this functionality is always present on Reliant UNIX 5.45B as a system in conformity with UNIX 98:

Symbolic constants and description	Defined
<code>_POSIX_THREADS</code> ☐ The implementation supports the threads option.	yes
<code>_POSIX_THREAD_ATTR_STACKADDR</code> ☐ The implementation supports the thread stack address attribute option.	yes
<code>_POSIX_THREAD_ATTR_STACKSIZE</code> ☐ The implementation supports the thread stack size attribute option.	yes
<code>_POSIX_THREAD_PROCESS_SHARED</code> ☐ The implementation supports the process-shared synchronization option.	yes
<code>_POSIX_THREAD_SAFE_FUNCTIONS</code> ☐ The implementation supports the thread-safe functions ("_r" functions) option.	yes

The following symbolic constants are either defined in `<unistd.h>` to have the value `-1` or are not defined, since Reliant UNIX 5.45B will not provide this feature.

Symbolic constants and description	Defined
<code>_XOPEN_REALTIME_THREADS</code> ☐ The implementation does not support the X/Open Realtime Threads Feature Group.	to <code>-1</code>
<code>_POSIX_THREAD_PRIORITY_SCHEDULING</code> ☐ The implementation does not support the thread execution scheduling option.	no
<code>_POSIX_THREAD_PRIO_INHERIT</code> ☐ The implementation does not support the priority inheritance option.	no
<code>_POSIX_THREAD_PRIO_PROTECT</code> ☐ The implementation does not support the priority protection option.	no

Implementation limits

The following runtime invariant values are in `<limits.h>` with the given minimal values:

Symbolic constants and description	Minimal value
<code>PTHREAD_THREADS_MAX</code> ☐ Threads within a process.	<code>_POSIX_THREAD_THREADS_MAX</code> (64) *)
<code>PTHREAD_KEYS_MAX</code> ☐ Maximum number of data keys that can be created by a process.	<code>_POSIX_THREAD_KEYS_MAX</code> (128)
<code>PTHREAD_DESTRUCTOR_ITERATIONS</code> ☐ Maximum number of attempts made to destroy a thread's thread-specific data values on thread exit.	<code>_POSIX_THREAD_DESTRUCTOR_ITERATIONS</code> (4)

PTHREAD_STACK_MIN □ Minimum stack size for a thread.	0

*) The maximum value cannot be larger than the process limit for an (unprivileged) user.

The configurable system variables available via `sysconf()` for UNIX 98 threads are:

Variable	Value in <unistd.h>	Value
PTHREAD_DESTRUCT_OR_ITERATIONS	_SC_THREAD_DESTRUCTOR_ITERATIONS	4
PTHREAD_KEYS_MAX	_SC_THREAD_KEYS_MAX	128
PTHREAD_STACK_MIN	_SC_THREAD_STACK_MIN	32.768
PTHREAD_THREADS_MAX	_SC_THREAD_THREADS_MAX	65.535
_POSIX_THREADS	_SC_THREADS	1
_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR	1
_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE	1
_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING	-1
_POSIX_THREAD_PRIORITY_INHERIT	_SC_THREAD_PRIORITY_INHERIT	-1
_POSIX_THREAD_PRIORITY_PROTECT	_SC_THREAD_PRIORITY_PROTECT	-1
_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED	1
_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS	1

Name space for UNIX 98 threads

In the UNIX 98 threads implementation in Reliant UNIX 5.45B, objects (data types) and function calls follow the naming convention

`pthread[_object]_t`, `pthread[_object]_operation[_np]`

Here, *object* optionally specifies the object (data type) more closely and *operation* specifies the operation on *object*. The suffix `_np` indicates a non-portable extension with respect to [UNIX 98].

The UNIX 98 threads implementation in Reliant UNIX 5.45B defines (and reserves) all identifiers with prefix `pthread` and `PTHREAD`, and, in addition to UNIX 98, `cma`, `CMA`, `libc`, and `rtld`.

Include files

All source using UNIX 98 threads must include the header file

```
#include <pthread.h>
```

This file resides in a standard place in the Reliant UNIX file system.

Compiler issues

The Reliant UNIX C/C++ Compiler CDS++ 2.0C [C/C++] supports compiling and linking of applications using UNIX 98 threads with several options as specified below. For support by older compiler versions see the UNIX 98 threads release notes.

Pre-Processor definitions and search paths

The compiler's `-K pthread` option indicates usage of UNIX 98 threads with the following defaults:

Predefined names

The preprocessor names `__SNI_SVR4`, `__SNI_THREAD_SUPPORT`, `__SNI_DCOSX`, `_REENTRANT`, `_POSIX_THREADS`, and `_UNIX98_THREADS` are always defined.

Include paths

The compiler automatically adds the path `/opt/pthread/include` to the list of standard directories searched for include files.

Standard libraries

The compiler automatically adds the library option `-l pthread` for the UNIX 98 threads API library `libpthread.so` to the link process (unless suppressed by another option) and at least the search paths like `/opt/pthread/lib[64s]` for thread-safe system libraries.

Note:

Check your use of `LD_RUN_PATH` and `LD_LIBRARY_PATH` in the compilation and runtime environment to avoid unintended interference between thread-safe and thread-unsafe libraries!

Startup routines

If `-K pthread` is given, the compiler automatically adds specific startup routines to the executable. These perform proper initialization of the UNIX 98 threads implementation in Reliant UNIX 5.45B calling `cma_init()` prior to calling `main()`.

Module *)	If ...
<code>/opt/CDS++/lib[64s]/pthr_c.o</code>	cc command without <code>-p</code> option
<code>/opt/CDS++/lib[64s]/pthr_mc.o</code>	cc command with <code>-p</code> option
<code>/opt/CDS++/lib[64s]/pthr.o</code>	CC command without <code>-p</code> option
<code>/opt/CDS++/lib[64s]/pthr_m.o</code>	CC command with <code>-p</code> option
<code>/opt/CDS++/lib[64s]/pthr_cf.o</code>	CC command with <code>-Xd</code> options
<code>/opt/CDS++/lib[64s]/pthr_mcf.o</code>	CC command with <code>-Xd</code> options

*) These module names and their location are not an external user interface of the Reliant UNIX C/C++ compiler.

The UNIX 98 threads implementation in Reliant UNIX 5.45B provides the following functions for its initialization in the startup modules. They will be called in the order given below and *before* any other UNIX 98 threads routine is used. Use of these functions outside of the startup modules is undefined.

```
void cma_init(void);
```

Initializes the UNIX 98 threads library `libpthread.so`.

```
void set_pthread_symb(void);
```

Sets up various callbacks from `libc_r.so` back into `libpthread.so`. These are needed to make the standard C library thread-safe.

```
void libc_r_init(void);
```

Initializes the thread-safe library `libc_r.so`.

```
void _networking_init(void);
```


Initializes the thread-safe libraries `libsocket.so` and `libnsl.so`.

Linking

For compiling and linking of UNIX 98 threads applications with CDS++ 2.0C, the compiler's `-K pthread` option *must* be used.

Note:

The specification of `-lpthread` option alone via the compile/link command line is not sufficient to build a UNIX 98 threads application or library correctly.

The compiler's `-K pthread` option implies several additional options within the compile/link command line (see previous sections). Because the order of the command line options is significant for correct building, you should take into account the following hints to avoid unexpected application behavior.

`-K pthread` does not support "static linking", do not use the `-dn` option.

Note:

If the `-K pthread` option is used, the `-dy` option ("dynamic linking") will be set implicitly. It ensures that the corresponding archive (`*.a`) will be used, if a library does not exist as a shared object (`*.so`).

Do not link the system libraries `libpthread`, `libc`, `libCstd`, or `libCrun` explicitly. These libraries will be included (as necessary) by the `-K pthread` option in the correct sequence.

Note:

Simultaneous use of the C++ runtime libraries `libCrun` and `libCFrun` in multi-threaded applications is not defined.

If you build a library using the `-G` option, the link option `-l` will not be set implicitly by the compiler. If you specify `-G`, you should not use the `-llibrary` option to avoid *library* being included in the "needed list" of the target library. In this way the target library becomes more universal and more flexible. You can specify any necessary libraries with the `-l` option later on when you build the executable `a.out`.

For several system libraries/archives (e.g. `libm`) there is a corresponding thread-safe counterpart with `"_r"` suffix (e.g. `libm_r`). Do not specify this suffix in a `-l` option. The `-K pthread` option implicitly uses the corresponding thread-safe library with `"_r"` suffix (if available).

You should avoid the `-Lpath` option because this can impact search paths which are implicitly set by `-K pthread`.

Profiling

Profiling requires relinking of the threaded application with the `-p` compiler option. The startup code starts a separate thread that collects the profile data into a text file named `mon.program` in the current working directory. Function call counts are not available. The created file `mon.out` is not useful.

```
% cc -p [-D_REENTRANT] -K pthread -o program source ...
```

Debugging

The symbolic debugger `dbx` that comes with the Reliant UNIX C Compiler CDS++ 2.0C is aware of UNIX 98 threads and offers functionality to debug individual threads. See [C/C++] for details.

The UNIX 98 threads implementation in Reliant UNIX 5.45B offers the following functions in `libcma_ext.so` to `dbx` to display thread states. The declaration and explanation of most of the arguments is given in `cma_deb_ext.h` (this file is not part of the UNIX 98 threads package). Use of these functions outside of `dbx` is undefined.

```
#include <cma_deb_ext.h>;
void cma_init_debug(deb_function_vector_t *deb_fv_p,
    cma_function_vector_t **a_cma_fv_p);
```

Sets up the use of threads debugging information via the mutual function pointers in the arguments. The debugger will pass a pointer to its function vector `deb_fv_p`, through which the UNIX 98 threads implementation will access the debugger specific routines. It will itself return the pointer `a_cma_fv_p` to its function vector for the external debugger to use. These functions are:

```
#include <dce/cma.h>;
```

```
void cma_debug_cmd(char *cmd, ...);
```

Parse commands in *cmd* and dispatches to execution code.

```
#include <cma_deb_ext.h>;
int cma__deb_getcontext(unsigned int thread_num,
    cma_t_deb_thread_context *context_p);
```

Gets the context of the thread with the given sequence number *thread_num* into the area pointed to by *context_p* in order to switch to another thread. This routine cannot be called for the current thread (i.e. the thread which took a breakpoint).

```
unsigned int cma__deb_get_self_sequence(void);
```

Returns the sequence number of the currently running thread.

```
int cma__deb_is_thread_valid(unsigned int thread_num);
```

Verifies whether the given thread sequence number *thread_num* refers to a known thread no matter what state that thread is in.

```
unsigned int cma__deb_pid_to_tid(pid_t pid);
```

Maps the clone's process ID *pid* to the corresponding thread ID.

In addition, dbx might use the following symbols to check their existence or retrieve their address (but not the data):

```
cma_g_def_tcb          /* TCB of main thread */
cma_g_known_threads    /* List of all threads */
cma__transfer_thread_ctx /* not available */
cma__transfer_thread_ctx_pc /* not available */
pthread_mutexattr_default /* Default mutex attribute */
```

C++ runtime system issues

UNIX 98 threads library takes care of the C++ stack unwinding mechanism during the thread termination as a result of `pthread_cancel()` or `pthread_exit()`. For this reason the application does not need to use the C++ routine `_SNI_extensions::unwind_exit()` in order to invoke C++ destructor and delete routines.

When the termination request is acted upon, the C++ destructor routines and C cancellation cleanup handlers are invoked in the following sequence:

1. First, all routines in the C++ destructor list are invoked one by one in LIFO sequence. When the last C++ destructor routine returns, the corresponding allocated resources are released. Thus all references to them are no more valid.

Note:

If `pthread_cleanup_push()` and `pthread_cleanup_pop()` are used within C++ in order to install a cancellation cleanup handler, the cancellation cleanup handler will be treated as a C++ destructor routine; that is, it will be added to the C++ destructor list and will be invoked as a typical C++ destructor routine when termination is requested.

2. Afterwards all the C cancellation cleanup handlers, which are installed within C by using `pthread_cleanup_push()` and `pthread_cleanup_pop()`, are invoked one by one in LIFO sequence. When the last cancellation cleanup handler returns, thread execution is terminated.

Note:

Because of the fixed calling sequence of C++ destructor routines and C cancellation cleanup handlers, you have to ensure the following:

Do not use references to resources – which are allocated in C++ – within a cancellation cleanup handler which is installed within C, because these references are no longer valid when this cleanup handler is invoked.

You can avoid this situation, if you compile the C source as a C++ source.

To support threads cancellation cleanup, UNIX 98 threads and the C/C++ runtime system offer each other functions as follows. The UNIX 98 threads library offers

```
void __pthread_exit(void *status);
```

The C/C++ runtime system offers

```
void __thread_unwind(void);
```

UNIX 98 threads API

In the following sections, the notation of function arguments in the form

type name = NULL

indicates that the value NULL may safely be used instead of *name*.

UNIX 98 threads objects

UNIX 98 threads specifies these objects (data types) in `<sys/types.h>`:

UNIX 98 threads object	Description
pthread_t	Thread ID
pthread_attr_t	Threads attribute object
pthread_cond_t	Condition variable
pthread_condattr_t	Condition variable attribute object
pthread_key_t	Thread Specific Data (TSD) key
pthread_once_t	Once-only execution
pthread_mutex_t	Mutual exclusion lock
pthread_mutexattr_t	Mutual exclusion lock attribute object
pthread_rwlock_t	Read/write lock
pthread_rwlockattr_t	Read/write lock attribute

These data types are opaque for the programmer using UNIX 98 threads. With the exception of `pthread_t`, there are no defined comparison or assignment operators for these data types.

Thread creation attributes

UNIX 98 threads attributes are collected into an attribute object defined by data type `pthread_attr_t`. An attribute is initialized and destroyed by the functions

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

All thread attributes in an attribute object are set by a function in the form

```
int pthread_attr_setName(pthread_attr_t *attr, Type t);
```

and retrieved by a function in the form

```
int pthread_attr_getName(const pthread_attr_t *attr, Type *t);
```

where *Name* and *Type* are from the table below.

Type and name (controlled by symbolic constant)	Value(s) with default
int detachstate <code>□</code> _POSIX_THREADS	PTHREAD_CREATE_DETACHED, <code>□</code> PTHREAD_CREATE_JOINABLE

<code>size_t guardsize</code> $\square$ <code>_POSIX_THREADS</code>	$\geq$ <code>PAGESIZE</code>
<code>size_t stacksize</code> $\square$ <code>_POSIX_THREAD_ATTR_STACKSIZE</code>	$\geq$ <code>PTHREAD_STACK_MIN</code>
<code>void *stackaddr</code> $\square$ <code>_POSIX_THREAD_ATTR_STACKADDR</code>	valid address
<code>struct sched_param schedparam</code> $\square$ <code>_POSIX_THREADS</code>	≥ 0
<code>int contentionscope</code> $\square$ <code>_POSIX_THREAD_PRIORITY_SCHEDULING</code>	<code>PTHREAD_SCOPE_SYSTEM</code> , $\square$ <code>PTHREAD_SCOPE_PROCESS</code>
<code>int inheritsched</code> $\square$ <code>_POSIX_THREAD_PRIORITY_SCHEDULING</code>	<code>PTHREAD_INHERIT_SCHED</code> , $\square$ <code>PTHREAD_EXPLICIT_SCHED</code>
<code>int schedpolicy</code> $\square$ <code>_POSIX_THREAD_PRIORITY_SCHEDULING</code>	<code>SCHED_FIFO</code> , <code>SCHED_RR</code> , $\square$ <code>SCHED_OTHER</code>

The UNIX 98 threads implementation in Reliant UNIX 5.45B does not define `_POSIX_THREAD_PRIORITY_SCHEDULING`. Thus the corresponding attributes `contentionscope`, `inheritsched`, and `schedpolicy` have no default value defined and the associated attribute access functions will return `ENOSYS`. However, accessing the `schedparam` attribute is supported but without effect.

If an application is managing its own thread stacks using the `stackaddr` attribute, the `guardsize` attribute is ignored and no stack overflow protection is provided. In this case, it is the responsibility of the application to manage stack overflow along with stack allocation.

Thread creation, termination, and scheduling

This includes the creation, control, and termination of multiple flows of control (threads) in the same process under the assumption of a common, shared address space.

```
int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr = NULL,
    void *(*routine)(void *), void *arg);
```

Creates a new *thread* with attributes *attr*, executing *routine* with *arg*.

Note:

There is a limit of `PTHREAD_THREADS_MAX` threads per process.

```
int pthread_detach(pthread_t thread);
```

Sets the attribute *detachstate* of the specified thread to `PTHREAD_CREATE_DETACHED`.

```
pthread_t pthread_self(void);
```

Returns the thread ID of the calling thread.

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

Compares thread IDs *t1* and *t2* for equality.

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

Terminates the calling thread (does not return). *status* will be returned to a thread synchronizing with the terminating thread.

```
int pthread_join(pthread_t thread,
    void **status = NULL);
```

Synchronizes with (waits for) the termination of the specified *thread* getting its termination *status*.

```
int pthread_atfork(void (*prepare)(void) = NULL,
    void (*parent)(void) = NULL,
    void (*child)(void) = NULL);
```

Registers functions to be called during `fork()` execution. *prepare* functions are called before `fork()` in reverse order of registration (LIFO), *parent* and *child* functions are called in the parent and child respectively in order of registration (FIFO).

The following two functions allow an application to express and get the intended level of concurrency in threads execution. The UNIX 98 threads implementation in Reliant UNIX 5.45B does not multiplex user threads to a smaller number of kernel execution entities, instead each thread is an individual kernel scheduled entity and concurrency of thread execution is always enabled at maximum level (limited by the number of processors in the system). Thus both functions have no effect except setting and returning an `int`.

```
int pthread_getconcurrency(void);
```

Get the *level* of concurrency expressed by a previous call.

```
int pthread_setconcurrency(int level);
```

Express the *level* of concurrency desired by the application.

The following two functions are subject to definition of `_POSIX_THREAD_PRIORITY_SCHEDULING`. Since the UNIX 98 threads implementation in Reliant UNIX 5.45B does not support this feature, they will always return `ENOSYS`.

```
#include <sched.h>
int pthread_getschedparam(pthread_t thread,
    int *policy, struct sched_param *parameters);
```

Get the scheduling *policy* and *parameters* of the specified *thread*.

```
#include <sched.h>
int pthread_setschedparam(pthread_t thread,
    int *policy, const struct sched_param *parameters);
```

Set the scheduling *policy* and *parameters* of the specified *thread*.

Thread execution scheduling

In the UNIX 98 threads implementation in Reliant UNIX 5.45B, threads are scheduled only by the kernel according to the scheduling attributes maintained by the kernel for each thread. These are equivalent to the scheduling attributes of Reliant UNIX processes and subject to their scheduling mechanisms and policies.

Thread cancellation

The thread cancellation mechanism allows a thread to terminate the execution of any other thread in the process in a controlled manner. Each thread maintains its own cancelability state.

Cancelability state

When cancelability is `PTHREAD_CANCEL_DISABLE`, cancellation requests against the target thread are held pending. By default, cancelability is set to `PTHREAD_CANCEL_ENABLE`.

Cancelability type

When cancelability is enabled and the cancelability type is `PTHREAD_CANCEL_ASYNCHRONOUS`, new or pending cancellation requests may be acted upon at any time. When cancelability is enabled and the cancelability type is `PTHREAD_CANCEL_DEFERRED`, cancellation requests are held pending until a cancellation point (see the paragraph **Cancellation points** below) is reached. The setting of the cancelability type has no immediate effect until cancelability is (re)enabled. The cancelability type is `PTHREAD_CANCEL_DEFERRED` in all newly created threads including the main thread.

A target thread is allowed to hold cancellation requests pending and to perform application-specific cleanup processing when the notice of cancellation is acted upon.

```
int pthread_cancel(pthread_t thread);
```

Places a cancellation request in the specified *thread*.

Note:

Canceled threads terminate with a status of `PTHREAD_CANCELED`.

```
void pthread_testcancel(void);
```

Introduces a cancellation point in the calling thread.

Note:

This function is a cancellation point.

```
int pthread_setcancelstate(int state, int *oldstate);
```

Sets the cancellation *state* of the calling thread to one of `PTHREAD_CANCEL_ENABLE` (the default for each newly created thread) or `PTHREAD_CANCEL_DISABLE` and returns the old state in *oldstate*.

```
int pthread_setcanceltype(int type, int *oldtype);
```

Sets the cancellation *type* of the calling thread to one of `PTHREAD_CANCEL_DEFERRED` (the default for each newly created thread) or `PTHREAD_CANCEL_ASYNCHRONOUS` and return the old type in *oldtype*.

```
void pthread_cleanup_push(void (*routine)(void*),  
    void *arg);
```

Pushes *routine* and *arg* onto the calling thread's cancellation stack.

Note:

Push and pop operations must appear at the same lexical level (same level of { and } in C language).

```
void pthread_cleanup_pop(int execute);
```

Pops the top item from the calling thread's cancellation stack and execute it if *execute* is nonzero.

Note:

The macros `pthread_cleanup_push()` and `pthread_cleanup_pop()` are/must be defined different for C and C++ (see also the section **C/C++ runtime system issues**).

Applications must carefully follow static lexical scoping rules in their execution behavior. For instance, use of `setjmp()`, `return`, `goto`, and so on, to leave user-defined cancellation scopes without doing the necessary scope pop operation will result in undefined behavior.

Use of asynchronous cancelability while holding resources which potentially need to be released may result in resource loss and even dead lock. Similarly, cancellation scopes may only be safely manipulated (pushed and popped) when the thread is in the *deferred* or *disabled* cancelability states.

Cancellation points

Cancellation points are points inside a function where a thread must act on any pending cancellation request when cancelability is enabled *if the function would block* (indefinitely). Every function that calls a function defined to contain cancellation points becomes itself a cancellation point.

UNIX 98 threads define cancellation points to occur when a thread is executing the following threads API functions:

Functions defined as cancellation points	
<code>pthread_cond_timedwait()</code>	<code>pthread_cond_wait()</code>
<code>pthread_join()</code>	<code>pthread_testcancel()</code>
<code>sigwait()</code>	

Note: □

`pthread_mutex_lock()` is not a cancellation point. For more functions including cancellation points see also `pthread_syslib_intro(3T)`.

Async-cancel safety

A target thread may disable cancellation altogether or reset it to *asynchronous cancelability*. During the period when asynchronous cancellation is enabled, the target thread must only execute code that does not modify global state and may only call routines that are explicitly stated as *async-cancel safe*.

A function is said to be *async-cancel safe* if it is written in a way that entering it from a thread with asynchronous cancelability enabled will not cause any global state to be violated. UNIX 98 threads define the following

functions from the threads API to be async-cancel safe.

Async-cancel safe functions	
pthread_cancel()	pthread_setcancelstate()
pthread_setcanceltype()	

Threads synchronization objects

Mutexes

Attributes of mutexes are collected into a mutex attribute object defined by data type `pthread_mutexattr_t`. A mutex attribute object is initialized and destroyed by the functions

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

All mutex attributes are set in a mutex attribute object by a function in the form

```
int pthread_mutexattr_setName(
    pthread_mutexattr_t *attr, Type t);
```

and retrieved by a function in the form

```
int pthread_mutexattr_getName(
    const pthread_mutexattr_t *attr, Type *t);
```

where *Name* and *Type* are from the table below.

Type and name (controlled by symbolic constant)	Value(s) with default
int pshared $\square$ _POSIX_THREAD_PROCESS_SHARED	PTHREAD_PROCESS_SHARED, PTHREAD_PROCESS_PRIVATE
int type	PTHREAD_MUTEX_ERRORCHECK, PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_RECURSIVE, PTHREAD_MUTEX_DEFAULT
int prioceiling $\square$ _POSIX_THREAD_PRIO_PROTECT	≥ 0
int protocol $\square$ _POSIX_THREAD_PRIO_PROTECT $\square$ _POSIX_THREAD_PRIO_INHERIT	PTHREAD_PRIO_NONE, PTHREAD_PRIO_PROTECT, PTHREAD_PRIO_INHERIT

The UNIX 98 threads implementation in Reliant UNIX 5.45B neither defines `_POSIX_THREAD_PRIO_PROTECT` nor `_POSIX_THREAD_PRIO_INHERIT`. Thus, the attributes `prioceiling` and `protocol` have no default value defined and the corresponding attribute access functions return `ENOSYS`.

Mutexes are synchronization variables that allow locking of critical code sections against simultaneous execution by more than one thread to avoid data races. Only one thread can lock (acquire) a mutex and proceed. Other threads will wait synchronously until a locked mutex is unlocked (released) by the thread that owns it.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr = NULL);
```

Initializes the indicated *mutex* with attributes *attr* or default attributes.

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

Destroys the indicated *mutex*.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Acquires the indicated *mutex* (waiting for it if necessary).

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

Tries to acquire the indicated *mutex* (never waiting for it).

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

Releases the (previously acquired) indicated *mutex*.

The following two functions are subject to definition of `_POSIX_THREAD_PRIO_PROTECT`. Since Reliant UNIX 5.45B does not support this feature, they will always return `ENOSYS`.

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

Get the *prioceiling* value of the indicated *mutex*.

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

Set the *prioceiling* value of the indicated *mutex* and return the old value in *oldprioceiling*.

Condition variables

Attributes of condition variables are collected into a condition variable attribute object defined by data type `pthread_condattr_t`. A condition variable attribute object is initialized and destroyed by the functions

```
int pthread_condattr_init(pthread_condattr_t *attr);
```

```
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

All attributes in a condition variable attribute object are set by a function in the form

```
int pthread_condattr_setName(
    pthread_condattr_t *attr, Type t);
```

and retrieved by a function in the form

```
int pthread_condattr_getName(
    const pthread_condattr_t *attr, Type *t);
```

where *Name* and *Type* are from the table below.

Type and name (controlled by symbolic constant)	Value(s) with <i>default</i>
<code>int pshared</code> ☐ <code>_POSIX_THREAD_PROCESS_SHARED</code>	<code>PTHREAD_PROCESS_SHARED</code> , <code>PTHREAD_PROCESS_PRIVATE</code>

Condition variables, in conjunction with mutexes, allow threads to wait for some arbitrary condition to occur, i.e. to be signaled on this condition variable. Condition variables are stateless in the sense that signals on them are not remembered if no thread is waiting.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr = NULL);
```

Initializes a condition variable *cond* with attributes *attr* or default attributes.

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

Destroys the condition variable *cond*.

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

Unblocks at least one thread currently blocked in the condition variable *cond*.

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


Unblocks all threads currently blocked in the condition variable *cond*.

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

Blocks on the condition variable *cond* protected by *mutex*.

Note:

This function is a cancellation point.

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

Blocks on the condition variable *cond* protected by *mutex* no longer than the specified *abstime*.

Note:

This function is a cancellation point.

Thread specific data

The thread-specific data (TSD) interface can be viewed as a two-dimensional array of values with *keys* serving as row index and thread IDs as the column index. Thus the keys are shared among all the threads while each thread has an individual *value* (usually a pointer) associated with each key.

```
int pthread_key_create(pthread_key_t *key,
    void (*destructor)(void *) = NULL);
```

Creates a data *key* and a *destructor* function associated with that key for the calling thread. The *destructor* function is called with the value associated with the key as argument upon thread exit.

Note:

There is the limit `PTHREAD_KEY_MAX` of keys per process and `PTHREAD_DESTRUCTOR_ITERATIONS` of iterative calls to *destructor* per thread exit.

```
int pthread_key_delete(pthread_key_t key);
```

Destroys the *key* and removes the associated values in all threads (but does not call any associated destructor function).

```
void *pthread_getspecific(pthread_key_t key);
```

Returns the value associated with *key* in the calling thread (or `NULL`).

```
int pthread_setspecific(pthread_key_t key,
    const void *value);
```

Sets *value* associated with *key* in the calling thread.

Read/write locks

Attributes of read/write locks are collected into a read/write lock attribute object defined by data type `pthread_rwlockattr_t`. A read/write lock attribute object is initialized and destroyed by the functions

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(
    pthread_rwlockattr_t *attr);
```

All read/write lock attributes are set in a read/write lock attribute object by a function in the form

```
int pthread_rwlockattr_setName(
    pthread_rwlockattr_t *attr, Type t);
```

and retrieved by a function in the form

```
int pthread_rwlockattr_getName(
    const pthread_rwlockattr_t *attr, Type *t);
```

where *Name* and *Type* are from the table below.

Type and name	Value(s) with default

(controlled by symbolic constant)	
<pre>int pshared $\square$ _POSIX_THREAD_PROCESS_SHARED</pre>	<pre>PTHREAD_PROCESS_SHARED, PTHREAD_PROCESS_PRIVATE</pre>

Multiple readers, single writer locks (read/write locks) allow many threads to simultaneously read data while allowing only one thread to write at any given time. Read/write locks can be used to synchronize threads in the current process and other processes if they are allocated in memory that is writable and shared among the cooperating processes and have been initialized for this behavior. The acquisition order is undefined, the UNIX 98 threads implementation in Reliant UNIX 5.45B favors writers over readers.

```
pthread_rwlock_t mutex = PTHREAD_RWLOCK_INITIALIZER;
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
    const pthread_rwlockattr_t *attr = NULL);
```

Initializes read/write lock *rwlock* with attributes *attr* or default attributes.

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Destroys the indicated read/write *rwlock*.

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

Acquires the indicated read/write *rwlock* for reading, possibly waiting until a thread holding the lock for writing releases it.

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

Tries to acquire the indicated read/write *rwlock* for reading.

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

.fi

Acquires the indicated read/write *rwlock* for writing, possibly waiting until a thread holding the lock for reading or writing releases it.

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

Tries to acquire the indicated read/write *rwlock* for writing.

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Unlocks the indicated read/write *rwlock*.

Once-only execution

```
pthread_once_t once = PTHREAD_ONCE_INIT;
```

Initialize the once-control variable *once*.

```
int pthread_once(pthread_once_t *once_control,
    void (*init_routine)(void));
```

Execute *init_routine* exactly once.

Non-portable extensions

The UNIX 98 threads implementation in Reliant UNIX 5.45B specifies the following non-portable extensions.

Non-portable means they do not exist in [UNIX 98].

```
unsigned int pthread_getunique_np(pthread_t thread);
```

Returns the sequence number from *thread* for printing and logging purposes. It is an integer from the range [0, 65535]. This function is implemented as a macro.

Thread-safe system libraries

The UNIX 98 threads implementation in Reliant UNIX 5.45B supplies system libraries with thread-safe system functions. The program development tool sets provide options that select the appropriate set of system libraries at compile and runtime. See also [pthread_syslib_intro\(3T\)](#).

Using thread-unsafe libraries

A multi-threaded UNIX 98 threads application might use thread-unsafe functions or libraries if it controls the calls in a way that avoids the issues as explained in the section **Thread-safeness and reentrance**. This can be achieved by always and only calling those functions from the main thread. Otherwise the behavior is undefined.

Thread-safeness and reentrance

A function may be "reentered" by several mechanisms: by calls from multiple threads, by a call from of an asynchronous signal handler that interrupted the function, by direct or indirect recursion.

Thread-safe indicates reentrance with respect to simultaneous calls by multiple threads. A function that is reentrant with respect to asynchronous signals is *async-signal safe*.

Data races

A program has a data race if it is possible for a thread to modify an addressable location at the same time that other threads are accessing the same location. Reading the same location in memory concurrently, it is not a data race. Programs and thread-safe functions must avoid data races by using the synchronization primitives of UNIX 98 threads. However, thread-safe functions do not protect access to memory under control of the caller. For example, the calling program must avoid the data race produced by two simultaneous calls to `memcpy(3C)` with overlapping destination areas.

Threads and process resources

Each thread has only a small individual state: the register set, program counter, stack pointer, and signal mask, the `errno` variable and the thread-specific data (if any). All other resources of a process are shared by all the threads.

Address space

Static data are accessible to all threads and require appropriate protection against data races.

File descriptors

Concurrent access to the same file by multiple threads (for example, reading the file record by record) must control the file offset before each access atomically.

Timers

Timers are process-global. The corresponding expiration signal is sent to the process, not a particular thread. Thus, threads need to cooperate to set per-thread timers.

Signal handlers

Are discussed below in a separate section.

Environment

Arguments (`argv`) and environment (`environ`) are part of the address space and need protection against data races.

Note:

The function `getenv(3C)` is thread-unsafe but does not have a `getenv_r` variant.

User ID and working directory

Changing the user ID or working directory in one thread might lead to denial of access to parts of the file system in other threads.

The `errno` variable

Usage of a thread-aware `<errno.h>` in any multi-threaded program is required to coordinate the access of the threads to the global `errno` value. It transforms all references to `errno` into `*cma_errno()`. `cma_errno` returns `&errno` in the main thread and a TSD cell otherwise. Thus, the main thread in the process actually uses `errno`. This allows to use thread-unaware libraries or modules using `errno` in the main thread.

Serializability

Most simultaneous calls to thread-safe functions appear to be "atomically", i.e. as if executed in a (arbitrary) serial order. However, this need not be the case. For example, a call to a function *f* that adds two items *A* and *B* to a list, and another simultaneous call that adds *C* and *D* may produce a list in the order *A, C, B, D*. This is not possible by serial execution of the two calls or if the add operation is defined to be atomically. Thread-safe

functions should specify any behaviors that are not serializable.

The standard I/O functions like `printf(3S)` are individually thread-safe and simultaneous calls using the same `FILE` object are serialized. However, the calling application or library has to make a series of such calls atomic and to protect the `FILE` object against a simultaneous `write(2)` to the underlying file.

Functions reading *successive* entries in a file are usually not thread-safe since information about which entry was last read is not maintained on a per-thread basis. Consequently, the application program must synchronize the use of these functions.

Synchronization strategies and deadlock avoidance

Threads' synchronization strategies must avoid data races. They vary in the degree to which they allow threads to overlap operation.

Structured code and data locking

Structured *code* locking assumes that shared data is only accessed by a set of specific functions. Typically, there is a single mutex ("monitor lock") associated with this set of functions that is acquired upon entering any of them and released on return. Thus only one thread at a time can be executing any of the functions.

Structured *data* locking assumes that shared data is only accessed by a set of specific functions *and* can be grouped into disjoint objects sharing no state. It associates a mutex ("object lock") directly with each object that is acquired whenever the object is manipulated. Operations on different objects can proceed concurrently.

A monitor or object lock needs to be released when nesting of these locks (needing another while already holding one), reentrant operations (reacquiring a lock already held), long operations (I/O to read in object data) or calls to functions of unknown locking behavior (outside of the current context) occur. This is necessary to enable concurrency and to avoid deadlocks. The shared data must be consistent when the lock is released and reevaluated before proceeding when the lock is reacquired.

Deadlock, lock ordering, resource deadlocks

There are various dangers of deadlock associated with synchronization issues. *Self*-deadlock or *recursive* deadlock occurs when a thread tries to acquire a lock that it already owns. *Nested* deadlock occurs where a thread holding the monitor lock of function set *A* blocks on function set *B*, but the path to release *B* goes through *A*. All of these can be avoided by releasing a monitor or object lock before calling outside functions.

Lock-ordering deadlock occurs where a thread holds *As* lock and tries to acquire *Bs* lock while another thread holds *Bs* lock and tries to acquire *As* lock. This can be avoided by defining a lock order or lock hierarchy in which locks are acquired.

Resource deadlock occurs where a thread needs a second resource of the same type to proceed but all such resources are already allocated as first resource by other threads and none is willing to give one up. The solution is to first trying to acquire all resources needed to progress and if that is not possible give them all up and restart.

Signal handling

Applications of UNIX 98 threads must handle signals differently from standard non-threaded Reliant UNIX applications. Signal masks are maintained on a per-thread basis. The action to be taken upon delivery of a signal is specified on a process-wide basis: default, ignore, execute signal catching function.

A signal is considered *generated* for a process or thread as a result of some event. It is considered *delivered* when the assigned signal action is taken. In the interval between generation and delivery, the signal is *pending*. A signal can be *blocked (masked)* from delivery via a signal mask.

There are *synchronously generated* signals (traps), generated by thread-internal events (illegal memory access, invalid instructions, etc.) and *asynchronously generated* signals (interrupts) generated by process-external events (terminal activity and timers) or via `kill()`.

Note:

Synchronous and asynchronous is a property of how the signal was generated and not a property of the signal number. All signals may be generated asynchronously.

Traps are always delivered to the triggering thread, interrupts are delivered once to the process where any (at

most one) thread that has the signal unmasked may service it. The choice of thread is unspecified. Whenever a signal with a specified signal action of terminate, stop, or continue is delivered to a thread, this signal action applies to the entire process.

```
#include <signal.h>
int pthread_sigmask(int how,
    const sigset_t *newmask = NULL,
    sigset_t *oldmask = NULL);
```

Examine or change the calling thread's signal mask.

```
#include <signal.h>
int pthread_kill(pthread_t thread, int signo);
```

Deliver signal *signo* to the indicated *thread*.

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
```

Synchronously awaits a signal from *set* and returns it in *sig*.

Note:

This function is a cancellation point. Using `sigwait(3T)` and `sigaction(2)` concurrently within a process on the same signal is unspecified.

Use of signal management functions `signal(2)`, `sighold(2)`, `sigignore(2)`, `sigpause(2)`, `sigprocmask(2)`, `sigrelse(2)`, and `sigset(2)` is unspecified in multi-threaded processes.

Async-signal safety

Code that runs with signals enabled must be carefully controlled with respect to potential interference with code executed from signal handlers. Only *async-signal safe* functions shall be called from signal handlers. A safe strategy is to await signals in a separate thread via `sigwait()` because this executes signal handling in the usual threads context.

A function is called *async-signal safe* if it may be called, without restrictions, from signal-catching functions invoked asynchronously with threads execution. All functions should be considered to be unsafe with respect to signals unless explicitly declared as *async-signal safe*. In the presence of signals, all functions will behave as defined when called from or interrupted by a signal-catching function, with a single exception: When a signal interrupts an unsafe function and the signal-catching function calls an unsafe function, the behavior is undefined.

UNIX 98 threads do not declare any function of its API to be *async-signal safe*. UNIX 98 threads functions cannot safely be called in a signal handler. However there are *async-signal safe* system and network interface. See `pthread_syslib_intro(3T)`.

RETURN VALUE

All **[POSIX.1c]** functions [except `pthread_exit()`, `pthread_getspecific()`, and `pthread_self()`] return zero (0) for success or an `errno` value if the operation fails.

FILES

The UNIX 98 threads implementation in Reliant UNIX 5.45B resides in the following files:

```
/usr/include/pthread.h
    UNIX 98 threads include file
/opt/pthread/lib[64s]/
    32-bit and 64-bit UNIX 98 threads libraries
/opt/pthread/readme/*
    Release note
```

LITERATURE

The following literature contains specific details and background:

[C/C++]

C/C++ Compiler V2.0B (Reliant UNIX), User Guide, Fujitsu Siemens Computers GmbH, Order No. U25438-J-Z145-3-76

[GO SOLO 2]

Go Solo 2 – The Authorized Guide to Version 2 of the Single UNIX Specification, The Open Group, May 1997, Document Number X909P, ISBN: 0-13-575589-8

[POSIX.1]

Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application Programming Interface (API) [C Language], ISO/IEC 9945-1: 1996, ANSI/IEEE Std. 1003.1, 1996 Edition

[POSIX.1c]

Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application Programming Interface (API) – Amendment 2: Threads Extension [C Language], IEEE Std. 1003.1c-1995

[UNIX 98]

The Single UNIX Specification, Version 2, The Open Group, 1997

[XSH]

CAE Specification, System Interfaces and Headers, Issue 5, The Open Group, January 1997, Document Number C606, ISBN: 1-85912-181-0

SEE ALSO

All manual pages listed by the apropos pthread command and the following:

sigprocmask(2), ctime(3C), directory(3C), getgrent(3C), getlogin(3C), getpwent(3C), rand(3C), string(3C), sysconf(3C), ttyname(3C), flockfile(3T), getc_unlocked(3T), sched_yield(3T), sigwait(3T), sched(5), types(5).

pthread_join(3-thr)**NAME**

`pthread_join` - causes the calling thread to wait for the termination of a specified thread

SYNOPSIS

```
#include <pthread.h>
int pthread_join(pthread_t thread, pthread_addr_t *status);
```

PARAMETERS

thread

Thread whose termination is awaited by the caller of this routine.

status

Status value of the terminating thread when that thread calls `pthread_exit()`.

DESCRIPTION

The `pthread_join()` routine causes the calling thread to wait for the termination of a specified thread. A call to this routine returns after the specified thread has terminated.

Any number of threads can call this routine. All threads are awakened when the specified thread terminates.

If the current thread calls this routine to join with itself, an error is returned.

The results of this routine are unpredictable if the value for *thread* refers to a thread object that no longer exists.

RETURN VALUES

If the thread terminates normally, the exit status is the value that is optionally returned from the thread's start routine.

If the function fails, `errno` may be set to one of the following values:

`EINVAL`

The value specified by *thread* is invalid.

`ESRCH`

The value specified by *thread* does not refer to a currently existing thread.

`EDEADLK`

A deadlock is detected.

SEE ALSO

`pthread_create(3-thr)`, `pthread_detach(3-thr)`, `pthread_exit(3-thr)`.

pthread_key_create(3T)**NAME**

pthread_key_create - thread-specific data key creation

SYNOPSIS

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key, void
(*destructor)(void*));
```

DESCRIPTION

This 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. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

If, after all the destructors have been called for all non-`NULL` values with associated destructors, there are still some non-`NULL` values with associated destructors, then 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 non-`NULL` values with associated destructors, implementations may stop calling destructors, or they may continue calling destructors until no non-`NULL` values with associated destructors exist, even though this might result in an infinite loop.

RETURN VALUE

If successful, the `pthread_key_create()` function stores the newly created key value at `*key` and returns zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_key_create()` function 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()` function will not return an error code of `EINTR`.

SEE ALSO

[pthread_getspecific\(3T\)](#), [pthread_intro\(3T\)](#), [pthread_key_delete\(3T\)](#), [pthread_setspecific\(3T\)](#), [pthread\(5\)](#).

pthread_key_delete(3T)**NAME**

pthread_key_delete - thread-specific data key deletion

SYNOPSIS

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key);
```

DESCRIPTION

This 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 VALUE

If successful, the `pthread_key_delete()` function returns zero. 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`.

SEE ALSO

[pthread_intro\(3T\)](#), [pthread_key_create\(3T\)](#), [pthread\(5\)](#).

pthread_kill(3T)**NAME**

`pthread_kill` - send a signal to a thread

SYNOPSIS

```
#include <signal.h>
int pthread_kill(pthread_t thread, int sig);
```

DESCRIPTION

The `pthread_kill()` function is used to request that a signal be delivered to the specified thread.

As in *kill*, if *sig* is zero, error checking is performed but no signal is actually sent.

RETURN VALUE

Upon successful completion, the function returns a value of zero. 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.

The `pthread_kill()` function will not return an error code of `EINTR`.

SEE ALSO

`kill(2)`, `raise(3C)`, `pthread_intro(3T)`, `pthread_self(3T)`, `pthread(5)`.

pthread_mutexattr_getpshared(3T)**NAME**

pthread_mutexattr_getpshared, pthread_mutexattr_setpshared - set and get process-shared attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr,
    int *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 initialised 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 initialised the mutex; if threads of differing processes attempt to operate on such a mutex, the behaviour is undefined.

The default value of the attribute is PTHREAD_PROCESS_PRIVATE.

RETURN VALUE

Upon successful completion, pthread_mutexattr_setpshared() returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, pthread_mutexattr_getpshared() returns zero 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_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.

These functions will not return an error code of EINTR.

SEE ALSO

pthread_cond_init(3T), pthread_create(3T), [pthread_intro\(3T\)](#), [pthread_mutexattr_init\(3T\)](#), pthread_mutex_init(3T), [pthread\(5\)](#).

pthread_mutexattr_init(3T)**NAME**

`pthread_mutexattr_init`, `pthread_mutexattr_destroy` - initialise and destroy mutex attributes object

SYNOPSIS

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);□
```

DESCRIPTION

The function `pthread_mutexattr_init()` initialises a mutex attributes object *attr* with the default value for all of the attributes defined by the implementation.

The effect of initialising an already initialised mutex attributes object is undefined.

After a mutex attributes object has been used to initialise one or more mutexes, any function affecting the attributes object (including destruction) does not affect any previously initialised mutexes.

The `pthread_mutexattr_destroy()` function destroys a mutex attributes object; the object becomes, in effect, uninitialised. 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-initialised using `pthread_mutexattr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

RETURN VALUE

Upon successful completion, `pthread_mutexattr_init()` and `pthread_mutexattr_destroy()` return zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_mutexattr_init()` function may fail if:

`ENOMEM`

Insufficient memory exists to initialise the mutex attributes object.

The `pthread_mutexattr_destroy()` function may fail if:

`EINVAL`

The value specified by *attr* is invalid.

These functions will not return an error code of `EINTR`.

SEE ALSO

`pthread_cond_init(3T)`, `pthread_create(3T)`, `pthread_intro(3T)`,
`pthread_mutexattr_init(3T)`, `pthread_mutex_init(3T)`, `pthread(5)`.

pthread_mutexattr_setprioceiling(3T)**NAME**

pthread_mutexattr_setprioceiling, pthread_mutexattr_getprioceiling - set and get prioceiling attribute of mutex attribute object (**REALTIME THREADS**)

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
    int prioceiling);

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

DESCRIPTION

ENOSYS is returned every time this interface is called, since **REALTIME THREADS** are not supported.

SEE ALSO

pthread_cond_init(3T), pthread_create(3T), [pthread_intro\(3T\)](#), pthread_mutex_init(3T), [pthread\(5\)](#).

pthread_mutexattr_setprotocol(3T)**NAME**

pthread_mutexattr_setprotocol, pthread_mutexattr_getprotocol - set and get protocol attribute of mutex attribute object (**REALTIME THREADS**)

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);

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

DESCRIPTION

ENOSYS is returned every time this interface is called, since **REALTIME THREADS** are not supported.

SEE ALSO

pthread_cond_init(3T), pthread_create(3T), [pthread_intro\(3T\)](#), pthread_mutex_init(3T), [pthread\(5\)](#).

pthread_mutexattr_settype(3T)**NAME**

pthread_mutexattr_gettype, pthread_mutexattr_settype - get or set a mutex type

SYNOPSIS

```
#include <pthread.h>
int pthread_mutexattr_gettype(pthread_mutexattr_t *attr, int
*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 behaviour. Attempting to unlock an unlocked mutex results in undefined behaviour.

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 which 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 which 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 which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.

PTHREAD_MUTEX_DEFAULT

Attempting to recursively lock a mutex of this type results in undefined behaviour. Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behaviour. Attempting to unlock a mutex of this type which is not locked results in undefined behaviour. An implementation is allowed to map this mutex to one of the other mutex types.

RETURN VALUE

If successful, the pthread_mutexattr_settype() function returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the pthread_mutexattr_gettype() function returns zero and stores the value of the *type* attribute of *attr* into the object referenced by the *type* parameter. Otherwise an error 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.

APPLICATION USAGE

It is advised that an application should not use a `PTHREAD_MUTEX_RECURSIVE` mutex with condition variables because the implicit unlock performed for a `pthread_cond_wait()` or `pthread_cond_timedwait()` may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

SEE ALSO

`pthread_cond_timedwait(3T)`, `pthread_cond_wait(3T)`, [pthread_intro\(3T\)](#), [pthread\(5\)](#).

pthread_mutex_init(3-thr)**NAME**

pthread_mutex_init - creates a mutex

SYNOPSIS

```
#include <pthread.h>
int pthread_mutex_init(
pthread_mutex_t *mutex,
pthread_mutexattr_t attr);
```

PARAMETERS

mutex

Mutex that is created.

attr

Attributes object that defines the characteristics of the created mutex. If you specify pthread_mutexattr_default, default attributes are used.

DESCRIPTION

The pthread_mutex_init() routine creates a mutex and initializes it to the unlocked state. If the thread that called this routine terminates, the created mutex is not automatically deallocated, because it is considered shared among multiple threads.

RETURN VALUES

If an error condition occurs, this routine returns -1, the mutex is not initialized, the contents of *mutex* are undefined, and *errno* may be set to one of the following values:

EAGAIN

The system lacks the necessary resources to initialize another mutex.

EINVAL

The value specified by *attr* is invalid.

ENOMEM

Insufficient memory exists to initialize the mutex.

SEE ALSO

pthread_mutex_lock(3-thr), pthread_mutex_trylock(3-thr), pthread_mutex_unlock(3-thr), pthread_mutexattr_create(3-thr), pthread_mutexattr_getkind_np(3-thr), pthread_mutexattr_setkind_np(3-thr).

pthread_mutex_lock(3-thr)**NAME**

pthread_mutex_lock - locks an unlocked mutex

SYNOPSIS

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

PARAMETERS

mutex

Mutex that is locked.

DESCRIPTION

The `pthread_mutex_lock()` routine locks a mutex. If the mutex is locked when a thread calls this routine, the thread waits for the mutex to become available.

The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the mutex in the locked state and with the current thread as the mutex's current owner.

If you specified a fast mutex in a call to `pthread_mutexattr_setkind_np()`, a deadlock can result if the current owner of a mutex calls this routine in an attempt to lock the mutex a second time. If you specified a recursive mutex in a call to `pthread_mutexattr_setkind_np()`, the current owner of a mutex can relock the same mutex without blocking. If you specify a nonrecursive mutex in a call to `pthread_mutexattr_setkind_np()`, an error is returned and the thread does not block if the current owner of a mutex calls this routine in an attempt to lock the mutex a second time.

The preemption of a lower-priority thread that locks a mutex possibly results in the indefinite blocking of higher-priority threads waiting for the same mutex. The execution of the waiting higher-priority threads is blocked for as long as there is a sufficient number of runnable threads of any priority between the lower-priority and higher-priority values. Priority inversion occurs when any resource is shared between threads with different priorities.

RETURN VALUES

If the function fails, `errno` may be set to one of the following values:

`EINVAL`

The value specified by *mutex* is invalid.

`EDEADLK`

A deadlock condition is detected.

SEE ALSO

`pthread_mutex_destroy(3-thr)`, `pthread_mutex_init(3-thr)`, `pthread_mutex_trylock(3-thr)`, `pthread_mutex_unlock(3-thr)`, `pthread_mutexattr_setkind_np(3-thr)`.

pthread_mutex_setprioceiling(3T)**NAME**

pthread_mutex_setprioceiling, pthread_mutex_getprioceiling - change the priority ceiling of a mutex (**REALTIME THREADS**)

SYNOPSIS

```
#include <pthread.h>

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

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

DESCRIPTION

ENOSYS is returned every time this interface is called, since **REALTIME THREADS** are not supported.

SEE ALSO

pthread_intro(3T), pthread_mutex_init(3T), pthread_mutex_lock(3T), pthread_mutex_trylock(3T), pthread_mutex_unlock(3T), pthread(5).

pthread_once(3-thr)**NAME**

`pthread_once` - calls an initialization routine executed by one thread, a single time

SYNOPSIS

```
#include <pthread.h>
int pthread_once(
    pthread_once_t *once_block,
    pthread_initroutine_t init_routine);
```

PARAMETERS

once_block

Address of a record that defines the one-time initialization code. Each one-time initialization routine must have its own unique `pthread_once_t` data structure.

init_routine

Address of a procedure that performs the initialization. This routine is called only once, regardless of the number of times it and its associated *once_block* are passed to `pthread_once()`.

DESCRIPTION

The `pthread_once()` routine calls an initialization routine executed by one thread, a single time. This routine allows you to create your own initialization code that is guaranteed to be run only once, even if called simultaneously by multiple threads or multiple times in the same thread.

For example, a mutex or a thread-specific data key must be created exactly once. Calling `pthread_once()` prevents the code that creates a mutex or thread-specific data from being called by multiple threads. Without this routine, the execution must be serialized so that only one thread performs the initialization. Other threads that reach the same point in the code are delayed until the first thread is finished.

This routine initializes the control record if it has not been initialized and then determines if the client one-time initialization routine has executed once. If it has not executed, this routine calls the initialization routine specified in *init_routine*. If the client one-time initialization code has executed once, this routine returns.

The `pthread_once_t` data structure is a record that allows client initialization operations to guarantee mutual exclusion of access to the initialization routine, and that each initialization routine is executed exactly once.

The client code must declare a variable of type `pthread_once_t` to use the client initialization operations. This variable must be initialized using the `pthread_once_init` macro, as follows:

```
static pthread_once_t myOnceBlock = pthread_once_init;
```

RETURN VALUES

If the function fails, `errno` may be set to one of the following values:

`EINVAL`

The value specified by a parameter is invalid.

pthread_rwlockattr_getpshared(3T)**NAME**

pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared - get and set process-shared attribute of read-write lock attributes object

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr,
    int *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 VALUE

If successful, the `pthread_rwlockattr_setpshared()` function returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the `pthread_rwlockattr_getpshared()` returns zero 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 may fail if:

`EINVAL`

The value specified by *attr* is invalid.

The `pthread_rwlockattr_setpshared()` function may fail if:

`EINVAL`

The new value specified for the attribute is outside the range of legal values for that attribute.

SEE ALSO

`pthread_intro(3T)`, `pthread_rwlockattr_init(3T)`, `pthread_rwlock_init(3T)`, `pthread_rwlock_rdlock(3T)`, `pthread_rwlock_unlock(3T)`, `pthread_rwlock_wrlock(3T)`, `pthread(5)`.

pthread_rwlockattr_init(3T)**NAME**

`pthread_rwlockattr_init`, `pthread_rwlockattr_destroy` - initialise and destroy read-write lock attributes object

SYNOPSIS

```
#include <pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);□
```

DESCRIPTION

The function `pthread_rwlockattr_init()` initialises 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 initialised read-write lock attributes object.

After a read-write lock attributes object has been used to initialise one or more read-write locks, any function affecting the attributes object (including destruction) does not affect any previously initialised 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-initialised by another call to `pthread_rwlockattr_init()`. An implementation may cause `pthread_rwlockattr_destroy()` to set the object referenced by *attr* to an invalid value.

RETURN VALUE

If successful, the `pthread_rwlockattr_init()` and `pthread_rwlockattr_destroy()` functions return zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_rwlockattr_init()` function will fail if:

`ENOMEM`

Insufficient memory exists to initialise the read-write lock attributes object.

The `pthread_rwlockattr_destroy()` function may fail if:

`EINVAL`

The value specified by *attr* is invalid.

SEE ALSO

[pthread_intro\(3T\)](#), [pthread_rwlockattr_getpshared\(3T\)](#), [pthread_rwlock_init\(3T\)](#), [pthread_rwlock_rdlock\(3T\)](#), [pthread_rwlock_unlock\(3T\)](#), [pthread_rwlock_wrlock\(3T\)](#), [pthread\(5\)](#).

pthread_rwlock_init(3T)**NAME**

`pthread_rwlock_init`, `pthread_rwlock_destroy` - initialise or destroy a read-write lock object

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwlock,
    const pthread_rwlockattr_t *attr);

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;
```

DESCRIPTION

The `pthread_rwlock_init()` function initialises 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 initialised, the lock can be used any number of times without being re-initialised. Upon successful initialisation, the state of the read-write lock becomes initialised and unlocked. Results are undefined if `pthread_rwlock_init()` is called specifying an already initialised read-write lock. Results are undefined if a read-write lock is used without first being initialised.

If the `pthread_rwlock_init()` function fails, `rwlock` is not initialised 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-initialised 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 uninitialised read-write lock results in undefined behaviour. A destroyed read-write lock object can be re-initialised 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 initialise read-write locks that are statically allocated. The effect is equivalent to dynamic initialisation by a call to `pthread_rwlock_init()` with the parameter `attr` specified as `NULL`, except that no error checks are performed.

RETURN VALUE

If successful, the `pthread_rwlock_init()` and `pthread_rwlock_destroy()` functions return zero. Otherwise, an error number is returned to indicate the error. The `EBUSY` and `EINVAL` error checks, if implemented, will 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 read-write lock specified by `rwlock`.

ERRORS

The `pthread_rwlock_init()` function will fail if:

`EAGAIN`

The system lacked the necessary resources (other than memory) to initialise another read-write lock.

`ENOMEM`

Insufficient memory exists to initialise the read-write lock.

`EPERM`

The caller does not have the privilege to perform the operation.

The `pthread_rwlock_init()` function may fail if:

EBUSY

The implementation has detected an attempt to re-initialise the object referenced by *rwlock*, a previously initialised but not yet destroyed read-write lock.

EINVAL

The value specified by *attr* is invalid.

The `pthread_rwlock_destroy()` function may fail if:

EBUSY

The implementation has detected an attempt to destroy the object referenced by *rwlock* while it is locked.

EINVAL

The value specified by *attr* is invalid.

SEE ALSO

`pthread_intro(3T)`, `pthread_rwlockattr_init(3T)`, `pthread_rwlock_rdlock(3T)`,
`pthread_rwlock_unlock(3T)`, `pthread_rwlock_wrlock(3T)`, `pthread(5)`.

pthread_rwlock_rdlock(3T)**NAME**

`pthread_rwlock_rdlock`, `pthread_rwlock_tryrdlock` - lock a read-write lock object for reading

SYNOPSIS

```
#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. It is unspecified whether the calling thread acquires the lock when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the `pthread_rwlock_rdlock()` call) until it can acquire the lock. Results are undefined if the calling thread holds a write lock on *rwlock* at the time the call is made.

Implementations are allowed to favour writers over readers to avoid writer starvation.

A thread may 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 function `pthread_rwlock_tryrdlock()` applies a read lock as in the `pthread_rwlock_rdlock()` function with the exception that the function fails if any thread holds a write lock on *rwlock* or there are writers blocked on *rwlock*.

Results are undefined if any of these functions are called with an uninitialised 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 VALUE

If successful, the `pthread_rwlock_rdlock()` function returns zero. Otherwise, an error number is returned to indicate the error.

The function `pthread_rwlock_tryrdlock()` returns zero 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_tryrdlock()` function will fail if:

EBUSY

The read-write lock could not be acquired for reading because a writer holds the lock or was blocked on it.

The `pthread_rwlock_rdlock()` and `pthread_rwlock_tryrdlock()` functions may fail if:

EAGAIN

The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded.

EDEADLK

The current thread already owns the read-write lock for writing.

EINVAL

The value specified by *rwlock* does not refer to an initialised read-write lock object.

APPLICATION USAGE

Realtime applications may encounter priority inversion when using read-write locks. The problem occurs when a high priority thread "locks" a read-write lock that is about to be "unlocked" 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 read-write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

SEE ALSO

`pthread_intro(3T)`, `pthread_rwlockattr_init(3T)`, `pthread_rwlock_init(3T)`,
`pthread_rwlock_unlock(3T)`, `pthread_rwlock_wrlock(3T)`, `pthread(5)`.

pthread_rwlock_unlock(3T)**NAME**

pthread_rwlock_unlock - unlock a read-write lock object

SYNOPSIS

```
#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 uninitialised read-write lock.

RETURN VALUE

If successful, the `pthread_rwlock_unlock()` function returns zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_rwlock_unlock()` function may fail if:

EINVAL

The value specified by *rwlock* does not refer to an initialised read-write lock object.

EPERM

The current thread does not own the read-write lock.

SEE ALSO

[pthread_intro\(3T\)](#), [pthread_rwlockattr_init\(3T\)](#), [pthread_rwlock_init\(3T\)](#), [pthread_rwlock_rdlock\(3T\)](#), [pthread_rwlock_wrlock\(3T\)](#), [pthread\(5\)](#).

pthread_rwlock_wrlock(3T)**NAME**

pthread_rwlock_wrlock, pthread_rwlock_trywrlock - lock a read-write lock object for writing

SYNOPSIS

```
#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 (that is, does not return from the `pthread_rwlock_wrlock()` call) until it can acquire the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.

Implementations are allowed to favour writers over readers to avoid writer starvation.

The function `pthread_rwlock_trywrlock()` 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). Results are undefined if any of these functions are called with an uninitialised 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 VALUE

If successful, the `pthread_rwlock_wrlock()` function returns zero. Otherwise, an error number is returned to indicate the error.

The function `pthread_rwlock_trywrlock()` returns zero 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_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.

The `pthread_rwlock_wrlock()` and `pthread_rwlock_trywrlock()` functions may fail if:

EDEADLK

The current thread already owns the read-write lock for writing or reading.

EINVAL

The value specified by `rwlock` does not refer to an initialised read-write lock object.

APPLICATION USAGE

Realtime applications may encounter priority inversion when using read-write locks. The problem occurs when a high priority thread "locks" a read-write lock that is about to be "unlocked" 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 read-write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

SEE ALSO

```
pthread_intro(3T), pthread_rwlockattr_init(3T), pthread_rwlock_init(3T),  
pthread_rwlock_rdlock(3T), pthread_rwlock_unlock(3T), pthread(5).
```

pthread_self(3-thr)**NAME**

`pthread_self` - obtains the identifier of the current thread

SYNOPSIS

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

DESCRIPTION

The `pthread_self()` routine allows a thread to obtain its own identifier. For example, this identifier allows a thread to set its own priority.

This value becomes meaningless when the thread object is deleted; that is, when the thread terminates its execution and `pthread_detach()` is called.

RETURN VALUES

Returns the identifier of the calling thread to `pthread_t`.

SEE ALSO

`pthread_create(3-thr)`, `pthread_setprio(3-thr)`, `pthread_setscheduler(3-thr)`.

pthread_setcancelstate(3T)**NAME**

pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel - set cancelability state

SYNOPSIS

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

DESCRIPTION

The `pthread_setcancelstate()` function atomically both sets the calling thread's cancelability state to the indicated *state* and returns the previous cancelability state at the location referenced by *oldstate*. Legal values for *state* are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`.

The `pthread_setcanceltype()` function atomically both sets the calling thread's cancelability type to the indicated *type* and returns the previous cancelability type at the location referenced by *oldtype*. Legal values for *type* are `PTHREAD_CANCEL_DEFERRED` and `PTHREAD_CANCEL_ASYNCHRONOUS`.

The cancelability state and type of any newly created threads, including the thread in which `main()` was first invoked, are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED` respectively.

The `pthread_testcancel()` function creates a cancellation point in the calling thread. The `pthread_testcancel()` function has no effect if cancelability is disabled.

RETURN VALUE

If successful, the `pthread_setcancelstate()` and `pthread_setcanceltype()` functions return zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_setcancelstate()` function may fail if:

`EINVAL`

The specified state is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.

The `pthread_setcanceltype()` function may fail if:

`EINVAL`

The specified type is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.

These functions will not return an error code of `EINTR`.

SEE ALSO

`pthread_cancel(3T)`, `pthread_intro(3T)`, `pthread(5)`.

pthread_setspecific(3-thr)**NAME**

`pthread_setspecific` - sets the thread-specific data associated with the specified key for the current thread

SYNOPSIS

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key,
pthread_addr_t value);
```

PARAMETERS

key Context key value that uniquely identifies the context value specified in *value*. This key value must have been obtained from `pthread_keycreate()`.

value

Address containing data to be associated with the specified key for the current thread; this is the thread-specific data.

DESCRIPTION

The `pthread_setspecific()` routine sets the thread-specific data associated with the specified key for the current thread. If a value has already been defined for the key in this thread, the new value is substituted for it.

Different threads can bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that are reserved for use by the calling thread.

RETURN VALUES

If the function fails, `-1` is returned, and `errno` may be set to the following value:

`EINVAL`

The key value is invalid.

SEE ALSO

`pthread_getspecific(3-thr)`, `pthread_keycreate(3-thr)`.

sched_yield(3T)**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 it again becomes the head of its thread list. It takes no arguments.

RETURN VALUE

The `sched_yield()` function returns 0 if it completes successfully, or it returns a value of `-1` and sets `errno` to indicate the error.

ERRORS

No errors are defined.

SEE ALSO

[pthread_intro\(3T\)](#), [sched\(5\)](#).

Functions from special libraries(3X)**sigwait(3-thr)****NAME**

`sigwait` - causes a thread to wait for an asynchronous signal

SYNOPSIS

```
#include <pthread.h>
int sigwait(sigset_t *set);
```

PARAMETERS

`set` Set of pending signals upon which the calling thread will wait.

DESCRIPTION

This routine causes a thread to wait for an asynchronous signal. It atomically chooses a pending signal from `set`, atomically clears it from the system's set of pending signals and returns that signal number. If no signal in `set` is pending at the time of the call, the thread is blocked until one or more signals becomes pending. The signals defined by `set` may be unblocked during the call to this routine and will be blocked when the thread returns from the call unless some other thread is currently waiting for one of those signals.

A thread must block the signals it waits for using `sigprocmask()` prior to calling this function.

If more than one thread is using this routine to wait for the same signal, only one of these threads will return from this routine with the signal number.

A call to `sigwait()` is a cancellation point.

RETURN VALUES

Possible return values are as follows:

EINVAL

One or more of the values specified by *set* is invalid.

EINVAL

One or more of the values specified by *set* is not blocked.

EINVAL

There are no values specified in *set*.

SEE ALSO

`pause(2)`, `sigpending(2)`, `sigprocmask(2)`, `pthread_cancel(3-thr)`,
`pthread_setasynccancel(3-thr)`, `sigsetops(3C)`, `signal(5)`.

NAME

`aioinit` - asynchronous I/O initialization (Industry Standard)

SYNOPSIS

```
#include <sys/kaioapi_pyr.h>
int aioinit(void);
```

DESCRIPTION

This function is contained in the `libspecaio.a` library.

`aioinit()` must have been called once before the other industry standard I/O functions can be called.

SEE ALSO

`aiopoll(3X)`, `aioread(3X)`, `aiostat(3X)`, `aiowrite(3X)`, `aio(7)`, `aio_ISTD(7)`.

aiopoll(3X)**NAME**

`aiopoll` - asynchronous I/O poll (Industry Standard)

SYNOPSIS

```
#include <sys/kaioapi_pyr.h>
aioch_t *
aiopoll(const void *key, int timeout);
```

DESCRIPTION

This function is contained in the `libspecaio.a` library.

`aiopoll()` is a new function which allows a waiting time until one or more asynchronous I/O jobs are completed. Status information on the completed jobs is then returned.

If `NULL` is supplied as the parameter value for `key`, a waiting time is allowed for the execution of any of the jobs pending for the calling process, regardless of which `aio_key` was set [see [aio_ISTD\(7\)](#)].

If a special keyword is specified as a parameter for the `key`, a waiting time is allowed for the execution of one of the jobs pending for this process, which has specified this value in its `aio_key` parameter.

It is therefore irrelevant whether the jobs returned were error-free or whether they were terminated due to an error.

Using the `timeout` parameter, it is possible to specify the maximum number of milliseconds to be waited for the completion of the selected job. If ready messages for several suitable jobs for the process are to be output, the address of a job field `aioch` [see [aio_ISTD\(7\)](#)] is output as the return value for the function. Any other completed jobs are linked by means of the `aio_next` parameter in the `aioch`. If `aio_next` is the `NULL` pointer, this signals the end of the chain.

If `timeout` occurs before any of the jobs have been completed, the `NULL` pointer is returned.

If `timeout` is equal to 0, `aiopoll()` returns immediately to the program from which it was called. Existing completed and selected jobs at the time of the call are returned. If there are no suitable jobs to be returned, the `NULL` pointer is returned instead.

If `timeout` is equal to -1, `aiopoll()` waits until any of the jobs for this process selected using `key` has been terminated. It should be noted that if `timeout` is equal to -1, it is important to ensure that a job can be completed, because if it cannot, the function must be explicitly terminated by a signal.

The I/O job-specific acknowledgements are set as relevant in the returned `aioch` job fields. In this way, the values `aio_nbytes` and `aio_errno` are defined and set. `aio_nbytes` contains the actual number of transferred bytes. `aio_errno` contains the error/status message as it would have been returned in analog form during normal I/O system calls such as [read\(2\)](#) or [write\(2\)](#).

DIAGNOSTICS

Error and status messages are output using the global variable `errno`. The `NULL` pointer is also returned as the return value.

`EINVAL`

Unpermitted negative value for `timeout`.

`ETIME`

There is no outstanding asynchronous I/O job for the process, or the `timeout` value has elapsed.

`EINTR`

If `timeout` = -1 and the function was terminated by means of a signal.

SEE ALSO

[aioinit\(3X\)](#), [aioread\(3X\)](#), [aiostat\(3X\)](#), [aiowrite\(3X\)](#), [aio\(7\)](#), [aio_ISTD\(7\)](#).

aioread(3X)**NAME**

aioread - asynchronous I/O read (Industry Standard)

SYNOPSIS

```
#include <sys/kaioapi_pyr.h>
int aioread(aiocb_t *aiocbp);
```

DESCRIPTION

This function is contained in the `libspecaio.a` library.

`aioread()` allows the caller to start an asynchronous read job.

The job parameters must be set in a control block provided by the calling process [see [aio_ISTD\(7\)](#)].

Additional information on the control block parameters to be set [see also [aio\(7\)](#)].

For the address of the user buffer to be set in `aio_buf`, the same alignment requirements apply as for SPECAIO read jobs.

`aio_nbytes` must contain the number of bytes to be read. The size of a transfer must be an integer multiple of the number of disk sectors (sector size 512 bytes), and small enough to be handled in a single I/O job on the device.

`aio_offset` must contain the start address of the area to be read in relation to the start of the file. The value must be an integer multiple of the number of disk sectors (sector size 512 bytes).

If a special keyword is specified as a parameter for the `aio_key`, it is possible to wait specifically for the ready message for this job in a later `aiopoll()` call. A number of asynchronous jobs can contain the same key value. If `aio_key` is specified equal to 0, it is not possible to specifically select this job. The use of the `aio_key` parameter is application-dependent.

Calling `aioread()` returns the value 0 if the asynchronous job could be initiated. Furthermore, the value `EINPROGRESS` is set in `aio_errno`.

If an error situation has been identified, the value `-1` is returned. In this case, the job was not started. The global variable `errno` contains a corresponding error message.

If the job was started without identifying any errors, the control block `aiocb` is logically assigned to the system. This means that no changes must be made to the control block, and the control block must not be released until the completion of the asynchronous job connected to it is signaled by means of an `aiopoll()` job. It should be noted that the buffer area set using `aio_buf` must be available for write access until completion of the job is signaled.

DIAGNOSTICS

Error and status messages are signaled by means of the global variable `errno`. A value of `-1` is also returned.

EFAULT

Error accessing the control block or buffer area.

EPERM

No read access possible to the file.

EAGAIN

Resource bottleneck in the system; there are no further internal control blocks available.

SEE ALSO

[aioinit\(3X\)](#), [aiopoll\(3X\)](#), [aiostat\(3X\)](#), [aiowrite\(3X\)](#), [aio\(7\)](#), [aio_ISTD\(7\)](#).

NAME

aiostat - asynchronous I/O status check (Industry Standard)

SYNOPSIS

```
#include <sys/kaioapi_pyr.h>
int aiostat(char *path, aiostat_t *buf);
```

DESCRIPTION

This function is contained in the `libspecaio.a` library.

By calling `aiostat()` it is possible to check whether asynchronous I/O access, and in particular the industry standard functions `aioread()`, `aiowrite()`, and `aiopoll()` are supported by the current operating system for the file specified in `path`.

The `path` parameter must contain the name of the file to be checked (analogous to the `open` system call).

`buf` must contain a pointer to an acknowledgement structure `aiostat_t` [see [aio_ISTD\(7\)](#)].

If the function returns the value 0 and if the value `AIO_KAIO` is set in `st_flags`, the industry standard interface is supported.

If the function returns the value -1 and the value 0 is set in `st_flags`, an error has occurred when opening the file (`errno` set accordingly), or asynchronous I/O is not supported for this file.

DIAGNOSTICS

EFAULT

Error when accessing the return field.

SEE ALSO

[aioinit\(3X\)](#), [aiopoll\(3X\)](#), [aioread\(3X\)](#), [aiowrite\(3X\)](#), [aio\(7\)](#), [aio_ISTD\(7\)](#).

NAME

aiowrite - asynchronous I/O write (Industry Standard)

SYNOPSIS

```
#include <sys/kaioapi_pyr.h>
int aiowrite(aiocb_t *aiocbp);
```

DESCRIPTION

This function is contained in the `libspecaio.a` library.

`aiowrite()` allows the caller to start an asynchronous write job.

The job parameters must be set in the control block provided by the calling process [see [aio_ISTD\(7\)](#)].

Additional information on the job parameters to be set [see also [aio\(7\)](#)].

For the address of the user buffer to be set in `aio_buf`, the same alignment requirements apply as for SPECAIO write jobs.

`aio_nbytes` must contain the number of bytes to be read. The size of a transfer must be an integer multiple of the number of disk sectors (sector size 512 bytes), and small enough to be handled in a single I/O job on the device.

`aio_offset` must contain the start address of the area to be read in relation to the start of the file. The value must be an integer multiple of the number of disk sectors (sector size 512 bytes).

If a special key value is specified as a parameter for the `aio_key`, it is possible to wait specifically for the ready message for this job in a later `aiopoll()` call. A number of asynchronous jobs can contain the same key value. If `aio_key` is specified as equal to 0, it is not possible to select this job specifically using `aiopoll()`. The use of the `aio_key` parameter is application-dependent.

Calling `aiowrite()` returns the value 0, if the asynchronous job could be initiated. The value `EINPROGRESS` is also set in `aio_errno`.

If an error situation has been identified, the value `-1` is returned. In this case, the job was not started. The global variable `errno` contains a corresponding error message.

If the job was started without identifying any errors, the control block `aiocb` is logically assigned to the system. This means that no changes must be made to the control block, and the control block must not be released until the completion of the asynchronous job connected to it has been reported by means of an `aiopoll()` job. It should be noted that the buffer area set using `aio_buf` must be available for read access until completion of the job is signaled.

DIAGNOSTICS

Error and status messages are signaled by means of the global variable `errno`. The value `-1` is also returned.

EFAULT

Error accessing the control block or buffer area.

EPERM

No read access to the file.

EAGAIN

Resource bottleneck in the operating system; there are no other internal control blocks available.

SEE ALSO

[aioinit\(3X\)](#), [aiopoll\(3X\)](#), [aioread\(3X\)](#), [aiostat\(3X\)](#), [aio\(7\)](#), [aio_ISTD\(7\)](#).

NAME

`assert` - verify program assertion

SYNOPSIS

```
#include <assert.h>
void assert(int expression);
```

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* returns the value zero, `assert()` prints the 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. The latter are respectively the values of the preprocessor macros `__FILE__` and `__LINE__`.

Compiling with the preprocessor control instruction `#define NDEBUG` ahead of the `#include assert.h` statement will stop assertions from being compiled into the program.

NOTES

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

SEE ALSO

[abort\(3C\)](#).

crypt(3X)**NAME**

`crypt`, `setkey`, `encrypt`, `des_crypt`, `des_setkey`, `des_encrypt`, `run_setkey`,
`run_crypt`, `crypt_close` - password and file encryption functions

SYNOPSIS

```
cc [ flag ... ] file ... -lcrypt
#include <crypt.h>

□
□
char *crypt(const char *key, const char *salt);
void setkey(const char *key);
void encrypt(char *block, int flag);
char *des_crypt(const char *key, const char *salt);
void des_setkey(const char *key);
void des_encrypt(char *block, int flag);
int run_setkey(int *p, const char *key);
int run_crypt(long offset, char *buffer, unsigned int count, □
□□□□□□□□□□□□ int *p);
int crypt_close(int *p);
```

DESCRIPTION

`des_crypt()` is the password encryption function. It is based on a one-way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

key is a user's typed password. *salt* is a two-character string chosen from the set [a-zA-Z0-9./]; this string is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the *salt* itself.

The `des_setkey()` and `des_encrypt()` functions provide access to the actual hashing algorithm. The argument of `des_setkey()` is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, thereby creating a 56-bit key that is set into the machine. This key is the key that will be used with the hashing algorithm to encrypt the string *block* with the function `des_encrypt()`.

The argument to the `des_encrypt()` entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by `des_setkey()`. If *flag* is zero, the argument is encrypted; if non-zero, it is decrypted.

Note that decryption is not provided in the international version of `crypt(3X)`. The international version is part of the C Development Set, and the domestic version is part of the Security Administration Utilities. If decryption is attempted with the international version of `des_encrypt()`, an error message is printed.

`crypt()`, `setkey()`, and `encrypt()` are front-end routines that invoke `des_crypt()`, `des_setkey()`, and `des_encrypt()` respectively.

The routines `run_setkey()` and `run_crypt()` are designed for use by applications that need cryptographic capabilities [such as `ed(1)` and `vi(1)`] that must be compatible with the `crypt(1)` user-level utility. `run_setkey()` establishes a two-way pipe connection with the `crypt()` utility, using *key* as the password argument. `run_crypt()` takes a block of characters and transforms the cleartext or ciphertext into their ciphertext or cleartext using the `crypt()` utility. *offset* is the relative byte position from the beginning of the file that the block of text provided in *buffer* is coming from. *count* is the number of characters in *buffer*,

and *p* is an array containing indices to a table of input and output file streams. When encryption is finished, `crypt_close()` is used to terminate the connection with the `crypt(1)` utility.

`run_setkey()` returns `-1` if a connection with the `crypt(1)` utility cannot be established. This result will occur in international versions of the Reliant UNIX system in which the `crypt(1)` utility is not available. If a null key is passed to `run_setkey()`, `0` is returned; otherwise, `1` is returned. `run_crypt()` returns `-1` if it cannot write output or read input from the pipe attached to `crypt()`. Otherwise it returns `0`.

The program must be linked with the object file access routine library `libcrypt.a`.

RESULT

In the international version of `crypt(3X)`, a flag argument of `1` to `encrypt()` or `des_encrypt()` is not accepted, and `errno` is set to `ENOSYS` to indicate that the functionality is not available.

NOTES

The return value in `crypt()` points to static data that is overwritten by each call.

SEE ALSO

`crypt(1)`, `login(1)`, `passwd(1)`, `getpass(3C)`, `passwd(4)`.

dladdr(3X)**NAME**

dladdr - translate address to symbolic information

SYNOPSIS

```
cc [ flag ... ] file ... -ldl [ library ... ]
#include <dlfcn.h>

int dladdr(void *address, Dl_info *dli);
```

DESCRIPTION

dladdr() is one of a family of routines that give the user direct access to the dynamic linking facilities. These routines are made available via the library loaded when the option **-ldl** is passed to the link-editor.

Note: These routines are available to dynamically-linked processes *only*.

dladdr() determines if the specified address is located within one of the mapped objects that make up the current applications address space. An address is considered to fall within a mapped object when it is between the base address, and the **_end** address of that object. If a mapped object fits this criteria, the symbol table made available to the run-time 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** structure must be preallocated by the user. The structure members are filled in by **dladdr()** based on the specified address. The **Dl_info** structure includes the following members:

```
const char    *dli_fname;
void          *dli_fbase;
const char    *dli_sname;
void          *dli_saddr;
size_t        dli_size;
int           dli_bind;
int           dli_type;
```

Descriptions of these members appear below.

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 nearest to the specified address. This symbol either has the same address, or is the nearest symbol with a lower address.

dli_saddr

Contains the actual address of the above symbol.

dli_size

Contains the size in bytes of the above symbol.

dli_bind

Contains the binding attributes of the above symbol.

dli_type

Contains the type of the above symbol.

RESULT

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** elements are filled.

NOTES

The `Dl_info` pointer elements point to addresses within the mapped objects. These may become invalid if objects are removed prior to these elements being used [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.

SEE ALSO

[ld\(1\)](#), [dlclose\(3C\)](#), [dlerror\(3C\)](#), [dlopen\(3C\)](#), [dlsym\(3C\)](#).

Chapter on "The C compilation system" in *Guide to Tools for Programming in C*.

getcounter(3X)**NAME**

`getcounter` - get microsecond timer value

SYNOPSIS

```
cc [ flag ... ] file ... -lusc [ library ... ]  
unsigned int getcounter(void);
```

DESCRIPTION

`getcounter` returns the actual value of the timer variable. This is the elapsed time in microseconds since the last `initcounter` function call. The range of values depends on the clock frequency of the CPUs in the system.

You can use the following formula to roughly calculate the time in minutes until an overflow will occur:

$$((2^{38}) - 1) / (\text{clock\_frequency\_in\_MHz} * 60 * 10^6)$$

The value used for the clock frequency is that output as the frequency with `/usr/ucb/mpcntl -I`.

A clock frequency of 200 MHz means, for example, that an overflow can be expected in 22 minutes.

DIAGNOSTICS

The value of 0xffffffff is returned, if there was no previous `initcounter` call or if an error occurs while accessing the timer (no `usc` driver configured, or `/dev/usc` not readable).

NOTES

This function returns valid values only if at least 6 minutes have elapsed since system startup and if all the CPUs in the system have the same clock frequency. Application programs have to take care of overflow.

SEE ALSO

`mpcntl(1M)`, `initcounter(3X)`.

initcounter(3X)**NAME**

`initcounter` - initialize microsecond timer

SYNOPSIS

```
cc [ flag ... ] file ... -lusc [ library ... ]  
void initcounter(void);
```

DESCRIPTION

`initcounter` initializes the timer variable with zero. The timer will be incremented with a resolution of 1 microsecond. The range of values depends on the clock frequency of the CPUs in the system.

You can use the following formula to roughly calculate the time in minutes until an overflow will occur:

$$((2^{38}) - 1) / (\text{clock\_frequency\_in\_MHz} * 60 * 10^6)$$

The value used for the clock frequency is that output as the frequency with `/usr/ucb/mpcctl -I`.

A clock frequency of 200 MHz means, for example, that an overflow can be expected in 22 minutes.

NOTES

This function returns valid values only if at least 6 minutes have elapsed since system startup and if all the CPUs in the system have the same clock frequency. Application programs have to take care of overflow.

SEE ALSO

`mpcctl(1M)`, `getcounter(3X)`.

maillock(3X)**NAME**

maillock, mailunlock - manage lockfile for user's mailbox

SYNOPSIS

```
cc [ flag ... ] file ... -lmail
#include <maillock.h>
□
□
int maillock(const char *user, int retrycnt);
int mailunlock(void);□
```

DESCRIPTION

The `maillock()` function attempts to create a lockfile for the user's mailfile. If a lockfile already exists, `maillock()` assumes the contents of the file is the process ID (as a null-terminated ASCII string) of the process that created the lockfile (presumably with a call to `maillock()`). If the process that created the lockfile is still alive, `maillock()` will sleep and try again *retrycnt* times before returning with an error indication. The sleep algorithm is to sleep for 5 seconds times the attempt number. That is, the first sleep will be for 5 seconds, the next sleep will be for 10 seconds, etc. until the number of attempts reaches *retrycnt*. When the lockfile is no longer needed, it should be removed by calling `mailunlock()`.

user is the login name of the user for whose mailbox the lockfile will be created. `maillock()` assumes that users' mailfiles are in the memory location provided, as defined in `maillock.h`.

The following return code definitions are contained in `maillock.h`.

```
#define L_SUCCESS      0    /* Lockfile created or removed */
#define L_NAMELEN      1    /* Recipient name > 13 chars */
#define L_TMPLOCK      2    /* Can't create tmp file */
#define L_TMPWRITE     3    /* Can't write pid into lockfile */
#define L_MAXTRY      4    /* Failed after retrycnt attempts */
#define L_ERROR        5    /* Check errno for reason */
#define L_MANLOCK      6    /* Unable to set mandatory lock */
```

NOTES

`mailunlock()` will only remove the lockfile created from the most previous call to `maillock()`. Calling `maillock()` for different users without intervening calls to `mailunlock()` will cause the initially created lockfile(s) to remain, potentially blocking subsequent message delivery until the current process finally terminates.

FILES

```
LIBDIR/llib-mail.ln
LIBDIR/libmail.a
/var/mail/*
/var/mail/*.lock
```


NAME

malloc, free, realloc, calloc, malloc, mallinfo - allocate memory

SYNOPSIS

```
cc [flag ...] file ... -lmalloc
#include <stdlib.h>
#include <malloc.h>
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
void *calloc(size_t nelem, size_t elsize);
#include <malloc.h>
int malloc(int cmd, int value);
struct mallinfo mallinfo(void);□
```

DESCRIPTION

malloc() and free() provide a simple general-purpose memory allocation package.

malloc() returns a pointer to a block of at least *size* bytes. The space is not initialized.

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 (but see malloc() 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().

realloc() changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes.

calloc() allocates memory space for an array of *nelem* elements of size *elsize*. The memory space is initialized to zeros.

malloc() provides for control over the allocation algorithm. The available values for *cmd* are defined in the malloc.h header file, and have the following meanings:

M_MXFAST

Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups and then does 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 is not recommended.

malloc() may be called repeatedly, but may not be called after the first small block is allocated.

mallinfo() provides instrumentation describing space usage. It returns the structure:

```

struct mallinfo {
    int arena;      /* total space in arena */
    int ordblks;    /* number of ordinary blocks */
    int smlblks;    /* number of small blocks */
    int hblkhd;     /* space in holding block headers */
    int hblks;      /* number of holding blocks */
    int usmlblks;   /* space in small blocks in use */
    int fsmblks;    /* space in free small blocks */
    int uordblks;   /* space in ordinary blocks in use */
    int fordblks;   /* space in free ordinary blocks */
    int keepcost;   /* space penalty if keep option is used */
}

```

This structure is defined in the `malloc.h` header file.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

RESULT

`malloc()`, `realloc()`, and `calloc()` 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 `mallopt()` is called after any allocation or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

NOTES

Note that 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 file for compatibility with old applications. New applications should include `stdlib.h` to access the prototypes for these functions.

The algorithm used by `malloc(3X)` is a "first-fit" algorithm. If you want to use the "best-fit" algorithm, you should use `malloc(3C)`.

SEE ALSO

`brk(2)`, `malloc(3C)`.

NAME

`mpcntl` - standard interface for multiprocessing

SYNOPSIS

```
cc [ flag ... ] file ... -lmproc
#include <sys/types.h>
#include <sys/mpcntl.h>
mpcntl(int req, void *arg);
```

DESCRIPTION

`mpcntl()` provides a number of multiprocessor services to application programmers wishing to exploit this environment.

Binding: allows a process to explicitly bind itself to a particular CPU. The binding may be exclusive, where multiple processes are allowed to bind to a single CPU, and all other processes not exclusively bound are excluded. Or the alternative is the default situation, where multiple processes share one CPU (without excluding other processes).

Affinity Control: defeats or re-enables the default affinity policies used by the kernel for scheduling a process in such a way as to try to preserve its cache state. A process for which affinity has been defeated will be available to any processor for scheduling.

Limited Scheduler Control: allows a process to temporarily suspend preemption by other processes until the process blocks or re-enables preemption. This also makes the process immune to all signals (except `SIGKILL`). The process can also elect to give up, or yield, the CPU to allow the system to schedule other processes. If the yielding process has high enough priority, it will not be preempted.

CPU Status: returns the number of CPUs in the machine and information on specific CPUs as requested. This also allows one to change the state information of a CPU.

req specifies the service requested.

arg, a pointer which may be a null pointer for certain requests. In most cases, *arg* is interpreted as a pointer to an `mpcntl()` request structure. It is defined in `mpcntl.h` as follows:

```
typedef uint_t cpumask_t;
typedef struct mpcreq {
    cpumask_t mpc_cpu;        /* CPU identifier */
    pid_t mpc_pid;           /* valid process ID */
} mpcreq_t;
```

The field *mpc_cpu* is a *cpumask_t* (an `int`) in which each bit is used to represent a logical CPU (bit 0 represents CPU 0, bit 1, CPU 1 etc.) This allows more than 1 CPU id to be passed on to the system at a time. The macro `CPUNO_TO_LCPUID` can be used to convert between cpu numbers and a bit in the *cpumask_t* mask. No requests require more than 1 CPU to be specified. The *mcp_pid* is a *pid_t* set to a valid process ID, as returned by a `getpid()` system call.

Inappropriate use of some of the `mpcntl()` operations could result in degraded system performance or, in some cases, deadlocks. To prevent naive uses, the user of these particular `mpcntl()` options must be either the superuser (user ID 0) or have been granted specific `mpcntl()` privileges by the superuser.

The following is a list of valid requests to `mpcntl()` and a description of *arg* for each:

`MPCNTL_BIND` and `MPCNTL_BINDXCLU`

Binds (or exclusively binds) a specified process to the specified CPU(s). Exclusive binding is similar to regular binding except that a CPU with one or more processes exclusively bound to it will not execute any processes other than those in the bound set. This provides a primitive form of CPU/process set management. *arg* points to the data structure *mpcreq_t* filled out to indicate the target logical CPU(s)

and a valid PID. `MPCNTL_BIND` and `MPCNTL_BINDXCLU` will allow a process to bind to a set of CPUs if multiple bits are set in `mpc_cpu`. This has the effect of reserving all processors in the bound set since it prevents those CPUs from executing other processes that are not bound explicitly to them. Thus, this should be used with extreme caution.

On asymmetric systems, care must be exercised to ensure that a process is not being bound away from a resource that it needs (for example, hardware floating point unit), since the system will attempt to correct such conditions by altering or canceling the binding. Upon successful return from the call, the process is run on a CPU in the target set (for example, it is already bound).

The binding attributes are inherited across `fork(2)` and `exec(2)` system calls. These options may fail if a CPU in the target set has been altered by `MPCNTL_CPUINFOSET` to preclude the requested operation (for example, if the CPU is marked as unavailable (`CPIFS_NAVAIL`) or if the type of binding requested is precluded by `CPIFS_BINDOFF` or `CPIFS_XBINDOFF`).

`MPCNTL_BINDXCLU` requires superuser privileges.

`MPCNTL_UNBIND`

Effectively this is the inverse of `MPCNTL_BIND` and `MPCNTL_BINDXCLU` since it unbinds the specified process from its current CPU(s). If an exclusive binding is being removed and the target process is the last process exclusively bound to one or more CPUs, those processors then become public. `arg` points to the data structure `mpcreq_t`, filled out to indicate the target PID. Existing bindings may be altered by successive calls to `MPCNTL_BIND` and/or `MPCNTL_BINDXCLU`. This allows old bindings to be canceled and new bindings set up without calling `MPCNTL_UNBIND`.

`MPCNTL_AFFINOFF` and `MPCNTL_AFFINON`

Defeats or re-enables the normal affinity mechanisms which attempt to weakly bind a process to the last CPU it ran on. A process which has affinity turned off is eligible to run on the next available CPU (still considering all normal priority levels). `arg` points to the data structure `mpcreq_t`, filled out to indicate the target PID. The default state, `MPCNTL_AFFINON` is restored upon the issuance of an `exec(2)` system call but disabled affinity is inherited across `fork(2)` system calls. These calls are ineffectual if affinity is disabled through a system tunable parameter.

`MPCNTL_NOPRMPT` and `MPCNTL_PRMPT`

Disables/enables the preemptability of the current process and optionally makes the process immune to signals that it is catching. The intent of this option is to allow a process to run in a critical region of user code without being preempted. A possible application for this would be where a "spinlock" situation applied. This behavior may be modified by `argp` which may take on the values of:

`MPCNTL_NOPRMPT_SIGS`

The preemptability status of the process does not affect signal handling (for example, held/ignored signals remain held/ignored, caught signals will be processed, and ignored signals will be ignored).

The process cannot be preempted whenever it is running. This attribute persists across voluntary process blocking (for example, system call) and involuntary blocking (for example, page fault). If a non-preemptable process gives up the CPU, it will then compete with other processes normally to obtain access to a CPU. It then becomes once more non-preemptable until it again blocks.

`MPCNTL_NOPRMPT_NOSIGS`

Masked, ignored or caught signals will affect the process' signal handling. The unpreemptable behavior will apply only until the kernel is involved (voluntarily by a system call or involuntarily due to an exception). Interrupts do not affect the preemptability of the process directly and are not considered to be an involuntary entry into the kernel. The first call to `MPCNTL_NOPRMPT` results in a system call to prime the mechanism. Subsequent `MPCNTL_PRMPT` and `MPCNTL_NOPRMPT` calls result in library-only operations (for example, the library simply updates the value of this location in user space without need for entering the operating system). This optimization is intended to make the adjustment of process preemptability very fast. Upon success, the call returns the previous preemptability status of the process (>0 if not preemptable, 0 if preemptable) and otherwise returns -1.

Extended privileges are required to use this option. This attribute is not inherited across `fork()` or `exec()` system calls.

MPCNTL_GETFLAG

Returns the target process' `mpcntl` flags and binding information. In this case, `arg` points to the `mpcreq_t` data structure, which specifies the target PID and the `cpumask_t` data structure that is used to return the target process' binding information. Upon success, `MPCNTL_GETFLAG` returns the flags and the `mpc_cpu` field of the structure pointed to by `arg` is filled in with the target process' binding information. On failure, `-1` is returned.

The flags can be interpreted by looking at the `#define` statements in `sys/proc.h`. The relevant flags are `PBIND`, `PBINDXCLU`, `PAFFINOFF` and `PNOPRMP` (which is divided into `PNOPRMP``SIG` and `PNOPRMP``TSIG`).

MPCNTL_RDTIMER

Reads the system "high resolution" timer and returns the value of the timer to the caller. The timer units are approximately microseconds and the user must expect to deal with a "wrap-around condition" since there may only be limited amount of resolution.

The timer values are not meaningful with respect to wall clock time and should only be used to measure interval duration. For implementations that do not have hardware to support this capability the call may return `-1` to indicate that the functionality is not available. The parameter `arg` is ignored and extended privileges are not required to use this option.

MPCNTL_YIELD

Forces the calling process to voluntarily give up the CPU. The system call does not return until the caller is rescheduled. If there is nothing else to run or this process has the best priority on the system, the yield will return immediately.

MPCNTL_CPUCNT

Returns a count of the CPUs on the system. `arg` points to a `cpumask_t` in user-space which is filled out as a bit mask to indicate CPUs that are in service and not marked as not-available. The return value from the `mpcntl()` call indicates the total count of configured CPUs.

The logical CPU spectrum extends from 0 to (`cpucnt - 1`) without holes and all CPUs are configured and started at system boot time. If `arg` is null, only the count is returned.

MPCNTL_CPUINFOGET

Returns a filled out `cpuinfo_t` structure which gives specific additional information about a particular CPU. The `cpuinfo_t` structure is defined as follows:

```
typedef struct cpuinfo {
    cpuid_t      cpi_cpuid;          /* logical ID of this CPU */
    int          cpi_cpustatus;      /* alterable CPU state flags */
    int          cpi_cpuattr;        /* CPU attribute flags */
    enum cputype cpi_cputype;        /* type of processor */
    int          cpi_cpurev;         /* revision of CPU */
    enum fputype cpi_fputype;        /* type of FP coprocessor */
    int          cpi_fprev;          /* revision of FP coprocessor */
    enum fputype cpi_fptype2;        /* type of FP coprocessor [2] */
    int          cpi_fprev2;         /* revision of FP coprocessor [2] */
    int          cpi_cachsize;       /* size (bytes) of secondary cache */
    int          cpi_cpuclock;       /* Speed in MHz of the CPU */
    int          cpi_bind;           /* # processes explicitly bound */
    int          cpi_xbind;          /* # processes explicitly bound */
    int          cpi_drvbind;        /* # drivers bound to CPU */
    int          cpi_physcpuid;      /* physical cpu ID */
    int          cpi_physintbind;    /* physical device intrs bound
                                     to cpu */
    int          cpi_pad[8];         /* reserved */
}
```

```
    } cpui    nfo_t;
MPCNTL_CPUINFOSET
```

This option is used to set the *cpu_cpustatus* flags (for example, to disable binding on the CPU or to remove the CPU from service) and the remainder of the *cpuinfo_t* structure is ignored. Typically, one should execute `MPCNTL_CPUINFOGET` to populate the structure, alter the flags as appropriate, and issue the `MPCNTL_CPUINFOSET` command. The following flags are available:

`CPIFS_INSV`

The CPU is in service. Clearing the flag removes the CPU from service. This may not be supported by all hardware (normal operation has this flag set). Removing a CPU from service causes process bindings to that CPU to be cleared.

`CPIFS_NAVAIL`

The CPU is not available for executing processes but may or may not be in service (normal operation has this flag cleared). Marking a CPU not-available causes process bindings to that CPU to be cleared.

`CPIFS_BINDOFF`

The CPU is unavailable for normal binding operations (`MPCNTL_BIND`) but existing processes that are bound to the processor are not moved (normal operation has this flag cleared). This influences the inherited binding attributes (across `fork()` or `exec()` system calls) by causing them to be ignored until binding is again allowed on the CPU.

`CPIFS_XBINDOFF`

The CPU is unavailable for exclusive binding operations (`MPCNTL_BINDXCLU`) but existing processes that are bound to the CPU are not moved (normal operation has this flag cleared). This influences the inherited binding attributes (across `fork()` or `exec()` system calls) by causing them to be ignored until binding is again allowed on the CPU.

`CPIFS_CACHEOFF`

The designated CPU cache is turned off. This may not be supported by all hardware (normal operation has this flag cleared).

`CPIFS_IOINTROFF`

The designated CPU is not taking IO interrupts. This may not be supported by all hardware (normal operation has this flag cleared). The user must have superuser privileges to exercise this option.

RESULT

On success, `mpcntl()` returns zero or the return value when one is warranted (for example, `MPCNTL_RDTIMER`); on failure `mpcntl()` returns a `-1` and `errno` is set to indicate the cause of the error.

The following error code descriptions are function-specific. You will find a general description in [intro_prm2\(2\)](#) or in [errno\(5\)](#).

EINVAL

One or more of the arguments to `mpcntl()` are invalid. This may be caused by an invalid PID or flag.

EPERM

The user ID is not that of superuser (0) or has not been granted the appropriate privileges to perform the request. This may also occur when attempting to issue an option for a target PID that the caller does not have permissions to alter.

ENXIO

The option is not supported by the hardware, the CPU ID is out of range, or the CPU is not in a state that will allow the operation.

FAULT

A pointer to an argument (for example, *arg*) is illegal.

BUSY The CPU has a type of binding which explicitly disallows the options, or an exclusive operation is attempting to leave no CPUs in a non-exclusive state.

SEE ALSO

`getpid(2)`, `signal(2)`, `types(5)`.

NAME

mp:
 madd, msub, mult, mdiv, mcmp, min, mout, pow, gcd, rpow, msqrt, sdiv, itom, xtom,
 mtox, mfree - multipleprecision integer arithmetic

SYNOPSIS

```
/usr/ucb/cc [ flag ... ] file ... -lucb -lmp
#include <mp.h>
madd(a, b, c)
MINT *a, *b, *c;
msub(a, b, c)
MINT *a, *b, *c;
mult(a, b, c)
MINT *a, *b, *c;
mdiv(a, b, q, r)
MINT *a, *b, *q, *r;
mcmp(a,b)
MINT *a, *b;
min(a)
MINT *a;
mout(a)
MINT *a;
pow(a, b, c, d)
MINT *a, *b, *c, *d;
gcd(a, b, c)
MINT *a, *b, *c;
rpow(a, n, b)
MINT *a, *b;
short n;
msqrt(a, b, r)
MINT *a, *b, *r;
sdiv(a, n, q, r)
MINT *a, *q;
short n, *r;
MINT *itom(n)
short n;
MINT *xtom(s)
char *s;
char *mtox(a)
MINT *a;
void mfree(a)
MINT *a;
```

DESCRIPTION

These routines perform arithmetic on integers of arbitrary length. The integers are stored using the defined type `MINT`. Pointers to a `MINT` should be initialized using the function `itom`, which sets the initial value to n . Alternatively, `xtom` may be used to initialize a `MINT` from a string of hexadecimal digits. `mfree` may be used to release the storage allocated by the `itom` and `xtom` routines.

`madd`, `msub` and `mult` assign to their third arguments the sum, difference, and product, respectively, of their first two arguments. `mdiv` assigns the quotient and remainder, respectively, to its third and fourth arguments. `sdiv` is like `mdiv` except that the divisor is an ordinary integer. `msqrt` produces the square root and remainder of its first argument. `mcmp` compares the values of its arguments and returns 0 if the two values are equal, >0 if the first argument is greater than the second, and <0 if the second argument is greater than the first. `rpow` calculates a raised to the power b , while `pow` calculates this reduced modulo m . `min` and `mout` do decimal input and output. `gcd` finds the greatest common divisor of the first two arguments, returning it in the third argument. `mtox` provides the inverse of `xtom`. To release the storage allocated by `mtox`, use `free`.

RETURN VALUE

Illegal operations and running out of memory produce messages and core images.

FILES

`/usr/ucblib/libmp.a`

SEE ALSO

`malloc(3C)`, `malloc(3X)`.

NAME

nullptr - null pointer control

SYNOPSIS

```
cc [ flag ... ] file ... -lext [ library ... ]  
#include <sys/sysmips.h>  
void nullptrcontrol(int trapnullptrs, int inherit);
```

DESCRIPTION

nullptr controls how null pointers are treated for this process and/or its descendants. By default read references to address 0 (such as dereferencing a null pointer) are silently satisfied by the kernel.

nullptrcontrol allows the user control over this effect for specific processes and their descendants.

If the first argument, *trapnullptrs*, is non-zero, dereferences of address 0 will not be satisfied for this process. This will typically generate a SIGSEGV signal.

If the first argument, *trapnullptrs*, is zero, dereferences of address 0 are satisfied for this process. This is the default.

If the second argument, *inherit*, is set, the value of the first argument is propagated to the processes children in addition to effecting the process.

RETURN VALUE

A return value of 0 is expected. A return value of -1 means some unknown error occurred.

SEE ALSO

[launchit\(1M\)](#).

spinlock(3X)**NAME**

`spinlock`, `initspin`, `spinunlock`, `cspinlock`, `yield` - user synchronization primitives

SYNOPSIS

```
cc [ flag ... ] file ... -lmproc
#include <ulocks.h>

□
□
void initspin(spinlock_t *lck, long spincnt);
□
□
void spinlock(spinlock_t *lck);
□
□
void spinunlock(spinlock_t *lck);
□
□
int cspinlock(spinlock_t *lck);
□
□
void yield();
```

DESCRIPTION

The most fundamental synchronization primitive for a multiprocessor is the simple spin lock, which is the essence of other synchronization primitives. This primitive is dependent upon an atomic read-modify-write operation that supplied by the system. Different CPU architectures provide this capability in different ways and various system architectures support this capability by varying means. Typically, these low-level operations are coded in machine language. To hide the vagaries of these implementations across machines a basic set of primitives is provided.

Typically, these primitives will be used by cooperating processes that share address space (such as shared memory) to control access to shared resources and data structures. The data structure that is the target of the primitives is shown below.

```
typedef struct spinlock {
    volatile int    sl_lock;
    long           sl_scount;
} spinlock_t;
```

The initialization primitive, `initspin()`, initializes a target lock for use by the primitives. `initspin()` takes two arguments: `lck` is a pointer to the target `spinlock_t` data structure and `spincnt` is the number of spins that will be attempted on a lock acquisition, `spinlock()`, before the calling process yields the CPU. Setting `spincnt` to `-1` implies an infinite number of spins (the process will not yield the CPU voluntarily).

The `spinlock()` acquisition function attempts to acquire the lock designated by `lck`. This function will either spin for some number iterations (determined by the `spincnt` initialization parameter) before giving up and yielding the CPU or spins indefinitely (`spincnt` is `-1`) until the lock is acquired bearing in mind that the operating system may choose to time slice the process out at any time.

The `spinunlock()` function releases (unlocks) the lock designated by `lck` and has the opposite effect to `spinlock()`.

The conditional spin lock operation, `cspinlock()`, attempts to acquire the target lock, `lck` and returns status indicating the disposition of the lock. If the lock is already held, the primitive does not spin on the acquisition

and failure is returned (non-zero). If the lock is available, it is acquired and success is returned (zero).

The `yield()` primitive allows the caller to relinquish the CPU (implemented through the `mpcntl_yield` option to `mpcnt()`) without actually blocking. This function may return immediately if the calling process has high enough priority to allow it to continue execution.

SEE ALSO

`mpcntl(3X)`.

NAME

`sputl`, `sgetl` - access long integer data in a machine-independent fashion

SYNOPSIS

```
cc [ flag ... ] file ... -lmld
#include <ldfcn.h>
void sputl(long value, char *buffer);
long sgetl(const char *buffer);
```

DESCRIPTION

`sputl()` takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

`sgetl()` retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of `sputl()` and `sgetl()` provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

NAME

statis - kernel statistics reporting and control

SYNOPSIS

```
cc [ flag ... ] file ... -lext [ library ... ]
#include <sys/types.h>
#include <sys/statis.h>
statis(char fac[STATIS_FACLEN], int cmd, void *buf,
int len);
```

DESCRIPTION

`statis()` performs a variety of reporting and control functions for kernel statistics and performance monitoring facilities. *fac* is the name of a statistics facility. *cmd* specifies the control or reporting function to be performed. *buf* and *len* are the address and size of a user buffer to receive reported values or to specify control values. The values reported or control values required are specific to each facility.

The return value from the call is either the number of bytes copied into *buf* or -1 if there has been an error.

The available values for *cmd* are:

STATIS_ON

Turn on statistics gathering for *fac*.

STATIS_OFF

Turn off statistics gathering for *fac*.

STATIS_GET

Obtain statistics for *fac*. *buf* should point to a memory area of at least appropriate size for the facility and *len* should specify the size of the memory area.

STATIS_CLR

Clear the statistics structures for *fac*.

STATIS_SIZ

Obtain the size of the statistics structure maintained by *fac*. *buf* should point to a variable of type `size_t` and *len* should be `sizeof(size_t)`.

STATIS_FLG

Obtain the current status flags associated with *fac*. *buf* should point to a variable of type `statis_flg_t` and *len* should be `sizeof(statis_flg_t)`.

STATIS_TIM

Obtain the time in seconds at which statistics collection began for *fac*. *buf* should point to a variable of type `time_t` and *len* should be `sizeof(time_t)`.

STATIS_TAB

Return the next *fac* following the specified *fac*. A loop using **STATIS_TAB** may obtain the name of each statistics facility in the kernel. The loop should start with `fac[0] = '\0'` and should terminate when the return value of the system call is zero. For each call *buf* should point to an area of memory of at least `STATIS_FACLEN` bytes in size.

STATIS_NOB

Turn a control knob associated with *fac*, if one exists. The buffer contents required and returned are specific to the facility.

The status flag returned by **STATIS_FLG** is of arithmetic type `statis_flg_t` and is a bitmask with the following bit assignments:

STATIS_PERCPU

On a multi-cpu machine, a separate statistics structure is kept for each cpu.

STATIS_DEFON

The collection of this statistic defaults to on when the system boots.

STATIS_ACCUM

This statistic facility is currently accumulating statistics. In other words, the facility has been turned on using **STATIS_ON**.

ERRORS

If **statis()** fails it will return **-1** and **errno** will be set to one of the following:

EINVAL

fac is not a known facility name, *cmd* is not one of the values described above, an attempt was made to get statistics that were not turned on, or a knob was turned on a facility for which no knob exists.

EFAULT

buf points to an invalid memory area.

The facility may specify other **errno** values upon return from a **STATIS_NOB** call to **statis()**.

EXAMPLES

The following example prints the facility table.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/statis.h>
int
tab(void)
{
    int ret;
    char fac[STATIS_FACLEN];
    fac[0] = '\0';
    while ((ret = statis(fac, STATIS_TAB, fac, sizeof(fac))) > 0) {
        printf("%s\n", fac);
    }
    if (ret < 0) {
        fprintf(stderr, "couldn't get facility table");
        return -1;
    }
    return 0;
}
```

The following example obtains the **sysinfo** facility statistics.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/statis.h>
#include <sys/sysinfo.h>
void
print_one_cpu_sysinfo(struct sysinfo *si, int cpu)
{
    printf("Some statistics for CPU %d \n", cpu);
    printf(" ..... \n");
    printf("  bread   = %d \n", si->bread);
    printf(" bwrite   = %d \n", si->bwrite);
    printf("  lread   = %d \n", si->lread);
    printf(" lwwrite  = %d \n", si->lwwrite);
    printf("  syscall = %d \n", si->syscall);
    printf(" ..... \n");
}
int
print_sysinfo(void)
{
```

```

    unsigned        ncpu, cpu;
    statis_flg_t     flags;
    size_t           size, bufsize;
    void             *buf;
    struct sysinfo    *si;
    if (statis("ncpu", STATIS_GET, &ncpu, sizeof(ncpu)) == -1)
        ncpu = 1;
    if (statis("sysinfo", STATIS_FLG,
               &flags, sizeof(statis_flg_t)) == -1) {
        fprintf(stderr, "couldn't get sysinfo flags");
        return -1;
    }
    if (statis("sysinfo", STATIS_SIZ,
               &size, sizeof(size)) == -1) {
        fprintf(stderr, "couldn't get sysinfo size");
        return -1;
    }
    if (sizeof(struct sysinfo) != size) {
        fprintf(stderr, "kernel doesn't match, recompile me!\n");
        "For 64bit kernel, compile application with 64bit (-Klp64)\n");
        return -1;
    }
    /*
     * If the kernel is collecting the statistics separately for each
     * cpu, then STATIS_GET will give us size bytes for each cpu.
     */
    if (flags & STATIS_PERCPU) {
        bufsize = size * ncpu;
    } else {
        bufsize = size;
        ncpu = 1;
    }
    buf = (void *) malloc(bufsize);
    if (statis("sysinfo", STATIS_GET, buf, bufsize) != bufsize) {
        fprintf(stderr, "couldn't get sysinfo statistics");
        return -1;
    }
    si = (struct sysinfo *) buf;
    for (cpu = 0; cpu < ncpu; cpu++) {
        print_one_cpu_sysinfo(si+cpu, cpu);
    }
    free(buf);
    return 0;
}

```

SEE ALSO

[statis\(5\)](#), [statis\(8\)](#).

Dateiformate(4)**Introduction****NAME**

intro_prm4 - introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C structure declarations for the file formats are given where applicable. Usually, the header files containing these structure declarations can be found in the directories `/usr/include` or `/usr/include/sys`. For inclusion in C language programs, however, the syntax `#include <filename.h>` or `#include <sys/filename.h>` should be used.

File formats and include files

Programmer's Reference Manual

a.out(4)

NAME

`a.out` - ELF (Executable and Linking Format) files

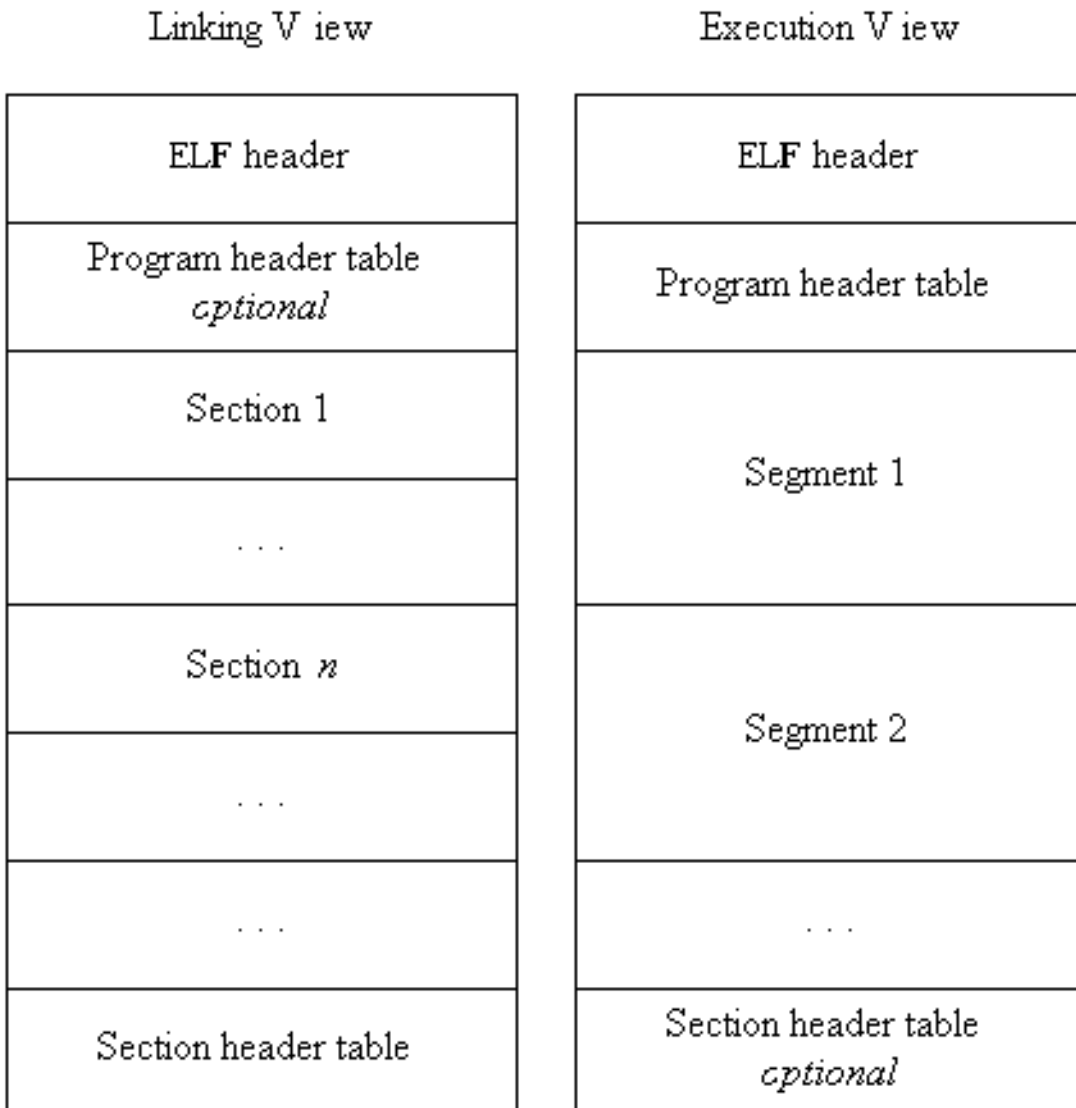
SYNOPSIS

```
#include <elf.h>
```

DESCRIPTION

The filename `a.out` is the default output filename from the link editor, `ld(1)`. The link editor will make an `a.out` executable if there were no errors in linking.

Programs that manipulate ELF files may use the library that `elf(3E)` describes. An overview of the file format follows. For more complete information, see the references given below. □



An ELF header resides at the beginning and holds a "road map" describing the file's organization. Sections hold the object file information for the linking view: instructions, data, symbol table, relocation information, and so on. Segments hold the object file information for the program execution view. A segment may contain one or more sections.

A program header table, if present, tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table; relocatable files do not need one. A section header table contains information describing the file's sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, etc. Files used during linking must have a section header table; other object files may or may not have one.

Although the figure shows the program header table immediately after the ELF header, and the section header table following the sections, actual files may differ. Moreover, sections and segments have no specified order. Only the ELF header must appear at the start of the file.

When an `a.out` file is loaded into memory for execution, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), and a stack. The text segment is not writable by the program; if other processes are executing the same `a.out` file, the processes will share a single text segment.

The data segment starts at the next maximal page boundary past the last text address. When the process

image is created, the part of the file holding the end of text and the beginning of data may appear twice. The duplicated chunk of text that appears at the beginning of data is never executed; it is duplicated so that the operating system may bring in pieces of the file in multiples of the actual page size without having to realign the beginning of the data section to a page boundary. Therefore, the first data address is the sum of the next maximal page boundary past the end of text plus the remainder of the last text address divided by the maximal page size. If the last text address is a multiple of the maximal page size, no duplication is necessary. The stack is automatically extended as required. The data segment is extended as requested by the `brk(2)` system call.

SEE ALSO

`cc(1)`, `ld(1)`, `brk(2)`, `elf(3E)`.

The "Object files" chapter in the *Guide to Tools for Programming in C*.

NAME

ar - archive file format

SYNOPSIS

```
#include <ar.h>
```

DESCRIPTION

The archive command `ar` is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor `ld(1)`.

Each archive begins with the archive magic string.

```
#define ARMAG "!<arch>\n" /* magic string */
#define SARMA 8           /* length of magic string */
```

Following the archive magic string are the archive file members. Each file member is preceded by a file member header which is of the following format:

```
#define ARFMAG "\n" /* header trailer string */

struct    ar_hdr          /* file member header */
{
    char    ar_name[16];   /* '/' terminated file member name */
    char    ar_date[12];   /* file member date */
    char    ar_uid[6];     /* file member user identification */
    char    ar_gid[6];     /* file member group identification */
    char    ar_mode[8];    /* file member mode (octal) */
    char    ar_size[10];   /* file member size */
    char    ar_fmag[2];
    /* header trailer string */
};
```

All data items in the file member headers are printable ASCII code. The numeric information contained in the headers is stored as decimal numbers (except for `ar_mode` which is in octal). Thus, if the archive contains printable files, it is also printable.

If the file member name fits, the `ar_name` field contains the name directly, and is terminated by a slash (/) and padded with blanks on the right. If the member's name does not fit, `ar_name` contains a slash (/) followed by a decimal representation of the name's offset in the archive string table described below.

The `ar_date` field is the modification date of the file at the time of its insertion into the archive. Common format archives can be moved from system to system as long as the portable archive command `ar` is used.

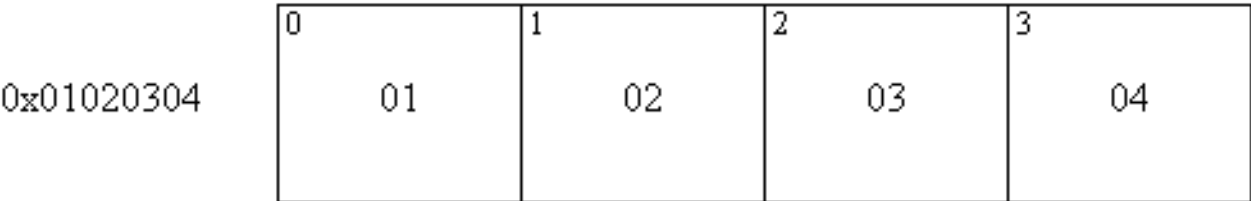
Each archive file member begins on an even byte boundary; a newline is inserted between files if necessary. Nevertheless, the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

Each archive that contains object files [see `a.out(4)`] includes an archive symbol table. This symbol table is used by the link editor `ld` to determine which archive members must be loaded during the link edit process. The archive symbol table (if it exists) is always the first file in the archive (but is never listed) and is automatically created and/or updated by `ar`.

The archive symbol table has a zero length name (i.e., `ar_name[0]` is "/"), `ar_name[1]` == ' ', etc.). All "words" in this symbol table have four bytes, using the machine-independent encoding shown below. All machines use the encoding described here for the symbol table, even if the machine's "natural" byte order is different.

□



The contents of this file are as follows:

1. The number of symbols. Length: 4 bytes.
2. The array of offsets into the archive file. Length: 4 bytes * "the number of symbols".
3. The name string table. Length: *ar_size* – 4 bytes * ("the number of symbols" + 1).

As an example, the following symbol table defines 4 symbols. The archive member at file offset 114 defines *name* and *object*. The archive member at file offset 426 defines *function* and a second version of *name*. □

Offset	+0	+1	+2	+3	
0	4				4 offset entries
4	114				name
8	114				object
12	426				function
16	426				name
20	n	a	m	e	
24	\0	o	b	j	
28	e	c	t	\0	
32	f	u	n	c	
36	t	i	o	n	
40	\0	n	a	m	
44	e	\0			

The number of symbols and the array of offsets are managed with `sget1()` and `sput1()`. The string table contains exactly as many null terminated strings as there are elements in the offsets array. Each offset from the array is associated with the corresponding name from the string table (in order). The names in the string table are all `%n 5` the defined global symbols found in the common object files in the archive. Each offset is the location of the archive header for the associated symbol.

If some archive member's name is more than 15 bytes long, a special archive member contains a table of filenames, each followed by a slash and a newline. This string table member, if present, will precede all "normal" archive members. The special archive symbol table is not a "normal" member, and must be first if it exists. The `ar_name` entry of the string table's member header holds a zero length name `ar_name[0] == '/'`, followed by one trailing slash (`ar_name[1] == '/'`), followed by blanks (`ar_name[2] == ' ' □`, etc.). Offsets into the string table begin at zero. Example `ar_name` values for short and long filenames appear below.

□

Offset	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	f	i	l	e	_	n	a	m	e	_
10	s	a	m	p	l	e	/	\n	l	o
20	n	g	e	r	f	i	l	e	n	a
30	m	e	x	a	m	p	l	e	/	\n

□

Name	ar_name	Note
short-name	short-name	Not in string table
file_name_sample	/0	Offset 0 in string table
longerfilenameexample	/18	Offset 18 in string table

NOTES

`strip()` will remove all archive symbol entries from the header. The archive symbol entries must be restored via the `-ts` options of the `ar` command before the archive can be used with the link editor `ld`.

SEE ALSO

`ar(1)`, `ld(1)`, `strip(1)`, `sputl(3X)`, `a.out(4)`.

NAME

core - core image file

DESCRIPTION

The Reliant UNIX system writes out a core image of a process when the process is terminated due to the receipt of some signals [see [signal\(2\)](#) and [signal\(5\)](#)]. The most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called `core` and is written in the process's working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a copy of the system's per-user data for the process.

The second section of the core image contains a copy of the data segment of the terminated process.

The third section of the core image is a copy of the stack segment of the terminated process.

After the data and stack core image follows an optional section for the shared memory.

SEE ALSO

[crash\(1M\)](#), [getrlimit\(2\)](#), [setuid\(2\)](#), [signal\(2\)](#), [ulimit\(2\)](#), [signal\(5\)](#).

NAME

`limits` - implementation-dependent constants

SYNOPSIS

```
#include <limits.h>
```

DESCRIPTION

The `<limits.h>` header defines various symbolic names. Different categories of names are described below.

The names represent various limits on resources that the system imposes on applications.

Implementations may choose any appropriate value for each limit, provided it is not more restrictive than the values listed in the "Minimum Acceptable Value" line below. Symbolic constant names beginning with `_POSIX` may be found in `<unistd.h>` [see [unistd\(4\)](#)].

Applications should not assume any particular value for a limit. To achieve maximum portability, an application should not require more resource than the quantity listed in the "Minimum Acceptable Value" line. However, an application wishing to avail itself of the full amount of a resource available on an implementation may make use of the value given in `<limits.h>` on that particular system, by using the symbolic names listed below. It should be noted, however, that many of the listed limits are not invariant, and at run time, the value of the limit may differ from those given in this header, for the following reasons:

- the limit is pathname-dependent
- the limit differs between the compile and run-time machines.

For these reasons, an application may use the `fpathconf(2)`, `pathconf(2)` and `sysconf(3C)` functions to determine the actual value of a limit at run time.

The items in the list ending in `_MIN` give the most negative values that the mathematical types are guaranteed to be capable of representing. Numbers of a more negative value may be supported on some systems, as indicated by the `<limits.h>` header on the system, but applications requiring such numbers are not guaranteed to be portable to all systems.

Run-time Invariant Values (Possibly Indeterminate)

A definition of one of the symbolic names in the following list will be omitted from `<limits.h>` on specific implementations where the corresponding value is equal to or greater than the stated minimum, but is indeterminate.

This might depend on the amount of available memory space on a specific instance of a specific implementation. The actual value supported by a specific instance will be provided by the `sysconf()` function.

ARG_MAX

Maximum length of argument to the `exec` functions including environment data.

Minimum Acceptable Value: `_POSIX_ARG_MAX`

ATEXIT_MAX

Maximum number of functions that may be registered with `atexit()`.

Minimum Acceptable Value: 32

CHILD_MAX

Maximum number of simultaneous processes per real user ID.

Minimum Acceptable Value: 25

IOV_MAX

Maximum number of `iovec` structures that one process has available for use with `readv()` or `writev()`.

Minimum Acceptable Value: `_XOPEN_IOV_MAX`

`OPEN_MAX`

Maximum number of files that one process can have open at any one time.

Minimum Acceptable Value: 20

`PAGESIZE`

Size in bytes of a page.

Minimum Acceptable Value: 1

`PAGE_SIZE`

Same as `PAGESIZE`. If either `PAGESIZE` or `PAGE_SIZE` is defined, the other will be defined with the same value.

`PASS_MAX`

Maximum number of significant bytes in a password (not including terminating null).

Minimum Acceptable Value: 8

`PTHREAD_DESTRUCTOR_ITERATIONS`

Maximum number of attempts made to destroy a thread's thread-specific data values on thread exit.

Minimum Acceptable Value: `_POSIX_THREAD_DESTRUCTOR_ITERATIONS`

`PTHREAD_KEYS_MAX`

Maximum number of data keys that can be created by a process.

Minimum Acceptable Value: `_POSIX_THREAD_KEYS_MAX`

`PTHREAD_STACK_MIN`

Minimum size in bytes of thread stack storage.

Minimum Acceptable Value: 0

`PTHREAD_THREADS_MAX`

Maximum number of threads that that can be created per process.

Minimum Acceptable Value: `_POSIX_THREAD_THREADS_MAX`

`STREAM_MAX`

The number of streams that one process can have open at one time. If defined, it has the same value as `FOPEN_MAX` [see [stdio\(3S\)](#)].

Minimum Acceptable Value: `_POSIX_STREAM_MAX`

`TZNAME_MAX`

Maximum number of bytes supported for the name of a time zone (not the `TZ` variable).

Minimum Acceptable Value: `_POSIX_TZNAME_MAX`

Pathname Variable Values

The values in the following list may be constants within an implementation or may vary from one pathname to another. For example, file systems or directories may have different characteristics.

A definition of one of the values will be omitted from the `<limits.h>` on specific implementations where the corresponding value is equal to or greater than the stated minimum, but where the value can vary depending on the file to which it is applied. The actual value supported for a specific pathname will be provided by the `pathconf()` function.

`FILESIZEBITS`

Minimum number of bits needed to represent, as a signed integer value, the maximum size of a regular file allowed in the specified directory.

Minimum Acceptable Value: 32

`LINK_MAX`

Maximum number of links to a single file.

Minimum Acceptable Value: `_POSIX_LINK_MAX`

`MAX_CANON`

Maximum number of bytes in a terminal canonical input line.

Minimum Acceptable Value: `_POSIX_MAX_CANON`

`MAX_INPUT`

Minimum number of bytes for which space will be available in a terminal input queue; therefore, the maximum number of bytes a portable application may require to typed as input before reading them.

Minimum Acceptable Value: `_POSIX_MAX_INPUT`

`NAME_MAX`

Maximum number of bytes in a filename, (not including terminating null).

Minimum Acceptable Value: `_POSIX_NAME_MAX`

`PATH_MAX`

Maximum number of bytes in a pathname, including the terminating null character.

Minimum Acceptable Value: `_POSIX_PATH_MAX`

`PIPE_BUF`

Maximum number of bytes that is guaranteed to be atomic when writing to a pipe.

Minimum Acceptable Value: `_POSIX_PIPE_BUF`

Run-time Inceasable Values

The magnitude limitations in the following list will be fixed by specific implementations. An application should assume that the value supplied by `<limits.h>` in a specific implementation is the minimum that pertains whenever the application is run under that implementation. A specific instance of a specific implementation may increase the value relative to that supplied by `<limits.h>` for that implementation. The actual value supported by a specific instance will be provided by the `sysconf()` function.

`BC_BASE_MAX`

Maximum obase values allowed by the `bc` utility.

Minimum Acceptable Value: `_POSIX2_BC_BASE_MAX`

`BC_DIM_MAX`

Maximum number of elements permitted in an array by the `bc` utility.

Minimum Acceptable Value: `_POSIX2_BC_DIM_MAX`

`BC_SCALE_MAX`

Maximum scale value allowed by the `bc` utility.

Minimum Acceptable Value: `_POSIX2_BC_SCALE_MAX`

`BC_STRING_MAX`

Maximum length of a string constant accepted by the `bc` utility.

Minimum Acceptable Value: `_POSIX2_BC_STRING_MAX`

`COLL_WEIGHTS_MAX`

Maximum number of weights that can be assigned to an entry of the `LC_COLLATE` order keyword in the locale definition file.

Minimum Acceptable Value: `_POSIX2_COLL_WEIGHTS_MAX`

`EXPR_NEST_MAX`

Maximum number of expressions that can be nested within parentheses by the `expr` utility.

Minimum Acceptable Value: `_POSIX2_EXPR_NEST_MAX`

`LINE_MAX`

Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing newline.

Minimum Acceptable Value: `_POSIX2_LINE_MAX`

`NGROUPS_MAX`

Maximum number of simultaneous supplementary group IDs per process.

Minimum Acceptable Value: 8

RE_DUP_MAX

Maximum number of repeated occurrences of a regular expression permitted when using the interval notation `\{m,n\}`.

Minimum Acceptable Value: `_POSIX2_RE_DUP_MAX`

Minimum Values

The symbolic constants in the following list are defined in `<limits.h>` with the values shown. These are symbolic names for the most restrictive value for certain features on a system conforming to this document. Related symbolic constants are defined elsewhere which reflect the actual implementation and which need not be as restrictive. A conforming implementation will provide values at least this large. A portable application must not require a larger value for correct operation.

_POSIX_ARG_MAX

Maximum length of argument to the `exec` functions including environment data.

Value: 4 096

_POSIX_CHILD_MAX

Maximum number of simultaneous processes per real user ID.

Value: 6

_POSIX_LINK_MAX

Maximum number of links to a single file.

Value: 8

_POSIX_MAX_CANON

Maximum number of bytes in a terminal canonical input queue.

Value: 255

_POSIX_MAX_INPUT

Maximum number of bytes allowed in a terminal input queue.

Value: 255

_POSIX_NAME_MAX

Maximum number of bytes in a file name (not including terminating null).

Value: 14

_POSIX_NGROUPS_MAX

Maximum number of simultaneous supplementary group IDs per process.

Value: 0

_POSIX_OPEN_MAX

Maximum number of files that one process can have open at any one time.

Value: 16

_POSIX_PATH_MAX

Maximum number of bytes in a pathname.

Value: 16

_POSIX_PIPE_BUF

Maximum number of bytes that is guaranteed to be atomic when writing to a pipe.

Value: 512

_POSIX_THREAD_DESTRUCTOR_ITERATIONS

The number of attempts made to destroy a thread's thread-specific data values on thread exit.

Value: 4

_POSIX_THREAD_KEYS_MAX

The number of data keys per process.

Value: 128

_POSIX_THREAD_THREADS_MAX

The number of threads per process.

Value: 64

`_POSIX_SSIZE_MAX`

The value that can be stored in an object of type `ssize_t`.

Value: 32 767

`_POSIX_STREAM_MAX`

The number of streams that one process can have open at one time.

Value: 8

`_POSIX_TZNAME_MAX`

Maximum number of bytes supported for the name of a time zone (not of `TZ` variable).

Value: 3

`_POSIX2_BC_BASE_MAX`

Maximum obase values allowed by the `bc` utility.

Value: 99

`_POSIX2_BC_DIM_MAX`

Maximum number of elements permitted in an array by the `bc` utility.

Value: 2 048

`_POSIX2_BC_BASE_MAX`

Maximum scale value allowed by the `bc` utility.

Value: 99

`_POSIX2_BC_STRING_MAX`

Maximum length of a string constant accepted by the `bc` utility.

Value: 1 000

`_POSIX2_COLL_WEIGHTS_MAX`

Maximum number of weights that can be assigned to an entry of the `LC_COLLATE` order keyword in the locale definition file.

Value: 2

`_POSIX2_EXPR_NEST_MAX`

Maximum number of expressions that can be nested within parentheses by the `expr` utility.

Value: 32

`_POSIX2_LINE_MAX`

Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing newline.

Value: 2 048

`_POSIX2_RE_DUP_MAX`

Maximum number of repeated occurrences of a regular expression permitted when using the interval notation `\{m,n\}`.

Value: 255

`_XOPEN_IOV_MAX`

Maximum number of `iovec` structures that one process has available for use with `readv()` or `writev()`.

Value: 16

Numerical limits

The values in the following lists are defined in `<limits.h>` and will be constant expressions suitable for use in `#if` preprocessing directives. Moreover, except for `CHAR_BIT`, `DBL_DIG`, `DBL_MAX`, `FLZ_DIG`, `FLT_MAX`, `LONG_BIT`, `WORD_BIT` and `MB_LEN_MAX`, the symbolic names will be defined as expressions of the correct

type.

If the value of an object of type `char` is treated as a signed integer when used in an expression, the value of `CHAR_MIN` is the same as that of `SCHAR_MIN` and the value of `CHAR_MAX` is the same as that of `SCHAR_MAX`. Otherwise, the value of `CHAR_MIN` is 0 and the value of `CHAR_MAX` is the same as that of `UCHAR_MAX`.

`CHAR_BIT`

Number of bits in a type `char`.

Minimum Acceptable Value: 8

`CHAR_MAX`

Maximum value of a type `char`.

Minimum Acceptable Value: `UCHAR_MAX` or `SCHAR_MAX`

`DBL_DIG`

Digits of precision of a type `double`.

Minimum Acceptable Value: 10

`DBL_MAX`

Maximum value of a type `double`.

Minimum Acceptable Value: $1E + 37$

`FLT_DIG`

Digits of precision of a type `float`.

Minimum Acceptable Value: 6

`FLT_MAX`

Maximum value of a `float`.

Minimum Acceptable Value: $1E + 37$

`INT_MAX`

Maximum value of an `int`.

Minimum Acceptable Value: 2 147 483 647

`LONG_BIT`

Number of bits in a `long int` (32-bit system).

Minimum Acceptable Value: 32

`LONG_BIT`

Number of bits in a `long int` (64-bit system).

Minimum Acceptable Value: 64

`LONG_MAX`

Maximum value of a `long int` (32-bit system).

Minimum Acceptable Value: +2 147 483 647

`LONG_MAX`

Maximum value of a `long int` (64-bit system).

Minimum Acceptable Value: 9 223 372 036 854 775 807

`LONG_LONG_MAX`

Maximum value of a `long int`.

Minimum Acceptable Value: 9 223 372 036 854 775 807

`MB_LEN_MAX`

Maximum number of bytes in a character, for any supported locale.

Minimum Acceptable Value: 1

`SCHAR_MAX`

Maximum value of a type `signed char`.

Minimum Acceptable Value: 127

`SHRT_MAX`

Maximum value of a type `short`.

Minimum Acceptable Value: 32 767

`SSIZE_MAX`

Maximum value of an object of type `ssize_t`.

Minimum Acceptable Value: `_POSIX_SSIZE_MAX`

`UCHAR_MAX`

Maximum value of a type `unsigned char`.

Minimum Acceptable Value: 255

`UINT_MAX`

Maximum value of a type `unsigned int`.

Minimum Acceptable Value: 4 294 967 295

`ULONG_MAX`

Maximum value of a type `unsigned long int` (32-bit system).

Minimum Acceptable Value: 4 294 967 295

`ULONG_MAX`

Maximum value of a type `unsigned long int` (64-bit system).

Minimum Acceptable Value: 18 446 744 073 709 551 615

`ULONGLONG_MAX`

Maximum value of a type `unsigned long int`.

Minimum Acceptable Value: 18 446 744 073 709 551 615

`USHRT_MAX`

Maximum value for a type `unsigned short int`.

Minimum Acceptable Value: 65 535

`WORD_BIT`

Number of bits in a word or type `int`.

Minimum Acceptable Value: 16

`CHAR_MIN`

Minimum value of a type `char`.

Minimum Acceptable Value: `SCHAR_MIN` or 0

`INT_MIN`

Minimum value of a type `int`.

Minimum Acceptable Value: -2 147 483 647

`LONG_MIN`

Minimum value of a type `long int` (32-bit system).

Minimum Acceptable Value: -2 147 483 647-1

`LONG_MIN`

Minimum value of a type `long int` (64-bit system).

Minimum Acceptable Value: -9 223 372 036 854 775 807-1

`LONGLONG_MIN`

Minimum value of a type `long int`.

Minimum Acceptable Value: -9 223 372 036 854 775 807-1

`SCHAR_MIN`

Minimum value of a type `signed char`.

Minimum Acceptable Value: -127

`SHRT_MIN`

Minimum value of a type `short`.

Minimum Acceptable Value: -32 767

Other Invariant Values

The following constants are defined on all systems in `<limits.h>`.

`CHARCLASS_NAME_MAX`

Maximum number of bytes in a character class name.

Minimum Acceptable Value: 14

`NL_ARGMAX`

Maximum value of digit in calls to the `print()` and `scanf()` functions.

Minimum Acceptable Value: 9

`NL_LANGMAX`

Maximum number of bytes in a `LANG` name.

Minimum Acceptable Value: 14

`NL_MSGMAX`

Maximum message number.

Minimum Acceptable Value: 32 767

`NL_NMAX`

Maximum number of bytes in an N-to-1 collation mapping.

Minimum Acceptable Value: not defined

`NL_SETMAX`

Maximum set number.

Minimum Acceptable Value: 255

`NL_TEXTMAX`

Maximum number of bytes in a message string.

Minimum Acceptable Value: `_POSIX2_LINE_MAX`

`NZERO`

Default process priority.

Minimum Acceptable Value: 20

`TMP_MAX`

Minimum number of unique pathnames generated by `tmpnam()`. Maximum number of times an application can call `tmpnam()` reliably.

Minimum Acceptable Value: 10 000

`TMP_MAX` is moved to `<stdio.h>` [see [stdio\(5\)](#)] to align with the ISO C standard.

`DBL_DIG`, `DBL_MAX`, `DBL_MIN`, `FLT_DIG`, `FLT_MAX`, and `FLT_MIN` are moved to `<float.h>` [see [float\(5\)](#)].

`DBL_MIN` and `FLT_MIN` are withdrawn since the values previously defined here were in direct conflict with the values in `<float.h>` as required by the ISO C standard.

SEE ALSO

[fpathconf\(2\)](#), [pathconf\(2\)](#), [sysconf\(3C\)](#), [unistd\(4\)](#), [float\(5\)](#), [lfs\(5\)](#), [stdio\(5\)](#).

NAME

sccsfile - format of SCCS file

DESCRIPTION

An SCCS (Source Code Control System) file is an ASCII file. It consists of six logical parts: the checksum, the delta table (information about each delta), user names (login names and/or numerical group IDs of users who may add deltas), flags (definitions of internal keywords), comments (arbitrary descriptive information about the file), and the body (the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the ASCII SOH (start of heading) character (octal 001). This character is hereafter referred to as the control character and will be represented graphically as @. Any line described below that is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form DDDDD represent a five-digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

@hDDDDD

The value of the checksum is the sum of all characters, except those of the first line. The @h provides a magic number of (octal) 064001, depending on byte order.

Delta table

The delta table consists of a variable number of entries of one of the following forms:

@s DDDDD/DDDDD/DDDDD@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD@i DDDDD ...

@x DDDDD ...

@g DDDDD ...

@m <MR number>

...

@c <comments> ...

...

@e

The first line (@s) contains the number of lines inserted/deleted/unchanged, respectively. The second line (@d) contains the type of the delta (normal: D or removed: R), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The @i, @x, and @g lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The @m lines (optional) each contain one MR number associated with the delta; the @c lines contain comments associated with the delta. The @e line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @u and @U. An empty list allows anyone to make a delta. Any line starting with a ! prohibits the succeeding group or user from making deltas.

Flags

Keywords used internally. See [admin\(1\)](#) for more information on their use. Each flag line takes the form:

@f <flag> <optional text>

The following flags are defined:

@f t <type of program>

@f v <program name>

@f i <keyword string>

@f b

@f m

<module name>

@f f <floor>

@f c <ceiling>

@f d <default-sid>

@f n

@f j

@f l

<lock-releases>

@f q <user defined>

@f z <reserved for use in interfaces>

The **t** flag defines the replacement text for the %Y% keyword. The **v** flag controls prompting for MR numbers in addition to comments; if the optional argument is present it defines an MR number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message causes a fatal error (the file will cannot be retrieved with "get", or the delta will not be made). When the **b** flag is present the -b keyletter may be used on the get command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the %M% keyword. The **f** flag defines the floor release; the release below which no deltas may be added. The **c** flag defines the ceiling release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a get command. The **n** flag causes delta to insert a null delta (a delta that applies no changes) in those releases that are skipped when a delta is made in a new release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes get to allow concurrent edits of the same base SID. The **l** flag defines a list of releases that are locked against editing. The **q** flag defines the replacement for the %Q% identification keyword. The **z** flag is used in specialized interface programs.

Comments

Arbitrary text is surrounded by the bracketing lines @t and @T. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines do not begin with the control character, control lines do. There are three kinds of control lines: insert, delete, and end, represented respectively by:

```
@I DDDDD@D DDDDD@E DDDDD
```

The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).

NAME

utmp, wtmp - entry formats

SYNOPSIS

```
#include <utmp.h>
#include <wtmp.h>
```

DESCRIPTION

These files, which hold user and accounting information for such commands as who, write, and login, have the following structure, defined in utmp.h:

```
#define UTMP_FILE "/var/adm/utmp"
#define WTMP_FILE "/var/adm/wtmp"
#define ut_name ut_user
struct utmp {
    char    ut_user[8];        /* user login name */
    char    ut_id[4];          /* /sbin/inittab id (created by */
                                /* process that puts entries in utmp) */
    char    ut_line[12];       /* device name (console, lnxx) */
    short   ut_pid;            /* process id */
    short   ut_type;           /* type of entry */
    struct  exit_status {
        short e_termination; /* process termination status */
        short e_exit;         /* error exit status */
    } ut_exit;                 /* exit status of a process
                                marked as DEAD_PROCESS */
    time_t  ut_time;           /* time entry was made */
};
/* Definitions for ut_type */
#define EMPTY          0
#define RUN_LVL         1
#define BOOT_TIME      2
#define OLD_TIME        3
#define NEW_TIME        4
#define INIT_PROCESS    5 /* process spawned by "init" */
#define LOGIN_PROCESS   6 /* a "getty" process waiting for login */
#define USER_PROCESS    7 /* a user process */
#define DEAD_PROCESS    8
#define ACCOUNTING      9
#define UTMAXTYPE       ACCOUNTING /* max legal value of ut_type */
/* Below are special strings or formats used in the "ut_line" */
/* field when accounting for something other than a process. */
/* No string for the ut_line field can be more than 11 chars + */
/* a null character in length. */
#define!RUNLVL_MSG!"run level %c"
#define!BOOT_MSG !"system boot"
#define!OTIME_MSG !"old time"
#define!NTIME_MSG !"new time"
```

The following functions are declared:

```
void endutent(void)
struct utmp *getutent(void)
struct utmp *getutid(const struct utmp*);
struct utmp *getutline(const struct utmp*);
struct utmp *pututline(const struct utmp*);
```

```
void setutent(void);  
int utmpname(const char *);
```

FILES

```
/var/adm/utmp  
/var/adm/wtmp
```

SEE ALSO

```
login(1), who(1), write(1), getut(3C).
```

NAME

utmpx, wtmpx - entry formats

SYNOPSIS

```
#include <utmpx.h>
#include <wtmpx.h>
```

DESCRIPTION

utmpx is an extended version of [utmp\(4\)](#).

These files, which hold user and accounting information for such commands as who, write, and login, have the following structure as defined by <utmpx.h>:

```
#define UTMPX_FILE    "/var/adm/utmpx"
#define WTMPX_FILE    "/var/adm/wtmpx"
#define ut_name       ut_user
#define ut_xtime       ut_tv.tv_sec
struct utmpx {
    char    ut_user[32];          /* user login name */
    char    ut_id[4];            /* inittab id */
    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;   /* process termination/exit status */
    struct timeval ut_tv;        /* time entry was made */
    long     ut_session;         /* session ID, used for windowing */
    long     pad[5];             /* reserved */
    short    ut_syslen;          /* significant length of ut_host */
    /* including terminating null */
    char     ut_host[257];        /* remote host name */
} ;

/* Definitions for ut_type */
#define EMPTY        0 /* No valid user accounting information */
#define RUN_LVL       1
#define BOOT_TIME     2 /* Time of system boot */
#define OLD_TIME      3 /* Time when system clock changed */
#define NEW_TIME      4 /* Time after system clock changed */
#define INIT_PROCESS  5 /* Process spawned by "init" */
#define LOGIN_PROCESS 6 /* A "getty" process waiting for login */
#define USER_PROCESS  7 /* A user process */
#define DEAD_PROCESS  8 /* A session leader who has exited */
#define ACCOUNTING     9
#define UTMAXTYPE     ACCOUNTING /* Largest legal value of ut_type */
/* Below are special strings or formats used in the "ut_line" */
/* field when accounting for something other than a process. */
/* No string for the ut_line field can be more than 11 chars + */
/* a null character in length. */
#define RUNLVL_MSG     "run-level %c"
#define BOOT_MSG       "system boot"
#define OTIME_MSG      "old time"
#define NTIME_MSG      "new time"
#define MOD_WIN        10
```

The following functions are declared:

```
void endutxent(void);
struct utmpx *getutxent(void);
```

```

struct utmpx *getutxid(const struct utmpx*);
struct utmpx *getutxline(const struct utmpx*);
struct utmpx *pututxline(const struct utmpx*);
void setutxent(void);
int utmpxname(const char*);
struct utmpx *makeutx(const struct utmpx*);
struct utmpx *modutx(const struct utmpx*);
void getutmp(const struct utmpx*, struct utmp*);
void getutmpx(const struct utmp*, struct utmpx*);
void updwtmp(const char*, struct utmp*);
void updwtmpx(const char*, struct utmpx*);

```

NOTES

The `timeval` structure is defined as described in `<sys/time.h>`. Inclusion of the `<utmpx.h>` header may also make visible all symbols from `<sys/time.h>`.

The `pid_t` type is defined through `typedef` as described in `<sys/types.h>`.

FILES

```

/var/adm/utmpx
/var/adm/wtmpx

```

SEE ALSO

`login(1)`, `who(1)`, `write(1)`, `getutx(3C)`, `time(5)`, `types(5)`.

Miscellaneous facilities and include files(5)

Introduction

NAME

`intro_prm5` - introduction to header files

DESCRIPTION

This chapter describes a number of header files whose contents may be of interest to C programmers.

Header files are also often referred to as include files.

Header files contain the definition of symbolic constants, common structures, preprocessor macros and predefined types. Each function description specifies the header that an application must include in order to use that function. If more than one header needs to be included, this must also be specified. The header files must be present on the system when the C source code is compiled. They need not be present when the programs are executed.

The header files generally reside in the directory `/usr/include` or `/usr/include/sys`. The directives `#include <filename.h>` or `#include <sys/filename.h>` should be used to link the header files into C programs.

The following header files are new in XPG4:

```

fnmatch.h
glob.h
iconv.h
monetary.h
regex.h
wchar.h
wordexp.h

```

NOTES

The descriptions of the header files comply with X/Open specifications. Proprietary enhancements are not always included.

When you invoke `man`, you specify the names of the header files *without* the `.h` ending, e.g. `man unistd` for the `<unistd.h>` header file. Likewise, the component `sys` can be omitted if it is part of the path, e.g. `man types` for the `<sys/types.h>` header file.

Caution: For the `time.h` header file, you have to invoke either `man time` for `<time.h>` or `man sys_time` for `<sys/time.h>`.

ascii(5) - nl_types(5)

 Programmer's Reference Manual
ascii(5)**NAME**

`ascii` - ISO 646 character set

DESCRIPTION

The following table contains the 7-bit ASCII ISO 646 character set (corresponds to the international version). The 8-bit ASCII ISO 8859-1 character set is contained in `meta-ascii(5)`.

Hex.	Oct.	Dec.	Graphic	Description	CTRL-
00	000	0	NUL	Null, no operation	@
01	001	1	SOH	Start of Heading	A
02	002	2	STX	Start of Text	B
03	003	3	ETX	End of Text	C
04	004	4	EOT	End of Transmission, <END0> key	D
05	005	5	ENQ	Enquiry	E
06	006	6	ACK	Acknowledge	F
07	007	7	BEL	Bell	G
08	010	8	BS	Backspace	H
09	011	9	HT	Horizontal Tabulation	I
0A	012	10	LF	Line Feed	J
0B	013	11	VT	Vertical Tabulation	K
0C	014	12	FF	Form Feed	L
0D	015	13	CR	Carriage Return	M
0E	016	14	SO	Shift Out	N
0F	017	15	SI	Shift In	O
10	020	16	DLE	Data Link Escape	P
11	021	17	DC1	Device Control 1 (= X-ON0)	Q
12	022	18	DC2	Device Control 2	R
13	023	19	DC3	Device Control 3 (= X-OFF0)	S
14	024	20	DC4	Device Control 4	T

15	025	21	NAK	Negative Acknowledge	U
16	026	22	SYN	Synchronous Idle	V
17	027	23	ETB	End of Transmission Block	W
18	030	24	CAN	Cancel	X
19	031	25	EM	End of Medium, quit (signal3)	Y
1A	032	26	SUB	Substitute Character	Z
1B	033	27	ESC	Escape	[
1C	034	28	FS	File Separator	\
1D	035	29	GS	Group Separator	]

Hex.	Oct.	Dec.	Graphic	Description	CTRL-
1E	036	30	RS	Record Separator	^
1F	037	31	US	Unit Separator	_
20	040	32	SP	SPACE0	
21	041	33	!	Exclamation point	
22	042	34	"	Quotation mark	
23	043	35	#	Number sign (hash mark)	
24	044	36	\$	Dollar sign	
25	045	37	%	Percent sign	
26	046	38	&	Ampersand	
27	047	39	'	Apostrophe	
28	050	40	(	Opening parenthesis	
29	051	41	)	Closing parenthesis	
2A	052	42	*	Asterisk	
2B	053	43	+	Plus	
2C	054	44	,	Comma	
2D	055	45	-	Hyphen (minus)	
2E	056	46	.	Period (point)	
2F	057	47	/	Slash	
30	060	48	0		
31	061	49	1		
32	062	50	2		
33	063	51	3		
34	064	52	4		
35	065	53	5		
36	066	54	6		
37	067	55	7		
38	070	56	8		

39	071	57	9		
3A	072	58	:	Colon	
3B	073	59	;	Semicolon	
3C	074	60	<	Less than sign	
3D	075	61	=	Equal sign	
3E	076	62	>	Greater than sign	
3F	077	63	?	Question mark	
40	100	64	@	Commercial at	

Hex.	Oct.	Dec.	Graphic	Description	CTRL-
41	101	65	A		
42	102	66	B		
43	103	67	C		
44	104	68	D		
45	105	69	E		
46	106	70	F		
47	107	71	G		
48	110	72	H		
49	111	73	I		
4A	112	74	J		
4B	113	75	K		
4C	114	76	L		
4D	115	77	M		
4E	116	78	N		
4F	117	79	O		
50	120	80	P		
51	121	81	Q		
52	122	82	R		
53	123	83	S		
54	124	84	T		
55	125	85	U		
56	126	86	V		
57	127	87	W		
58	130	88	X		
59	131	89	Y		
5A	132	90	Z		
5B	133	91	[	Opening square bracket	
5C	134	92	\	Backslash	

5D	135	93	]	Closing square bracket	
5E	136	94	^	Caret (circumflex)	
5F	137	95	_	Underscore (low line)	
60	140	96	`	Gravis	
61	141	97	a		
62	142	98	b		
63	143	99	c		
64	144	100	d		

Hex.	Oct.	Dec.	Graphic	Description	CTRL-
65	145	101	e		
66	146	102	f		
67	147	103	g		
68	150	104	h		
69	151	105	i		
6A	152	106	j		
6B	153	107	k		
6C	154	108	l		
6D	155	109	m		
6E	156	110	n		
6F	157	111	o		
70	160	112	p		
71	161	113	q		
72	162	114	r		
73	163	115	s		
74	164	116	t		
75	165	117	u		
76	166	118	v		
77	167	119	w		
78	170	120	x		
79	171	121	y		
7A	172	122	z		
7B	173	123	{	Opening brace (curly bracket)	
7C	174	124		Vertical line	
7D	175	125	}	Closing brace (curly bracket)	
7E	176	126	~	Tilde	
7F	177	127	DEL	Delete, rub out (signal2)	

SEE ALSO

`meta-ascii(5)`.

NAME

`assert` - verify program assertion

SYNOPSIS

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

DESCRIPTION

The `<assert.h>` header defines the `assert()` macro. It refers to the macro `NDEBUG` which is not defined in the header. If `NDEBUG` is defined as a macro name before the inclusion of this header, the `assert()` macro is defined simply as:

```
#define assert(ignore) ((void) 0)
```

otherwise the macro behaves as described in [assert\(3X\)](#).

The `assert()` macro is implemented as a macro, not as a function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

SEE ALSO

[assert\(3X\)](#).

NAME

cpio - cpio archive values

SYNOPSIS

```
#include <cpio.h>
```

DESCRIPTION

Values needed by the *c_mode* field of the *cpio* archive format are described by:

Name	Description	Value (octal)
C_IRUSR	read by owner	0000400
C_IWUSR	write by owner	0000200
C_IXUSR	execute by owner	0000100
C_IRGRP	read by group	0000040
C_IWGRP	write by group	0000020
C_IXGRP	execute by group	0000010
C_IROTH	read by others	0000004
C_IWOTH	write by others	0000002
C_IXOTH	execute by others	0000001
C_ISUID	set user ID	0004000
C_ISGID	set group ID	0002000
C_ISVTX	on directories, restricted deletion flag	0001000
C_ISDIR	directory	0040000
C_ISFIFO	FIFO	0010000
C_ISREG	regular file	0100000
C_ISBLK	block special	0060000
C_ISCHR	character special	0020000
C_ISCTG	reserved	0110000
C_ISLNK	symbolic link	0120000
C_ISSOCK	socket	0140000

The header defines the symbolic constant:
MAGIC "070707"

SEE ALSO

[cpio\(1\)](#).

NAME

ctype - character types

SYNOPSIS

```
#include <ctype.h>
```

DESCRIPTION

The <ctype.h> header declares the following as functions and may also define them as macros:

```
int  isalnum(int c);
int  isalpha(int c);
int  isascii(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);
int  toascii(int c);
int  tolower(int c);
int  toupper(int c);
```

The following are defined as macros:

```
int  _toupper(int c);
int  _tolower(int c);
```

SEE ALSO

[conv\(3C\)](#), [ctype\(3C\)](#), [mbchar\(3C\)](#), [mbstring\(3C\)](#), [setlocale\(3C\)](#), [locale\(5\)](#).

NAME

dlfcn - dynamic linking

SYNOPSIS

```
#include <dlfcn.h>
```

DESCRIPTION

The `<dlfcn.h>` header defines at least the following macros for use in the construction of a `dlopen()` mode argument:

`RTLD_LAZY`

Relocations are performed at an implementation-dependent time.

`RTLD_NOW`

Relocations are performed when the object is loaded.

`RTLD_GLOBAL`

All symbols are available for relocation processing of other modules.

`RTLD_LOCAL`

All symbols are not made available for relocation processing by other modules.

The header `<dlfcn.h>` declares the following functions which may also be defined as macros.

```
void *dlopen(const char *, int);
void *dlsym(void *, const char *);
int dlclose(void *);
char *dlerror(void);
```

SEE ALSO

[dlclose\(3C\)](#), [dlerror\(3C\)](#), [dlopen\(3C\)](#), [dlsym\(3C\)](#), [dladdr\(3X\)](#).

NAME

`environ` - user environment

DESCRIPTION

When a process begins execution, `exec` routines make available an array of strings called the "environment" [see `exec(2)`]. By convention, these strings have the form `variable=value`, for example, `PATH=/sbin:/usr/sbin`. These environmental variables provide a way to make information about a program's environment available to applications. The following environmental variables must be set in the target run-time environment.

HOME The name of the user's login directory, set by `login(1)` from the password file [see `passwd(4)`].

LANG The string used to specify information about the locale, that allows users to work with different national conventions. The `setlocale(3C)` function uses the `LANG` environment variable when it is called with `" "` as the `locale` argument. `LANG` is used as the default locale if the corresponding environment variable for a particular category is unset.

For example, when `setlocale` is invoked as

```
setlocale(LC_CTYPE, " ")
```

the `LC_CTYPE` environment variable is checked first, to see if it is set and non-null. If `LC_CTYPE` is not set or null, then `setlocale` will check the `LANG` environment variable to see if it is set and non-null. If both `LANG` and `LC_CTYPE` are unset or null, the default C locale will be used to set the `LC_CTYPE` category. Most commands will invoke

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

prior to any other processing. This allows the command to be used with different national conventions by setting the appropriate environment variables.

The following environment variables are supported to correspond with each category of `setlocale(3C)`:

LC_COLLATE

This category specifies the collation sequence being used. The information corresponding to this category is stored in a database created by the `colltbl(1M)` command. This environment variable affects `strcoll(3C)` and `strxfrm(3C)`.

LC_CTYPE

This category specifies character classification, character conversion, and widths of multibyte characters. The database created by the `chrtbl(1M)` command. The default for C corresponds to the 7-bit character set. This environment variable is used by `ctype(3C)`, `mbchar(3C)`, and many commands; for example: `cat(1)`, `ed(1)`, `ls(1)`, and `vi(1)`.

LC_MESSAGES

This category specifies the language of the message database being used. For example, an application may have one message database with French messages, and another database with German messages. Message databases are created by the `mkmsgs(1)` command. This environment variable is used by `exstr(1)`, `gettext(1)`, `gettext(3C)`, and `srchtxt(1)`.

LC_MONETARY

Used for a particular locale. The information corresponding to this category is stored in a database created by the `montbl(1M)` command. This environment variable is used by `localeconv(3C)`.

LC_NUMERIC

This category specifies the decimal and thousands delimiters. The information corresponding to this category is stored in a database created by the `chrtbl(1M)` command. The default for C corresponds to `.` as the decimal delimiter and no thousands delimiter. This environment variable

is used by `localeconv(3C)`, `printf(3S)`, and `strtod(3C)`.

LC_TIME

This category specifies date and time formats. The information corresponding to this category is stored in a database specified in `LC_TIME(4)`. The default for C corresponds to U.S. date and time formats. This environment variable is used by many commands and functions; for example: `at(1)`, `calendar(1)`, `date(1)`, `strftime(3C)`, and `getdate(3C)`.

MSGVERB

Controls which standard format message components `fmtmsg` selects when messages are displayed to `stderr` [see `fmtmsg(1)` and `fmtmsg(3C)`].

SEV_LEVEL

Defines severity levels and associates and prints strings with them in standard format error messages [see `addseverity(3C)`, `fmtmsg(1)`, and `fmtmsg(3C)`].

NETPATH

A colon-separated list of network identifiers. A network identifier is a character string used by the Network Selection component of the system to provide application-specific default network search paths. A network identifier must consist of non-NULL characters and must have a length of at least 1. No maximum length is specified. Network identifiers are normally chosen by the superuser. A network identifier is also the first field in any `/etc/netconfig` file entry. `NETPATH` thus provides a link into the `/etc/netconfig` file and the information about a network contained in that network's entry. `/etc/netconfig` is maintained by the superuser. The library routines described in `getnetpath(3N)` access the `NETPATH` environment variable.

NLSPATH

Contains a sequence of templates which `catopen(3C)` uses when attempting to locate message catalogs. Each template consists of an optional prefix, one or more substitution fields, a filename and an optional suffix.

```
NLSPATH="/system/nlslib/%N.cat"
```

defines that `catopen` should look for all message catalogs in the directory `/system/nlslib`, where the catalog name should be constructed from the `name` parameter passed to `catopen()`, `%N`, plus the suffix `.cat`.

Substitution fields consist of a `%` symbol, followed by a single-letter keyword. The following keywords are currently defined:

```
%N    The value of the name parameter passed to catopen().
%L    The value of LANG.
%l    The language element from LANG.
%t    The territory element from LANG.
%c    The codeset element from LANG.
%%    A single % character.
```

An empty string is substituted if the specified value is not currently defined. The separators `_` and `.` are not included in `%t` and `%c` substitutions.

Templates defined in `NLSPATH` are separated by colons. A leading colon or two adjacent colons `::` is equivalent to specifying `%N`.

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

indicates to `catopen` that it should look for the requested message catalog in `name`, `name.cat` and `/nlslib/$LANG/name.cat`.

PATH The sequence of directory prefixes that `sh(1)`, `time(1)`, `nice(1)`, `nohup(1)`, etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons. `login(1)` sets `PATH=/usr/bin` [see `sh(1)`].

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as `vi(1)`, which may exploit special capabilities of that terminal.

TZ Time zone information. The contents of this environment variable are used by the functions `ctime(3C)`, `localtime` [see `ctime(3C)`], `strftime(3C)` and `mktime(3C)` to override the default timezone. If the first character of **TZ** is a colon, the behavior is implementation defined, otherwise **TZ** has the form:

std offset [dst [offset],[start[/time],end[/time]]]

std and *dst* contain three or more bytes that are the designation for the standard (*std*) and daylight savings time (*dst*) timezones. Only *std* is required. If *dst* is missing, then daylight savings time does not apply in this environment. Upper- and lower-case letters are allowed. Any characters except a leading colon, digits, a comma, a minus sign or a plus sign are allowed.

offset indicates the value one must add to the local time to arrive at Coordinated Universal Time.

The offset has the form:

hh [:mm [:ss]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst*, daylight savings time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between 0 and 24, and the minutes (and seconds) if present between 0 and 59. Out of range values may cause unpredictable behavior. If preceded by a `-`, the timezone is east of the Prime Meridian; otherwise it is west (which may be indicated by an optional preceding `+` sign).

start/time, *end/time* indicates when to change to and back from daylight savings time, where *start/time* describes when the change from standard time to daylight savings time occurs, and *end/time* describes when the change back happens. Each *time* field specifies when the change is made.

The formats of *start* and *end* are one of the following:

Jn The Julian day *n* ($1 \leq n \leq 365$). Leap days are not counted. That is, in all years, February 28 is day 59 and March 1 is day 60. It is impossible to refer to the occasional February 29.

n The zero-based Julian day ($0 \leq n \leq 365$); leap days are counted, and it is possible to refer to February 29.

Mm.n.d

The *d*th day, ($0 \leq d \leq 6$) of week *n* of month *m* of the year ($1 \leq n \leq 5$, $1 \leq m \leq 12$), where week 5 means "the last day number *d* in month *m*", and may occur in either the fourth or the fifth week. Week 1 is the first week in which the *d*th day occurs. Day zero is Sunday.

Implementation specific defaults are assumed for *start* and *end* if these optional fields are not given.

The *time* has the same format as *offset* except that no leading sign (`-` or `+`) is allowed. The default if *time* is not given is 02:00:00.

Further names may be placed in the environment by the `export` command and *name=value* arguments in `sh(1)`, or by `exec(2)`. It is unwise to conflict with certain shell variables that are frequently exported by `.profile` files: `MAIL`, `PS1`, `PS2`, `IFS` [see `profile(4)`].

SEE ALSO

`cat(1)`, `date(1)`, `ed(1)`, `fmtmsg(1)`, `login(1)`, `ls(1)`, `mkmsgs(1)`, `nice(1)`, `nohup(1)`, `sh(1)`, `sort(1)`, `time(1)`, `vi(1)`, `chrtbl(1M)`, `colltbl(1M)`, `monttbl(1M)`, `exec(2)`, `addseverity(3C)`, `catopen(3C)`, `ctime(3C)`, `ctype(3C)`, `fmtmsg(3C)`, `getdate(3C)`, `gettext(3C)`, `localeconv(3C)`, `mbchar(3C)`, `mktime(3C)`, `strcoll(3C)`, `strftime(3C)`, `strtod(3C)`, `strxfrm(3C)`, `getnetpath(3N)`, `printf(3S)`, `LC_TIME(4)`, `netconfig(4)`, `passwd(4)`, `profile(4)`, `timezone(4)`.

Manual "Programmers Guide: Internationalization – Localization".

NAME

errno - system error numbers

SYNOPSIS

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

DESCRIPTION

The `<errno.h>` header provides a declaration for `errno` and gives non-zero values for the following symbolic constants. Their values are unique except as noted below:

E2BIG

Argument list too long.

EACCES

Permission denied.

EADDRINUSE

Address in use.

EADDRNOTAVAIL

Address not available.

EADV Advertise error.

EAFNOSUPPORT

Address family not supported.

EAGAIN

Resource unavailable, try again (may be the same value as `EWOULDBLOCK`).

EALREADY

Connection already in progress.

EBADE

Bad exchange descriptor.

EBADF

Bad file descriptor.

EBADFD

File descriptor in bad state.

EBADMSG

Bad message.

EBADR

Bad request descriptor.

EBADRQC

Bad request code.

EBADSLT

Invalid slot.

EBFONT

Bad font file format.

EBUSY

Device or resource busy.

ECANCELED

Operation canceled.

ECHILD

No child processes.

ECHRNG

	Channel number out of range.
ECOMM	Communication error on send
ECONNABORTED	Connection aborted.
ECONNREFUSED	Connection refused.
ECONNRESET	Connection reset.
EDEADLK	Resource deadlock would occur.
EDEADLOCK	File locking deadlock.
EDESTADDRREQ	Destination address required.
EDISJOINT	Lnode hierarchy is disjointed.
EDOM	Mathematics argument out of domain of function.
EDQUOT	Reserved.
EEXIST	File exists.
EFAULT	Bad address.
EFBIG	File too large.
EGROUPLOOP	Loop in lnode hierarchy.
EHANDLER	Signal handler exists.
EHOSTDOWN	Host is down.
EHOSTUNREACH	Host is unreachable.
EIDRM	Identifier removed.
EILSEQ	Illegal byte sequence.
EINIT	Reserved.
EINPROGRESS	Connection in progress.
EINTR	Interrupted function.
EINVAL	Invalid argument.
EIO	I/O error.
EIORESID	Block not fully transferred.

EISCONN
Socket is connected.

EISDIR
Is a directory.

EISNAM
Is a name file.

EL2HLT
Level 2 halted.

EL2NSYNC
Level 2 not synchronized.

EL3HLT
Level 3 halted.

EL3RST
Level 3 reset.

ELIBACC
Can not access a needed shared library.

ELIBBAD
Accessing a corrupted shared library.

ELIBEXEC
Can not exec a shared library directly.

ELIBMAX
Attempting to link in more shared libraries than system limit.

ELIBSCN
.lib section in a.out corrupted.

ELNRNG
Link number out of range.

ELOGINLIM
Login limit reached.

ELOOP
Too many levels of symbolic links.

EMFILE
Too many open files.

EMLINK
Too many links.

EMSGSIZE
Message too large.

EMULTIHOP
Multihop attempted.

ENAMETOOLONG
Filename too long.

ENAVAIL
Not available.

ENETDOWN
Network is down.

ENETRESET
Network dropped connection because of reset.

ENETUNREACH
Network unreachable.

ENFILE
Too many files open in system.

ENOANO	Anode table overflow.
ENOATTACH	Attach to Inode not allowed.
ENOBUFFS	No buffer space available.
ENOCSSI	No CSI structure available.
ENODATA	No message is available on the STREAM head read queue.
ENODEV	No such device.
ENOENT	No such file or directory.
ENOEXEC	Executable file format error.
ENOLCK	No locks available.
ENOLIMFILE	Lnode database not open.
ENOLINK	Link has been severed.
ENOLOGIN	No logins allowed for user.
ENOMEM	Not enough space.
ENOMSG	No message of the desired type.
ENONET	Machine is not on the network.
ENOPKG	Package not installed.
ENOPROTOPT	Protocol not available.
ENOSPC	No space left on device.
ENOSR	No STREAM resources.
ENOSTR	Not a STREAM.
ENOSYS	Function not supported.
ENOTBLK	Block device required.
ENOTCONN	The socket is not connected.
ENOTDIR	Not a directory.
ENOTEMPTY	Directory not empty.

ENOTNAM	Not a name file.
ENOTSOCK	Not a socket.
ENOTTY	Not a typewriter. Inappropriate I/O control operation.
ENOTUNIQ	Name not unique on network.
ENXIO	No such device or address.
EOPNOTSUPP	Operation not supported on socket.
EOVERFLOW	Value too large to be stored in data type.
EPERM	Operation not permitted.
EPFNOSUPPORT	Protocol family not supported.
EPIPE	Broken pipe.
EPROCLIM	Process limit reached.
EPROTO	Protocol error.
EPROTONOSUPPORT	Protocol not supported.
EPROTOTYPE	Socket type not supported.
ERANGE	Result too large.
EREMCHG	Remote address changed.
EREMDEV	Error 142.
EREMOTE	Object is remote.
EREMOTEIO	Remote I/O error.
ERESTART	Restartable system call.
EROFS	Read-only file system.
ESHUTDOWN	Cannot send after socket shutdown.
ESOCKTNOSUPPORT	Socket type not supported.
ESPIPE	Invalid seek.
ESRCH	No such process.

ESRMNT	Surmount error.
ESTALE	Stale NFS file handle.
ESTRPIPE	If pipe/FIFO, don't sleep in stream head.
ETIME	Stream <code>ioctl()</code> timeout.
ETIMEDOUT	Connection timed out.
ETOOMANYREFS	Too many references: cannot splice.
ETXTBSY	Text file busy.
EUCLEAN	Structure needs cleaning.
EUNATCH	Protocol driver not attached.
EUSERS	Too many users (for UFS).
EWouldBlock	Operation would block (may be the same value as <code>EAGAIN</code>).
EXDEV	Cross-device link.
EXFULL	Message tables full.

SEE ALSO

Section "ERROR NUMBERS" in [intro_prm2\(2\)](#).

NAME

fcntl - file control options

SYNOPSIS

```
#include <fcntl.h>
```

DESCRIPTION

The `<fcntl.h>` header file defines the following requests and arguments for use by the functions `fcntl`.

Unique values for commands used by `fcntl()`:

`F_DUPFD`

Duplicate file descriptor

`F_GETFD`

Get file descriptor flags

`F_SETFD`

Set file descriptor flags

`F_GETFL`

Get file status flags

`F_SETFL`

Set file status flags

`F_GETLK`

Get record locking information

`F_SETLK`

Set record locking information

`F_SETLKW`

Set record locking information; wait if blocked

`F_GETOWN`

Get process or process group ID to receive SIGURG signals

`F_SETOWN`

Set process or process group ID to receive SIGURG signals

File descriptor flags used for `fcntl()`:

`FD_CLOEXEC`

Close the file descriptor upon execution of an exec function [see [exec\(2\)](#)]

Unique values for `l_type` used for record locking with `fcntl()`:

`F_RDLCK`

Shared or read lock

`F_UNLCK`

Unlock

`F_WRLCK`

Exclusive or write lock

The values used for `l_whence`, `SEEK_SET`, `SEEK_CUR` and `SEEK_END` are defined as described in `<unistd.h>`.

Sets of values for *oflag* used by `open()` (bitwise distinct):

`O_CREAT`

Create file if it does not exist

`O_EXCL`

Exclusive use flag

`O_NOCTTY`
Do not assign controlling tty

`O_TRUNC`
Truncate flag

File status flags used for `open()` and `fcntl()`:

`O_APPEND`
Set append mode

`O_NDELAY`
Non-blocking mode

`O_NONBLOCK`
Non-blocking mode (POSIX)

`O_SYNC`
Synchronous writes

Mask for use with file access modes:

`O_ACCMODE`
Mask for file access modes

File access modes used for `open()` and `fcntl()`:

`O_RDONLY`
Open for reading only

`O_RDWR`
Open for reading and writing

`O_WRONLY`
Open for writing only

The symbolic names for file modes for use as values of `mode_t` are defined as described in `<sys/types.h>`.

Asynchronous I/O-specific file marker:

`FRAIOSIG`
Produces a signal when the job has ended (defined in `<file.h>`).

This marker defines a specific mode which is of importance in conjunction with asynchronous I/O. The marker is file-specific.

If the marker is set for a file, then the signal `SIGEMT` is sent to the issuing process for all asynchronous read and write jobs, when the asynchronous I/O job has ended.

This mode is generally not set [see also [aio\(7\)](#)].

The `flock` structure describes a file lock. It includes the following members:

```
short  l_type;    /* Type of lock */
short  l_whence;  /* Flag for starting offset */
off_t  l_start;   /* Relative offset in bytes */
off_t  l_len;     /* Size; if 0 then until EOF */
long   l_sysid;   /* Returned with F_GETLK */
pid_t  l_pid;     /* Returned with F_GETLK */
```

The `mode_t`, `off_t` and `pid_t` types are defined as described in `<sys/types.h>`.

The following are declared as functions and may also be defined as macros:

```
int  creat(const char *path, mode_t mode);
int  fcntl(int fildes, int cmd, ...);
int  open(const char *path, int oflag, ...);
```

Inclusion of the `<fcntl.h>` header may also make visible all symbols from `<sys/stat.h>` and `<unistd.h>`.

The `flock64` structure is defined in the same way as the `flock` structure with the exception of the following members:

`off64_t l_start`

Relative offset in bytes.

`off64_t l_len`

Size.

Additional values for *cmd* used by `fcntl()`:

`F_GETLK64`

Get record locking information using `struct flock64`.

`F_SETLK64`

Establish a record lock using `struct flock64`.

`F_SETLKW64`

Establish a record lock, blocking, using `struct flock64`.

An additional file status flag, used by `open()` and `fcntl()`, is defined:

`O_LARGEFILE`

The offset maximum in the open file description is the largest value that can be represented correctly in an object of type `off64_t`.

The following are declared as functions and may also be defined as macros:

```
int creat64(const char *path, mode_t mode);
```

```
int open64(const char *path, int oflag, ...);
```

SEE ALSO

[creat\(2\)](#), [exec\(2\)](#), [fcntl\(2\)](#), [open\(2\)](#), [unistd\(4\)](#), [lfs\(5\)](#), [stat\(5\)](#), [types\(5\)](#), [aio\(7\)](#).

NAME

float - floating types

SYNOPSIS

```
#include <float.h>
```

DESCRIPTION

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic. The following parameters are used to define the model for each floating-point type:

- s sign (1)
- b base or radix of exponent representation (an integer > 1)
- e exponent (an integer between a minimum $e_{\min}$ and a maximum $e_{\max}$)
- p precision (the number of base-*b* digits in the significant)
- f_k non-negative integers less than *b* (the significant digits)

A normalized floating-point number x ($f_1 > 0$ if $x \neq 0$) is defined by the following model:

$$x = s \times b^e \times \sum_{k=1}^p f_k \times b^{-k}, \quad e_{\min} \leq e \leq e_{\max}$$

FLT_RADIX will be a constant expression suitable for use in the #if preprocessing directives. All except FLT_RADIX and FLT_ROUNDS have separate names for all three floating-point types. The floating-point model representation is provided for all macro names except FLT_ROUNDS.

The rounding mode for floating-point addition is characterized by the value of FLT_ROUNDS:

- 1 indeterminable
- 0 toward 0.0
- 1 to nearest
- 2 toward positive infinity
- 3 toward negative infinity

All other values for FLT_ROUNDS characterize implementation-dependent rounding behavior. The macro names given in the following list will be defined as expressions with values that are equal or greater in magnitude (absolute value) to those shown, with the same sign.

Name	Description	Value
FLT_RADIX	radix of exponent representation, <i>b</i>	2
FLT_MANT_DIG	number of base-FLT_RADIX digits in the floating-point significant, <i>p</i>	
DBL_MANT_DIG		
LDBL_MANT_DIG		
FLT_DIG	number of decimal digits, <i>q</i> , such that any floating-point number with <i>q</i> decimal digits can be rounded into a floating-point number	6
DBL_DIG		10

LBL_DIG	with p radix b digits and back again without change to the q decimal digits, $\square$	10
	$\left\lfloor (p-1) \times \log_{10} b \right\rfloor + \begin{cases} 1 & \text{if } b \text{ is a power of } 10 \\ 0 & \text{otherwise} \end{cases}$	
FLT_MIN_EXP DBL_MIN_EXPLDBL _MIN_EXP	minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number, $e_{\min}$	
FLT_MIN_10_EXP $\square$	minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\square$	-37 $\square$
DBL_MIN_10_EXP $\square$	$\left\lfloor \log_{10} b^{e_{\min}^{-1}} \right\rfloor$	-37
LDBL_MIN_10_EXP	$\square$	
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number, $e_{\max}$	
FLT_MAX_10_EXP $\square$	maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\square$	37 $\square$
DBL_MAX_10_EXP $\square$	$\left\lfloor \log_{10}((1 - b^{-p}) \times b^{e_{\max}}) \right\rfloor$	37
LDBL_MAX_10_EXP	$\square$	

The macro names given in the following list will be defined as expressions with values that will be equal to or greater than those shown.

FLT_MAX	maximum representable finite floating-point number, $\square$	1E+37
DBL_MAX	$(1 - b^{-p}) \times b^{e_{\max}}$	1E+37
LDBL_MAX	$\square$	1E+37

The macro names given in the following list will be defined as expressions with values that will be equal to or less than those shown.

FLT_EPSILON	the difference between 1.0 and the least value greater than 1.0 that is representable in the given floating-point type, $b^{(1-p)}$	1E-5
DBL_EPSILON	$\square$	1E-9
LDBL_EPSILON	$\square$	1E-9
FLT_MIN	minimum normalized positive floating-point number, $\square$	1E-37
DBL_MIN	$b^{(e_{\min}-1)}$	1E-37
LDBL_MIN	$\square$	1E-37

NAME

fmtmsg - message display structures

SYNOPSIS

```
#include <fmtmsg.h>
```

DESCRIPTION

The `<fmtmsg.h>` header defines the following macros, which expand to constant integral expressions:

MM_HARD	Source of the condition is hardware.
MM_SOFT	Source of the condition is software.
MM_FIRM	Source of the condition is firmware.
MM_APPL	Condition detected by application.
MM_UTIL	Condition detected by utility.
MM_OPSYS	Condition detected by operating system.
MM_RECOVER	Recoverable error.
MM_NRECOV	Non-recoverable error.
MM_HALT	Error causing application to halt.
MM_ERROR	Application has encountered a non-fatal fault.
MM_WARNING	Application has detected unusual non-error condition.
MM_INFO	Informative message.
MM_NOSEV	No severity level provided for the message.
MM_PRINT	Display message on standard error.
MM_CONSOLE	Display message on system console.

The table below indicates the null values and identifiers for `fmtmsg()` arguments. The `<fmtmsg.h>` header defines the macros in the **Identifier** column, which expand to constant expressions that expand to expressions of the type indicated in the **Type** column:

Argument	Type	Null-Value	Identifier
label	char*	(char*) 0	MM_NULLLBL
severity	int	0	MM_NULLSEV
class	long int	0L	MM_NULLMC
text	char*	(char*) 0	MM_NULLTXT
action	char*	(char*) 0	MM_NULLACT
tag	char*	(char*) 0	MM_NULLTAG

The `<fmtmsg.h>` header also defines the following macros for use as return values for `fmtmsg(3C)`:

MM_OK	The function succeeded.
MM_NOTOK	The function failed completely.
MM_NOMSG	

The function was unable to generate a message on standard error, but otherwise succeeded.

`MM_NOCON`

The function was unable to generate a console message, but otherwise succeeded.

The following is declared as a function and may also be defined as a macro:

```
int fmtmsg(long classification, const char *label, int severity,  
           const char *text, const char *action, const char *tag);
```

SEE ALSO

`fmtmsg(3C)`.

NAME

fnmatch - filename-matching types

SYNOPSIS

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

DESCRIPTION

The `fnmatch.h` header defines the flags and return values used by the `fnmatch(3C)` function. The following constants are defined:

`FNM_BADPAT`

An unknown pattern or an unknown flag was specified.

`FNM_NOMATCH`

The string does not match the specified *pattern*.

`FNM_PATHNAME`

Slash in *string* only matches slash in *pattern*.

`FNM_PERIOD`

Leading period in *string* must be exactly matched by period in *pattern*.

`FNM_NOESCAPE`

Disable backslash escaping.

`FNM_RANGEOK`

A range specification in the format `[m-n-r]` is interpreted as `[m-r]`.

`FNM_RANGE_ERR`

A range specification in the format `[m-n-r]` is detected as an error.

`FNM CLRANGE_ERR`

A range specification in the format `[ [=m=] -r` is detected as an error.

`FNM_NOSYS`

The implementation does not support this function.

The following is declared as a function and may also be declared as a macro:

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

SEE ALSO

`fnmatch(3C)`.

NAME

ftw - file tree traversal

SYNOPSIS

```
#include <ftw.h>
```

DESCRIPTION

The `<ftw.h>` header defines the `FTW` structure that includes at least the following members:

```
int  base
int  level
```

The `<ftw.h>` header defines macros for use as values of the third argument to the application-supplied function that is passed as the second argument to `ftw()` and `nftw()`:

```
FTW_F      File.
FTW_D      Directory.
FTW_DNR    Directory without read permission.
FTW_DP     Directory with subdirectories visited.
FTW_NS     Unknown type, stat() failed.
FTW_SL     Symbolic link.
FTW_SLN    Symbolic link that names a non-existent file.
```

The `<ftw.h>` header defines macros for use as values of the fourth argument to `nftw()`:

```
FTW_PHYS   Physical walk, does not follow symbolic links. Otherwise, nftw() will follow links but will not walk down
            any path that crosses itself.
FTW_MOUNT  The walk will not cross a mount point.
FTW_CHDIR  The walk will change to each directory before reading it.
FTW_DEPTH  All subdirectories will be visited before the directory itself.
FTW_NONFS  No NFS filesystems are traversed.
```

The following are declared as functions and may also be defined as macros:

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

```
int nftw(const char *path, int (*fn)
         (const char *, const struct stat *, int, struct FTW*),
         int depth, int flags);
```

The `<ftw.h>` header defines the `stat` structure and the symbolic names for `st_mode` and the file type test macros as described in `<sys/stat.h>`.

Inclusion of the `<ftw.h>` header may also make visible all symbols from `<sys/stat.h>`.

The following are declared as functions and may also be defined as macros:

```
int ftw64(const char *path,
          int (*fn)(const char *, const struct stat64 *, int), int ndirs);
int nftw64(const char *path,
            int (*fn)(const char *, const struct stat64 *, int, struct FTW *),
            int depth, int flags);
```

SEE ALSO

[ftw\(3C\)](#), [nftw\(3C\)](#), [lfs\(5\)](#), [stat\(5\)](#).

NAME

glob - pathname pattern-matching types

SYNOPSIS

```
#include <glob.h>
```

DESCRIPTION

The `glob.h` header defines the structures and symbolic constants used by the `glob(3C)` function. The structure type `glob_t` contains at least the following members:

```
size_t gl_pathc
```

Count of paths matched by *pattern*.

```
char **gl_pathv
```

Pointer to a list of matched pathnames.

```
size_t gl_offs
```

Slots to reserve at the beginning of `gl_pathv`.

The following constants are provided as values for the *flags* argument:

```
GLOB_APPEND
```

Append generated pathnames to those previously obtained.

```
GLOB_DOOFFS
```

Specify how many null pointers to add to the beginning of `pglob->gl_pathv`.

```
GLOB_ERR
```

Cause `glob()` to return on error.

```
GLOB_MARK
```

Each pathname that is a directory that matches *pattern* has a slash appended.

```
GLOB_NOCHECK
```

If *pattern* does not match any pathname, then return a list consisting of only *pattern*.

```
GLOB_NOESCAPE
```

Disable backslash escaping.

```
GLOB_NOSORT
```

Do not sort the pathnames returned.

The following constants are defined as error return values:

```
GLOB_ABORTED
```

The scan was stopped because `GLOB_ERR` was set or `(*errfunc)()` returned non-zero.

```
GLOB_BADPAT
```

An invalid pattern was specified.

```
GLOB_NOMATCH
```

The pattern does not match any existing pathname, and `GLOB_NOCHECK` was not set in *flags*.

```
GLOB_NOSPACE
```

An attempt to allocate memory failed.

```
GLOB_NOSYS
```

The implementation does not support this function.

The following are declared as functions and may also be declared as macros:

```
int glob(const char *pattern, int flags, int (*errfunc)
        (const char *epath, int errno), glob_t *pglob);
```

```
void globfree(glob_t *pglob);
```

The implementation may define additional macros or constants using names beginning with `GLOB_`.

SEE ALSO

`glob(3C)` .

NAME

grp - group structure

SYNOPSIS

```
#include <grp.h>
```

DESCRIPTION

The `<grp.h>` header declares the structure `group` which includes the following members:

```
char    *gr_name
```

The name of the group.

```
gid_t    gr_gid
```

Numerical group ID.

```
char    **gr_mem
```

Pointer to a null-terminated array of character pointers to member names.

The `gid_t` type is defined as described in `<sys/types.h>` [see [types\(5\)](#)].

The following are declared as functions and may also be defined as macros:

```
struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
int getgrgid_r(gid_t gid, struct group *grp, char *buffer,
               size_t bufsize, struct group **result);
int getgrnam_r(const char *name, struct group *grp,
               char *buffer, size_t bufsize, struct group **result);
struct group *getgrent(void);
void endgrent(void);
void setgrent(void);
```

SEE ALSO

[getgrent\(3C\)](#), [types\(5\)](#).

NAME

iconv - codeset conversion facility

SYNOPSIS

```
#include <iconv.h>
```

DESCRIPTION

The `iconv.h` header defines the following data type through `typedef`:

```
iconv_t
```

Identifies the conversion from one codeset to another.

The following are declared as functions and may also be declared as macros:

```
iconv_t iconv_open(const char *tocode, const char *fromcode);
```

```
size_t iconv(iconv_t cd, char **inbuf, size_t *inbytesleft,  
             char **outbuf, size_t *outbytesleft);
```

```
int iconv_close(iconv_t cd);
```

SEE ALSO

[iconv\(3C\)](#), [iconv_close\(3C\)](#), [iconv_open\(3C\)](#).

NAME

`inttypes` - fixed size integral types

SYNOPSIS

```
#include <inttypes.h>
```

DESCRIPTION

The `<inttypes.h>` header includes definitions of at least the following types:

`int8_t`

8-bit signed integral type.

`int16_t`

16-bit signed integral type.

`int32_t`

32-bit signed integral type.

`int64_t`

64-bit signed integral type.

`uint8_t`

8-bit unsigned integral type.

`uint16_t`

16-bit unsigned integral type.

`uint32_t`

32-bit unsigned integral type.

`uint64_t`

64-bit unsigned integral type.

`intptr_t`

Signed integral type large enough to hold any pointer.

`uintptr_t`

Unsigned integral type large enough to hold any pointer.

NAME

ipc - interprocess communication access structure

SYNOPSIS

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

DESCRIPTION

The `<sys/ipc.h>` header is used by three mechanisms for interprocess communication (IPC): messages, semaphores and shared memory. All use a common structure type, `ipc_perm` to pass information used in determining permission to perform an IPC operation.

The structure `ipc_perm` contains the following members:

```
uid_t    uid
    owner's user ID
gid_t    gid
    owner's group ID
uid_t    cuid
    creator's user ID
gid_t    cgid
    creator's group ID
mode_t    mode
    read/write permission
```

The `uid_t`, `gid_t`, `mode_t` and `key_t` types are defined as described in `<sys/types.h>`.

Definitions are given for the following constants:

Mode bits:

```
IPC_CREAT
    Create entry if key does not exist.
IPC_EXCL
    Fail if key exists.
IPC_NOWAIT
    Error if request must wait.
```

Keys:

```
IPC_PRIVATE
    Private key.
```

Control Commands:

```
IPC_RMID
    Remove identifier.
IPC_SET
    Set options.
IPC_STAT
    Get options.
```

The following is declared as a function and may also be defined as a macro:

```
key_t  ftok(const char *path, int id);
```

SEE ALSO

[ftok\(3C\)](#), [types\(5\)](#).

NAME

iso646 - alternative spellings

SYNOPSIS

```
#include <iso646.h>
```

DESCRIPTION

The `<iso646.h>` header defines the following eleven macros (on the left) that expand to the corresponding tokens (on the right):

<code>and</code>	<code>&&</code>
<code>and_eq</code>	<code>&=</code>
<code>bitand</code>	<code>&</code>
<code>bitor</code>	<code> </code>
<code>compl</code>	<code>~</code>
<code>not</code>	<code>!</code>
<code>not_eq</code>	<code>!=</code>
<code>or</code>	<code> </code>
<code>or_eq</code>	<code> =</code>
<code>xor</code>	<code>^</code>
<code>xor_eq</code>	<code>^=</code>

NAME

langinfo - language information constants

SYNOPSIS

```
#include <langinfo.h>
```

DESCRIPTION

This header file contains the constants used to identify data objects for the language environment [see [nl_langinfo\(3C\)](#)]. The type of the constants, `nl_item`, is defined as described in `nl_types.h`.

The entries under **Category** indicate in which `setlocale()` category each item is defined.

Constant	Category	Meaning
CODESET	LC_CTYPE	codeset name
D_T_FMT	LC_TIME	string for formatting date and time
D_FMT	LC_TIME	date format string
T_FMT	LC_TIME	time format string
T_FMT_AMP M	LC_TIME	a.m. or p.m. time format string
AM_STR	LC_TIME	Ante Meridian affix
PM_STR	LC_TIME	Post Meridian affix
DAY_1	LC_TIME	name of the first day of the week (e.g. Sunday)
DAY_2	LC_TIME	name of the second day of the week (e.g. Monday)
DAY_3	LC_TIME	name of the third day of the week (e.g. Tuesday)
DAY_4	LC_TIME	name of the fourth day of the week (e.g. Wednesday)
DAY_5	LC_TIME	name of the fifth day of the week (e.g. Thursday)
DAY_6	LC_TIME	name of the sixth day of the week (e.g. Friday)
DAY_7	LC_TIME	name of the seventh day of the week (e.g. Saturday)
ABDAY_1	LC_TIME	abbreviated name of the first day of the week
ABDAY_2	LC_TIME	abbreviated name of the second day of the week
ABDAY_3	LC_TIME	abbreviated name of the third day of the week
ABDAY_4	LC_TIME	abbreviated name of the fourth day of the week
ABDAY_5	LC_TIME	abbreviated name of the fifth day of the week
ABDAY_6	LC_TIME	abbreviated name of the sixth day of the week
ABDAY_7	LC_TIME	abbreviated name of the seventh day of the week
MON_1	LC_TIME	name of the first month of the year
MON_2	LC_TIME	name of the second month
MON_3	LC_TIME	name of the third month
MON_4	LC_TIME	name of the fourth month
MON_5	LC_TIME	name of the fifth month

Constant	Category	Meaning
MON_6	LC_TIME	name of the sixth month
MON_7	LC_TIME	name of the seventh month
MON_8	LC_TIME	name of the eighth month
MON_9	LC_TIME	name of the ninth month
MON_10	LC_TIME	name of the tenth month
MON_11	LC_TIME	name of the eleventh month
MON_12	LC_TIME	name of the twelfth month
ABMON_1	LC_TIME	abbreviated name of the first month
ABMON_2	LC_TIME	abbreviated name of the second month
ABMON_3	LC_TIME	abbreviated name of the third month
ABMON_4	LC_TIME	abbreviated name of the fourth month
ABMON_5	LC_TIME	abbreviated name of the fifth month
ABMON_6	LC_TIME	abbreviated name of the sixth month
ABMON_7	LC_TIME	abbreviated name of the seventh month
ABMON_8	LC_TIME	abbreviated name of the eighth month
ABMON_9	LC_TIME	abbreviated name of the ninth month
ABMON_10	LC_TIME	abbreviated name of the tenth month
ABMON_11	LC_TIME	abbreviated name of the eleventh month
ABMON_12	LC_TIME	abbreviated name of the twelfth month
ERA	LC_TIME	era description segments
ERA_D_FMT	LC_TIME	era date format string
ERA_D_T_FMT	LC_TIME	era date and time format string
ERA_T_FMT	LC_TIME	era time format string
ALT_DIGITS	LC_TIME	alternative symbols for digits
RADIXCHAR	LC_NUMERIC	radix character
THOUSEP	LC_NUMERIC	separator for thousands
YESEXPR	LC_MESSAGES	affirmative response expression
NOEXPR	LC_MESSAGES	negative response expression
YESSTR	LC_MESSAGES	affirmative response for yes/no queries
NOSTR	LC_MESSAGES	negative response for yes/no queries
CRNCYSTR	LC_MONETARY	currency symbol, preceded by – if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character

If the locale's value for `p_cs_precedes` and `n_cs_precedes` do not match, the value of `nl_langinfo(CRNCYSTR)` is unspecified.

The `langinfo.h` header declares the following as a function:

```
char *nl_langinfo(nl_item item);
```

Inclusion of the `<langinfo.h>` header may also make visible all symbols from `<nl_types.h>`.

APPLICATION USAGE

Users are advised to use the `strftime()` function to access date and time information defined in category `LC_TIME`. The `localeconv()` function should be used to access information corresponding to `RADIXCHAR`, `THOUSEP` and `CRNCYSTR`.

SEE ALSO

`localeconv(3C)`, `nl_langinfo(3C)`, `strfmon(3C)`, `strftime(3C)`.

"Programmer's Guide: Internationalization – Localization".

NAME

`lfs` - Large File Summit

DESCRIPTION**The problem with large files (> 2 GB)**

The 32-bit-oriented UNIX systems support file sizes of at most $2^{31}-1$ bytes. This restriction results from the definition of the `off_t` data type as "signed 32-bit long". However, this is no longer sufficient for present day applications which contain video, audio and image files as well as large databases. While the present 32-bit systems can easily handle the computing aspects of application processing, they also have to be able to support maximum file sizes which are much larger than traditional file sizes ("Large File Support", LFS).

The problem can be solved with a full portation from 32 to 64 bit where, for example, the data type definition for `off_t` is changed to a "signed 64-bit long".

This type of portation would affect both the kernel as well as all applications, and would involve a serious amount of effort. A 32-bit UNIX system which specifically supports file sizes > 2 GB or a 64-bit UNIX system, which "by definition" can handle very large file sizes, has to take account of the following factors:

- existing 32-bit binary programs cannot be changed,
- existing application software will, if necessary, be ported gradually from 32 to 64 bit (portation as 64-bit program)
- existing application software will only be ported to 64 bit in terms of file processing because of the effort involved (the application will remain a 32-bit program and will simply use the LFS interfaces).

Furthermore, interoperability between 32-bit and 64-bit applications must be guaranteed.

Leading system suppliers and vendors have come together in the framework of the "Large File Summit" to come up with a series of amendments to the existing Single UNIX Specification (SUS) on the basis of which both new and old programs can access files of "any" size. These amendments will be incorporated in the next SUS from X/Open. Further details of the "Large File Summit" and the LFS specification can be found at <http://www.sas.com/standards/large.file>.

Requirements

The following requirements were considered in the LFS specification:

- Protection of existing programs (in relation to overflow of data defined as `off_t`, e.g. `st_size` in `struct stat`)
- Protection of large files against programs that can only process files of up to 2 GB (and which would endanger the data of a large file through incorrect inputs)
- Access to files of far in excess of 2 GB on 32-bit operating systems or 32-bit applications on 64-bit operating systems
- Full compliance with SUS
- Extension of SUS

Concepts

The consistent implementation of a small number of key technical concepts served as a "guide" for putting together the LFS specification:

Various `off_t` sizes

During a transition period moving from pure 32-bit to pure 64-bit systems and applications, two sizes of `off_t` (and derived data types) must be supported:

- in the case of systems with a mixed `off_t` environment (`off_t` can be 32-bit or 64-bit long in any binary program),

- in the case of systems with a selectable `off_t` environment (`off_t` is either 32-bit or 64-bit long in any binary program),
- in the case of networks, client-servers can have different `off_t` definitions.

Offset maximum

Most but unfortunately not all numeric data in the SUS is defined as "non-transparent" (opaque) data types (i.e. the SUS generally does not use the basic data types of the C language; `ulimit(2)` is an exception here, for example, which means that the `ulimit` system call is not dealt with in the LFS specification). These derived data types can be used instead of the basic data types of the C language in order to produce a portable, source-compatible interface to the operating system and thus avoid the aforementioned overflow problems, for example. In the case of the linked binary programs, however, the "non-transparent" datatypes must be mapped to basic types (e.g. to 32-bit long). In the case of data, for example, that is used to display the file size or the current file position, an overflow can arise in the value range if the file to be processed is larger than 2 GB.

To protect these pure 32-bit binary programs and to protect the data in a large file, the concept of the "offset maximum" has been introduced: When a file is opened [e.g. with `open(2)` or `creat(2)`] the application program indicates (implicitly with 64-bit programs or explicitly using a flag) whether it can process files in excess of 2 GB. In this case, the so-called "offset maximum" is placed internally in the kernel in the file description linked with the file descriptor: the "offset maximum" corresponds to the value for the maximum file position for positioning in the file.

In the case of 32-bit programs, which do not specify with `open/creat` that they can process a large file, an "offset maximum" of 2 GB–1 is used.

All operations of an application,

- whose "offset maximum" does not match the current file size [`open(2)` , `creat(2)` etc.]
- or which cannot display the current file size correctly [`stat(2)` , `fstat(2)` , `fcntl(2)` etc.]
- or where it is attempted to position beyond the "offset maximum" [explicitly with `lseek(2)` , implicitly with `write(2)` / `read(2)`]

are aborted with errors.

EOVERFLOW

In the current SUS, an error is not defined for the situation where the "offset maximum" is exceeded or where a date cannot be displayed correctly. `EOVERFLOW` is an existing error type which has to be included in the descriptions of the relevant system interfaces so that the new error condition can be passed on to the applications.

Development models

In addition to the interface definitions for using different sizes for `off_t`, development models are also considered in the LFS specification which can be used to translate the source code of existing applications which are to process files in excess of 2 GB.

It is necessary here to specify the size of `off_t` as well as all assigned data types:

Selectable `off_t`

In this model, the sizes of `off_t` valid for compilation are defined and the corresponding libraries, include files and data types are selected during the compiling and linking process. All binary programs involved use the same size for `off_t` as provided by the development environment as 32-bit or 64-bit long (e.g. as data type "long long", if the application is compiled as a 32-bit program with LFS).

Explicit `off_t`

In this model, the sizes of `off_t` are specified during application development (and thereby established in the source code). The explicitly specified system interface uses an `off_t` of a specified length.

In a 32-bit application, the use of `open()` implies, for example, an `off_t` of 32 bits, and the use of `open64()` or `open(O_LARGEFILE)` implies an `off64_t` of 64 bits. (This also applies for

using other explicit 64-bit versions, such as `lseek64()`, see the listing under **SEE ALSO**.) This model is also extremely useful for supporting gradual conversions but is not supported directly in the SUS. It simply represents a temporary extension of the SUS (see LFS specification, chapter 3).

NOTES

Files larger than 2 GB and file systems larger than 4 GB are only supported for the file system type `vxfs` (Veritas).

SEE ALSO

`creat64(2)`, `fstat64(2)`, `fstatvfs64(2)`, `getrlimit64(2)`, `lstat64(2)`, `mmap64(2)`, `open64(2)`, `setrlimit64(2)`, `stat64(2)`, `statvfs64(2)`, `fgetpos64(3C)`, `fsetpos64(3C)`, `ftruncate64(3C)`, `ftw64(3C)`, `lockf64(3C)`, `nftw64(3C)`, `readdir64(3C)`, `truncate64(3C)`, `fopen64(3S)`, `freopen64(3S)`, `fseeko64(3S)`, `ftello64(3S)`, `lseek64(3S)`, `tmpfile64(3S)`.

NAME

libgen - definitions for pattern matching functions

SYNOPSIS

```
#include <libgen.h>
```

DESCRIPTION

The header <libgen.h> declares the following external variable:

```
extern char* __loc1
```

Used by `regex()` to report pattern location.

The following are declared as functions and may also be defined as macros:

```
char *basename (char *path);  
char *dirname (char *path);  
char *regcmp(const char *string1, ... );  
char *regex(const char *re1, const char *subject, ... );
```

SEE ALSO

[regcmp\(3G\)](#) .

NAME

locale - category macros

SYNOPSIS

```
#include <locale.h>
```

DESCRIPTION

The `<locale.h>` header provides a definition for structure `lconv`, which includes at least the following members:

```
char    *currency_symbol
char    *decimal_point
char    frac_digits
char    *grouping
char    *int_curr_symbol
char    int_frac_digits
char    *mon_decimal_point
char    *mon_grouping
char    *mon_thousands_sep
char    *negative_sign
char    n_cs_precedes
char    n_sep_by_space
char    n_sign_posn
char    *positive_sign
char    p_cs_precedes
char    p_sep_by_space
char    p_sign_posn
char    *thousands_sep
```

The `<locale.h>` header defines `NULL` (as defined in `<stddef.h>`) and at least the following as macros:

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME
```

which expand to distinct integral-constant expressions, for use as the first argument to the `setlocale()` function.

Additional macro definitions, beginning with the characters `LC_` and an upper-case letter, may also be given here.

The following are declared as functions and may also be defined as macros:

```
struct lconv *localeconv (void);
char setlocale(int category, const char *locale);
```

SEE ALSO

[localeconv\(3C\)](#), [setlocale\(3C\)](#).

NAME

`math` - mathematical functions and constants

SYNOPSIS

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

DESCRIPTION

This header file contains declarations of all the functions in the Math Library, as well as various functions in the C Library that return floating-point values.

It defines the structure and constants used by the `matherr(3M)` error-handling mechanisms, including the following constant used as an error-return value:

HUGE The maximum value of a single-precision floating-point number.

The `<math.h>` header provides for the following constants. The values are of type `double` and are accurate within the precision of the `double` type.

<code>M_E</code>	Value of e
<code>M_LOG2E</code>	Value of $\log_2 e$
<code>M_LOG10E</code>	Value of $\log_{10} e$
<code>M_LN2</code>	Value of $\log_e 2$
<code>M_LN10</code>	Value of $\log_e 10$
<code>M_PI</code>	Value of π
<code>M_PI_2</code>	Value of $\pi/2$
<code>M_PI_4</code>	Value of $\pi/4$
<code>M_1_PI</code>	Value of $1/\pi$
<code>M_2_PI</code>	Value of $2/\pi$
<code>M_2_SQRTPI</code>	Value of $2/\sqrt{\pi}$
<code>M_SQRT2</code>	Value of $\sqrt{2}$
<code>M_SQRT1_2</code>	Value of $1/\sqrt{2}$

The following mathematical constants are also defined in this header file:

MAXFLOAT

The maximum value of a single-precision floating point number.

HUGE_VAL

Positive infinity of type `double` (need not be representable as `float`), used to indicate errors.

For the definitions of various machine-dependent constants, see `values(5)`.

The following are declared as functions and may also be defined as macros:

```

double acos(double x);
double asin(double x);
double atan(double x);
double atan2(double y, double x);
double ceil(double x);
double cos(double x);
double cosh(double x);
double exp(double x);
double fabs(double x);
double floor(double x);
double fmod(double x, double y);
double frexp(double value, int *exp);
double ldexp(double value, int exp);
double log(double x);
double
log10(double x);
double modf(double value, double *iptr);
double pow(double x, double y);
double sin(double x);
double sinh(double x);
double sqrt(double x);
double tan(double x);
double tanh(double x);
double erf(double x);
double erfc(double x);
double gamma(double x);
double hypot(double x, double y);
double j0(double x);
double j1(double x);
double jn(int n, double x);
double
lgamma(double x);
double y0(double x);
double y1(double x);
double yn(int n, double x);
int  isnan(double x);
double acosh(double x);
double asinh(double x);
double atanh(double x);
double cbrt(double x);
double expm1(double x);
int  ilogb(double x);
double log1p(double x);
double logb(double x);
double nextafter(double x, double y);
double remainder(double x, double y);
double
rint(double x);
double scalb(double x, double n);

```

The following external variable is defined:

```
extern int signgam;
```

SEE ALSO

`frexp(3C)`, `isnan(3C)`, `bessel(3M)`, `erf(3M)`, `exp(3M)`, `floor(3M)`, `gamma(3M)`, `hypot(3M)`, `matherr(3M)`, `sinh(3M)`, `trig(3M)`, `values(5)`.

meta-ascii(5)**NAME**

meta-ascii - ISO 8859-1 character set

DESCRIPTION

The ISO 8859-1 code is an 8-bit code which represents a real superset of the ASCII code and contains most of the metacharacters used in the European language area.

The ASCII table assignment in `ascii(5)` is valid for the code assignment of 0 – 127₁₀. The following tables specify the assignment for the 128₁₀ – 255₁₀ code area.

Hex.	Oct.	Dec.	Graphic	Description
80	200	128		– undefined control code –
81	201	129		– undefined control code –
82	202	130		– undefined control code –
83	203	131		– undefined control code –
84	204	132	IND	Index
85	205	133	NEL	Next Line
86	206	134	SSA	Start of Selected Area
87	207	135	ESA	End of Selected Area
88	210	136	HTS	Horizontal Tabulation Set
89	211	137	HTJ	Horizontal Tab with Justification
8A	212	138	VTs	Vertical Tabulation Set
8B	213	139	PLD	Partial Line Down
8C	214	140	PLU	Partial Line Up
8D	215	141	RI	Reverse Index
8E	216	142	SS2	Single Shift 2
8F	217	143	SS3	Single Shift 3
90	220	144	DCS	Device Control String
91	221	145	PU1	Private Use One
92	222	146	PU2	Private Use Two
93	223	147	STS	Set Transmit State
94	224	148	CCH	Cancel Character
95	225	149	MW	Message Waiting
96	226	150	SPA	Start of Protected Area
97	227	151	EPA	End of Protected Area
98	230	152		– undefined control code –
99	231	153		– undefined control code –
9A	232	154		– undefined control code –

9B	233	155	CSI	Control Sequence Introducer
9C	234	156	ST	String Terminator
9D	235	157	OSC	Operating System Control
9E	236	158	PM	Privacy Message
9F	237	159	APC	Application Program Control

Hex.	Oct.	Dec.	Graphic	Description
A0	240	160		NBSP0, Non Breaking Space (blank as part of a word)
A1	241	161	¡	Inverse exclamation mark
A2	242	162	¢	U.S. cent symbol
A3	243	163	£	British pound symbol
A4	244	164	¤	General currency symbol
A5	245	165	¥	Japanese yen symbol
A6	246	166	ERROR	Broken vertical line
A7	247	167	§	Section symbol
A8	250	168	¨	0Umlaut (diaeresis) accent
A9	251	169	ERROR	Copyright symbol
AA	252	170	ª	Feminine ordinal indicator
AB	253	171	«	Opening guillemets (angle quotes)
AC	254	172	¬	NOT0 sign
AD	255	173	YH0	SHY0, Soft Hyphen (possible separator in a word)
AE	256	174	ERROR	Registered symbol
AF	257	175	—	Over line (high line, Macron)
B0	260	176		Degree (ring)
B1	261	177		Plus/minus sign
B2	262	178	²	0Superscript 2
B3	263	179	³	0Superscript 3
B4	264	180	´	Acute accent
B5	265	181	µ	Lowercase mu (micro sign)
B6	266	182	ERROR	0Pilcrow (paragraph sign)
B7	267	183	ERROR	0Middle dot
B8	270	184	ERROR	Cedilla
B9	271	185	¹	0Superscript 1
BA	272	186	º	Masculine ordinal indicator
BB	273	187	»	Closing guillemets (angle quotes)
BC	274	188	¼	One fourth (one quarter)

BD	275	189	½	One half
BE	276	190	¾	Three fourth
BF	277	191	¿	Inverse question mark

Hex.	Oct.	Dec.	Graphic	Description
C0	300	192	À	0Uppercase A grave accent
C1	301	193	Á	0Uppercase A acute accent
C2	302	194	Â	0Uppercase A circumflex
C3	303	195	Ã	0Uppercase A tilde
C4	304	196	Ä	0Uppercase A umlaut or diaeresis
C5	305	197	Å	0Uppercase A degree
C6	306	198	Æ	0Uppercase AE ligature
C7	307	199	Ç	0Uppercase C cedilla
C8	310	200	È	0Uppercase E grave accent
C9	311	201	É	0Uppercase E acute accent
CA	312	202	Ê	0Uppercase E circumflex
CB	313	203	Ë	0Uppercase E umlaut or diaeresis
CC	314	204	Ì	0Uppercase I grave accent
CD	315	205	Í	0Uppercase I acute accent
CE	316	206	Î	0Uppercase I circumflex
CF	317	207	Ï	0Uppercase I umlaut or diaeresis
D0	320	208	Ð	0Uppercase Icelandic ETH
D1	321	209	Ñ	0Uppercase N tilde
D2	322	210	Ò	0Uppercase O grave accent
D3	323	211	Ó	0Uppercase O acute accent
D4	324	212	Ô	0Uppercase O circumflex
D5	325	213	Õ	0Uppercase O tilde
D6	326	214	Ö	0Uppercase O umlaut or diaeresis
D7	327	215	¥	0Multiply sign
D8	330	216	Ø	0Uppercase O crossbar
D9	331	217	Ù	0Uppercase U grave accent
DA	332	218	Ú	0Uppercase U acute accent
DB	333	219	Û	0Uppercase U circumflex
DC	334	220	Ü	0Uppercase U umlaut or diaeresis
DD	335	221	Ý	0Uppercase Y acute accent
DE	336	222	þ	0Uppercase Icelandic Thorn
DF	337	223	ß	0Sharp s

Hex.	Oct.	Dec.	Graphic	Description

E0	340	224	à	0Lowercase a grave accent
E1	341	225	á	0Lowercase a acute accent
E2	342	226	â	0Lowercase a circumflex
E3	343	227	ã	0Lowercase a tilde
E4	344	228	ä	0Lowercase a umlaut or diaeresis
E5	345	229	å	0Lowercase a degree
E6	346	230	æ	0Lowercase ae ligature
E7	347	231	ç	0Lowercase c cedilla
E8	350	232	è	0Lowercase e grave accent
E9	351	233	é	0Lowercase e acute accent
EA	352	234	ê	0Lowercase e circumflex
EB	353	235	ë	0Lowercase e umlaut or diaeresis
EC	354	236	ì	0Lowercase i grave accent
ED	355	237	í	0Lowercase i acute accent
EE	356	238	î	0Lowercase i circumflex
EF	357	239	ï	0Lowercase i umlaut or diaeresis
F0	360	240	Ö	0Lowercase Icelandic ETH
F1	361	241	ñ	0Lowercase n tilde
F2	362	242	ò	0Lowercase o grave accent
F3	363	243	ó	0Lowercase o acute accent
F4	364	244	ô	0Lowercase o circumflex
F5	365	245	õ	0Lowercase o tilde
F6	366	246	ö	0Lowercase o umlaut or diaeresis
F7	367	247		0Division sign
F8	370	248	ø	0Lowercase o crossbar
F9	371	249	ù	0Lowercase u grave accent
FA	372	250	ú	0Lowercase u acute accent
FB	373	251	û	0Lowercase u circumflex
FC	374	252	ü	0Lowercase u umlaut or diaeresis
FD	375	253	\zy´	0Lowercase y acute accent
FE	376	254	þ	0Lowercase Icelandic Thorn
FF	377	255	ERROR	0Lowercase y umlaut or diaeresis

NOTES

Depending on the type of terminal you are using, you can display each of the characters listed here using the `echo '\octal-number'` command.

SEE ALSO

[ascii\(5\)](#).

NAME

mman - memory management declarations

SYNOPSIS

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

DESCRIPTION

The following protection options are defined:

PROT_READ

Page can be read.

PROT_WRITE

Page can be written.

PROT_EXEC

Page can be executed.

PROT_NONE

Page can not be accessed.

The following *flag* options are defined:

MAP_SHARED

Share changes.

MAP_PRIVATE

Changes are private.

MAP_FIXED

Interpret addr exactly.

The following flags are defined for `msync()`:

MS_ASYNC

Perform asynchronous writes.

MS_SYNC

Perform synchronous writes.

MS_INVALIDATE

Invalidate mappings.

The `size_t` and `off_t` types are defined as described in `<sys/types.h>`.

The following are declared in `<sys/mman.h>` as functions and may also be defined as macros:

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd,
           off_t off);
```

```
int mprotect(void *addr, size_t len, int prot);
```

```
int msync(void *addr, size_t len, int flags);
```

```
int munmap(void *addr, size_t len);
```

The following is declared as a function and may also be defined as a macro:

```
void *mmap64(void *addr, size_t len, int prot, int flags,
             int fd, off64_t offset);
```

SEE ALSO

[mmap\(2\)](#), [mprotect\(2\)](#), [munmap\(2\)](#), [msync\(3C\)](#), [lfs\(5\)](#).

NAME

monetary - monetary types

SYNOPSIS

```
#include <monetary.h>
```

DESCRIPTION

The `<monetary.h>` header defines the following data types through `typedef`:

`size_t`

As described in `<stddef.h>`.

`ssize_t`

As described in `<sys/types.h>`.

The following is declared as a function and may also be defined as a macro:

`ssize_t`

```
strfmon(char *s, size_t maxsize, const char *format, ...);
```

SEE ALSO

[strfmon\(3C\)](#).

NAME

msg - message queue structures

SYNOPSIS

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

DESCRIPTION

The `<sys/msg.h>` header defines the following constant and members of the structure `msqid_ds`.

The following data types are defined through `typedef`:

`msgqnum_t`

Used for the number of messages in the message queue.

`msglen_t`

Used for the number of bytes allowed in a message queue.

These types are unsigned integer types that are able to store values at least as large as a type `unsigned short`.

Message operation flag:

`MSG_NOERROR`

No error if big message.

The structure `msqid_ds` contains the following members:

`struct ipc_perm msg_perm`
operation permission structure

`msgqnum_t msg_qnum`
number of messages currently on queue

`msglen_t msg_qbytes`
maximum number of bytes allowed on queue

`pid_t msg_lspid`
process ID of last `msgsnd()`

`pid_t msg_lrpid`
process ID of last `msgrcv()`

`time_t msg_stime`
time of last `msgsnd()`

`time_t msg_rtime`
time of last `msgrcv()`

`time_t msg_ctime`
time of last change

The `pid_t`, `time_t`, `key_t` and `size_t` types are defined as described in `<sys/types.h>`.

The following are declared as functions and may also be defined as macros:

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

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

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp,
           int msgflg);
```

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

In addition, all of the symbols from `<sys/ipc.h>` will be defined when this header is included.

SEE ALSO

[msgctl\(2\)](#), [msgget\(2\)](#), [msgrcv\(2\)](#), [msgsnd\(2\)](#), [types\(5\)](#).

NAME

ndbm - definitions for ndbm database operations

SYNOPSIS

```
#include <ndbm.h>
```

DESCRIPTION

The `<ndbm.h>` header defines the `datum` type as a structure that includes at least the following members:

```
void *dptr
```

A pointer to the application's data.

```
size_t dsize
```

The size of the object pointed to by `dptr`.

The `size_t` type is defined through `typedef` as described in `<stddef.h>` [see [stddef\(5\)](#)].

The `<ndbm.h>` header defines the `DBM` type through `typedef`.

The following constants are defined as possible values for the `store_mode` argument to `dbm_store()`:

```
DBM_INSERT
```

Insertion of new entries only.

```
DBM_REPLACE
```

Allow replacing existing entries.

The following are declared as functions and may also be defined as macros:

```
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);
```

The `mode_t` type is defined through `typedef` as described in `<sys/types.h>` [see [types\(5\)](#)].

SEE ALSO

[ndbm\(3C\)](#), [stddef\(5\)](#), [types\(5\)](#).

nl_types(5)**NAME**

`nl_types` - native language data types

SYNOPSIS

```
#include <nl_types.h>
```

DESCRIPTION

This header file contains the following definitions:

`nl_catd`

used by the message catalog functions `catopen()`, `catgets()` and `catclose()` to identify a catalog descriptor.

`nl_item`

used by `nl_langinfo()` to identify the language environment. Values for objects of type `nl_item` are defined in `<langinfo.h>`.

In addition, the following constants are defined in `<nl_types.h>`:

`NL_SETD`

used by `gencat()` when no `$set` directive is specified in a message text source file. This constant can be used in subsequent calls to `catgets()` as the value of the set identifier parameter.

`NL_MSGMAX`

maximum number of messages per set

`NL_SETMAX`

maximum number of sets per catalog.

`NL_TEXTMAX`

maximum size of a message.

`DEF_NLSPAT`

the default search path for locating catalogs.

`NL_CAT_LOCALE`

is passed to `catopen()` as *oflag* argument to ensure that message catalog selection depends on `LC_MESSAGES` and not directly on the `LANG` environment variable.

`NL_MAXPATHLEN`

maximum length of path specification of the catalog name for `catopen()`

`NL_MAX_OPENED`

maximum number of catalogs that can be open simultaneously.

The following are declared as functions and may also be defined as macros:

```
int      catclose(nl_catd catd);
```

```
char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

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

SEE ALSO

`gencat(1)`, `catgets(3C)`, `catopen(3C)`, `nl_langinfo(3C)`, `langinfo(5)`.

"Programmer's Guide: Internationalization – Localization".

poll(5) - syslog(5)

NAME

`poll` - definitions for the `poll()` function

SYNOPSIS

```
#include <poll.h>
```

DESCRIPTION

The `<poll.h>` header defines the `pollfd` structure that includes at least the following member:

```
int    fd
```

The following descriptor being polled.

```
short int  events
```

The input event flags (see below).

```
short int  revents
```

The output event flags (see below).

The `<poll.h>` header defines the following type through typedef:

```
nfds_t
```

An unsigned integral type used for the number of file descriptors.

The following symbolic constants are defined, zero or more of which may be OR-ed together to form the `events` or `revents` members in the `pollfd` structure:

`POLLIN`

Same effect as `POLLRDNORM` | `POLLRDBAND`.

`POLLRDNORM`

Data on priority band 0 may be read.

`POLLRDBAND`

Data on priority bands greater than 0 may be read.

`POLLPRI`

High priority data may be read.

`POLLOUT`

Same value as `POLLWRNORM`.

`POLLWRNORM`

Data on priority band 0 may be written.

`POLLERR`

An error has occurred (`revents` only).

`POLLHUP`

Device has been disconnected (`revents` only).

`POLLNVAL`

Invalid `fd` member (`revents` only).

The `<poll.h>` header declares the following function which may also be defined as a macro:

```
int poll(struct pollfd fds[], nfds_t nfd, int timeout);
```

SEE ALSO

[poll\(2\)](#).

NAME

pthread - threads

SYNOPSIS

#include <pthread.h>

DESCRIPTION

The <pthread.h> header defines the following symbols:

```

PTHREAD_CANCEL_ASYNCHRONOUS
PTHREAD_CANCEL_ENABLE
PTHREAD_CANCEL_DEFERRED
PTHREAD_CANCEL_DISABLE
PTHREAD_CANCELED
PTHREAD_COND_INITIALIZER
PTHREAD_CREATE_DETACHED
PTHREAD_CREATE_JOINABLE
PTHREAD_EXPLICIT_SCHED
PTHREAD_INHERIT_SCHED
PTHREAD_MUTEX_DEFAULT
PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_INITIALIZER
PTHREAD_MUTEX_RECURSIVE
PTHREAD_ONCE_INIT
PTHREAD_PRIO_INHERIT
PTHREAD_PRIO_NONE
PTHREAD_PRIO_PROTECT
PTHREAD_PROCESS_SHARED
PTHREAD_PROCESS_PRIVATE
PTHREAD_RWLOCK_INITIALIZER
PTHREAD_SCOPE_PROCESS
PTHREAD_SCOPE_SYSTEM

```

The pthread_attr_t, pthread_cond_t, pthread_condattr_t, pthread_key_t, pthread_mutex_t, pthread_mutexattr_t, pthread_once_t, pthread_rwlock_t, pthread_rwlockattr_t and pthread_t types are defined as described in <sys/types.h> [see [types\(5\)](#)].

The following are declared as functions and may also be declared as macros.

```

int  pthread_attr_destroy(pthread_attr_t *);
int  pthread_attr_getdetachstate(const pthread_attr_t *, int *);
int  pthread_attr_getguardsize(const pthread_attr_t *, size_t *);
int  pthread_attr_getinheritsched(const pthread_attr_t *, int *);
int  pthread_attr_getschedparam(const pthread_attr_t *,
    struct sched_param *);
int  pthread_attr_getschedpolicy(const pthread_attr_t *, int *);
int  pthread_attr_getscope(const pthread_attr_t *, int *);
int  pthread_attr_getstackaddr(const pthread_attr_t *, void **);
int  pthread_attr_getstacksize(const pthread_attr_t *, size_t *);
int  pthread_attr_init(pthread_attr_t *);
int  pthread_attr_setdetachstate(pthread_attr_t *, int);
int  pthread_attr_setguardsize(pthread_attr_t *, size_t);
int  pthread_attr_setinheritsched(pthread_attr_t *, int);
int  pthread_attr_setschedparam(pthread_attr_t *,
    const struct sched_param *);

```

```

int  pthread_attr_setschedpolicy(pthread_attr_t *, int);
int  pthread_attr_setscope(pthread_attr_t *, int);
int  pthread_attr_setstackaddr(pthread_attr_t *, void *);
int  pthread_attr_setstacksize(pthread_attr_t *, size_t);
int  pthread_cancel(pthread_t);
void pthread_cleanup_push(void (*)(void*), void *);
void pthread_cleanup_pop(int);
int  pthread_cond_broadcast(pthread_cond_t *);
int  pthread_cond_destroy(pthread_cond_t *);
int  pthread_cond_init(pthread_cond_t *, const pthread_condattr_t *);
int  pthread_cond_signal(pthread_cond_t *);
int  pthread_cond_timedwait(pthread_cond_t *,
    pthread_mutex_t *, const struct timespec *);
int  pthread_cond_wait(pthread_cond_t *);
int  pthread_condattr_destroy(pthread_condattr_t *);
int  pthread_condattr_getpshared(const pthread_condattr_t *, int *);
int  pthread_condattr_init(pthread_condattr_t *);
int  pthread_condattr_setpshared(pthread_condattr_t *, int);
int  pthread_create(pthread_t *, const pthread_attr_t *,
    void (*)(void*), void *);
int  pthread_detach(pthread_t);
int  pthread_equal(pthread_t, pthread_t);
void pthread_exit(void *);
int  pthread_getconcurrency(void);
int  pthread_getschedparam(pthread_t, int *, struct sched_param *);
void *pthread_getspecific(pthread_key_t);
int  pthread_join(pthread_t, void **);
int  pthread_key_create(pthread_key_t *, void (*)(void*));
int  pthread_key_delete(pthread_key_t);
int  pthread_mutex_destroy(pthread_mutex_t *);
int  pthread_mutex_getprioceiling(const pthread_mutex_t *, int *);
int  pthread_mutex_init(pthread_mutex_t *, const pthread_mutexattr_t *);
int  pthread_mutex_lock(pthread_mutex_t *);
int  pthread_mutex_setprioceiling(pthread_mutex_t *, int, int *);
int  pthread_mutex_trylock(pthread_mutex_t *);
int  pthread_mutex_unlock(pthread_mutex_t *);
int  pthread_mutexattr_destroy(pthread_mutexattr_t *);
int  pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *, int *);
int  pthread_mutexattr_getprotocol(const pthread_mutexattr_t *, int *);
int  pthread_mutexattr_getpshared(const pthread_mutexattr_t *, int *);
int  pthread_mutexattr_gettype(pthread_mutexattr_t *, int *);
int  pthread_mutexattr_init(pthread_mutexattr_t *);
int  pthread_mutexattr_setprioceiling(pthread_mutexattr_t *, int);
int  pthread_mutexattr_setprotocol(pthread_mutexattr_t *, int);
int  pthread_mutexattr_setpshared(pthread_mutexattr_t *, int);
int  pthread_mutexattr_settype(pthread_mutexattr_t *, int);
int  pthread_once(pthread_once_t *, void (*)(void));
int  pthread_rwlock_destroy(pthread_rwlock_t *);
int  pthread_rwlock_init(pthread_rwlock_t *,
    const pthread_rwlockattr_t *);
int  pthread_rwlock_rdlock(pthread_rwlock_t *);
int  pthread_rwlock_tryrdlock(pthread_rwlock_t *);
int  pthread_rwlock_trywrlock(pthread_rwlock_t *);
int  pthread_rwlock_unlock(pthread_rwlock_t *);
int  pthread_rwlock_wrlock(pthread_rwlock_t *);
int  pthread_rwlockattr_destroy(pthread_rwlockattr_t *);
int  pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *, int *);
int  pthread_rwlockattr_init(pthread_rwlockattr_t *);
int  pthread_rwlockattr_setpshared(pthread_rwlockattr_t *, int);

```



```
pthread_t pthread_self(void);
int pthread_setcancelstate(int, int *);
int pthread_setcanceltype(int, int *);
int pthread_setconcurrency(int);
int pthread_setschedparam(pthread_t, int *,
    const struct sched_param *);
int pthread_setspecific(pthread_key_t, const void *);
void pthread_testcancel(void);
```

Inclusion of the `<pthread.h>` header will make visible symbols defined in the headers `<sched.h>` and `<time.h>`.

SEE ALSO

```
pthread_attr_getguardsize(3T), pthread_attr_init(3T), pthread_attr_setscope(3T),
pthread_cancel(3T), pthread_cleanup_push(3T), pthread_condattr_init(3T),
pthread_cond_init(3T), pthread_cond_signal(3T), pthread_cond_wait(3T),
pthread_create(3T), pthread_detach(3T), pthread_equal(3T), pthread_exit(3T),
pthread_getconcurrency(3T), pthread_getschedparam(3T), pthread_intro(3T),
pthread_join(3T), pthread_key_create(3T), pthread_key_delete(3T),
pthread_mutexattr_gettype(3T), pthread_mutexattr_init(3T),
pthread_mutexattr_setprotocol(3T), pthread_mutex_init(3T), pthread_mutex_lock(3T),
pthread_mutex_setprioceiling(3T), pthread_once(3T), pthread_rwlockattr_init(3T),
pthread_rwlock_init(3T), pthread_rwlock_rdlock(3T), pthread_rwlock_unlock(3T),
pthread_rwlock_wrlock(3T), pthread_self(3T), pthread_setcancelstate(3T),
pthread_setspecific(3T), sched(5), time(5), types(5).
```

NAME

pwd - password structure

SYNOPSIS

```
#include <pwd.h>
```

DESCRIPTION

The `<pwd.h>` header provides a definition for `struct passwd`, which includes the following members:

```
char    *pw_name
    User's login name.
uid_t    pw_uid
    Numerical user ID.
gid_t    pw_gid
    Numerical group ID.
char    *pw_dir
    Initial working directory.
char    *pw_shell
    Program to use as shell.
```

The `gid_t` and `uid_t` types are defined as described in `<sys/types.h>` [see [types\(5\)](#)].

The following are declared as functions and may also be defined as macros:

```
struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
int    getpwnam_r(const char *nam, 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);
void    endpwent(void);
struct passwd *getpwent(void);
void    setpwent(void);
```

SEE ALSO

[getpwent\(3C\)](#), [types\(5\)](#).

NAME

re_comp - regular-expression-matching functions for re_comp()

SYNOPSIS

```
#include <re_comp.h>
```

DESCRIPTION

The following are declared as functions and may also be declared as macros:

```
char *re_comp(const char *string);  
int re_exec(const char *string);
```

SEE ALSO

re_comp(3C) .

NAME

regex - regular-expression-matching types

SYNOPSIS

```
#include <regex.h>
```

DESCRIPTION

The `regex.h` header defines the structures and symbolic constants used by the `regcomp()`, `regexexec()`, `regerror()`, and `regfree()` functions.

The structure type `regex_t` contains at least the following member:

Type	Name	Description
<code>size_t</code>	<code>re_nsub</code>	Number of parenthesized subexpressions.

The type `regoff_t` is defined as a signed arithmetic type that can hold the largest value that can be stored in either a type `off_t` or type `ssize_t`. The structure type `regmatch_t` contains at least the following members:

Type	Name	Description
<code>regoff_t</code>	<code>rm_so</code>	Byte offset from start of <i>string</i> to start to substring.
<code>regoff_t</code>	<code>rm_eo</code>	Byte offset from start of <i>string</i> of the first character after the end of substring.

Values for the *cflags* parameters to the `regcomp()` function:

- `REG_EXTENDED`
Use Extended Regular Expressions.
- `REG_ICASE`
Ignore case in match.
- `REG_NOSUB`
Report only success or fail in `regexexec()`.
- `REG_NEWLINE`
Change the handling of newline.
- `REG_LIT_NL`
The `nn` string is treated as a newline character.
- `REG_VIMODE`
`<<` or `>>` are interpreted as the start or the finish of a word.
- `REG_NSUB_ANCHOR`
The characters `^` and `$` do not function as anchors within subexpressions.
- `REG_EMPTY`
`regcomp()` returns an error value if *pattern* is a null string or empty.

Values for the *eflags* parameter to the `regexexec()` function.

- `REG_NOTBOL`
The circumflex character (`^`), when taken as a special character, will not match the beginning of *string*.
- `REG_NOTEOL`
The dollar sign (`$`), when taken as a special character, will not match the end of *string*.

The following constants are defined as error return values:

- `REG_NOMATCH`

`regexec()` failed to match.

`REG_INVARG`
An invalid parameter was specified.

`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_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.

`REG_EPATTERN`
A null or empty pattern was specified as a parameter and `regcomp()` was called with the `REG_EMPTY` flag.

`REG_ENOSYS`
The implementation does not support the function.

The following are declared as functions and may also be declared as macros:

```
int regcomp(regex_t *preg, const char *pattern, int cflags);
int regexec(const regex_t *preg, const char *string,
            size_t nmatch, regmatch_t pmatch[], int cflags);
size_t regerror(int errcode, const regex_t *preg,
                char *errbuf, size_t errbuf_size);
void regfree(regex_t *preg);
```

The implementation may define additional macros or constants using names beginning with `REG_`.

SEE ALSO

[regcmp\(1\)](#), [regcomp\(3C\)](#), [regex\(3C\)](#), [regcmp\(3G\)](#), [regexpr\(3G\)](#), [expressions\(5\)](#), [regexp\(5\)](#)

NAME

regex:
 compile, step, advance, loc1, loc2, locs - compile/match regular expressions

SYNOPSIS

```
#include <regex.h>
#define INIT declarations
#define GETC(void) getc code
#define PEEKC(void) peekc code
#define UNGETC(void) ungetc code
#define RETURN(ptr) return code
#define ERROR(val) error code
□
int□□□ advance(const char *string, const char *expbuf);
char□ *compile(char *instring, char *expbuf, const char *endbuf,□
□□□□□□□□□□□□□□ int eof);
int□□□ step(const char *string, const char *expbuf);
extern char *loc1, *loc2, *locs;
```

DESCRIPTION

These functions are general purpose routines for manipulating regular expressions in programs that perform regular expression matching. These functions are defined by the `<regex.h>` header file.

The functions `step()` and `advance()` do pattern matching given a character string and a compiled regular expression as input.

The function `compile()` takes as input a regular expression as defined below and produces a compiled expression that can be used with `step()` or `advance()`.

A regular expression specifies a set of character strings. A member of this set of strings is said to be matched by the regular expression. Some characters have special meaning when used in a regular expression; other characters stand for themselves.

The regular expressions available for use with the `regex` functions are constructed as follows:

Expression

	Meaning
<i>c</i>	the character <i>c</i> where <i>c</i> is not a special character.
<code>\c</code>	the character <i>c</i> where <i>c</i> is any character, except a digit in the range 1–9.
<code>^</code>	the beginning of the line being compared.
<code>\$</code>	the end of the line being compared.
<code>.</code>	any character in the input.
<code>[s]</code>	any character in the set <i>s</i> , where <i>s</i> is a sequence of characters and/or a range of characters, e.g. <code>[c–c]</code> .
<code>[^s]</code>	any character not in the set <i>s</i> , where <i>s</i> is defined as above.
<i>r</i> *	zero or more successive occurrences of the regular expression <i>r</i> . The longest leftmost match is chosen.
<i>rx</i>	the occurrence of regular expression <i>r</i> followed by the occurrence of regular expression <i>x</i> (concatenation).
<i>r</i> { <i>m</i> , <i>n</i> }	

any number of m through n successive occurrences of the regular expression r . The regular expression $r\{m\}$ matches exactly m occurrences; $r\{m,\}$ matches at least m occurrences.

$\backslash(r)\backslash n$

the regular expression r . When $\backslash n$ (where n is a number greater than zero) appears in a constructed regular expression, it stands for the regular expression x where x is the n th regular expression enclosed in $\backslash($ and $\backslash)$ that appear earlier in the constructed regular expression. For example, $\backslash(r)\backslash x\backslash(y)\backslash z2$ is the concatenation of regular expressions $rxzy$.

The following characters have a special meaning unless they appear within square brackets or a $\backslash$ precedes them: $.$, $*$, $[$, $\backslash$. Other special characters such as $\$$ have special meanings in more restricted contexts.

The character $^$ at the beginning of an expression permits a successful match only immediately after a newline, and the character $\$$ at the end of an expression requires a trailing newline.

Two characters have special meaning only when used within square brackets. The character $-$ denotes a range, $[c-c]$, unless it is just after the open bracket or before the closing bracket, $[-c]$ or $[c-]$ in which case it has no special meaning. When used within square brackets, the character $^$ has the meaning *complement of* if it immediately follows the open square bracket (example: $[^c]$); elsewhere between square brackets (example: $[^]$) it stands for the ordinary character $^$.

The special meaning of the $\backslash$ operator can be escaped only by preceding it with another $\backslash$, e.g. $\backslash\backslash$.

Programs must have the following five macros declared before the `#include <regex.h>` statement. These macros are used by the `compile` routine. The macros `GETC`, `PEEKC`, and `UNGETC` operate on the regular expression given as input to `compile()`.

GETC This macro returns the value of the next character (byte) in the regular expression pattern. Successive calls to `GETC` should return successive characters of the regular expression.

PEEKC

This macro returns the next character (byte) in the regular expression. Immediately successive calls to `PEEKC` should return the same character, which should also be the next character returned by `GETC`.

UNGETC(*c*)

This macro causes the argument c to be returned by the next call to `GETC` and `PEEKC`. No more than one character of pushback is ever needed and this character is guaranteed to be the last character read by `GETC`. The return value of the macro `UNGETC(c)` is always ignored.

RETURN(*ptr*)

This macro is used on normal exit of the `compile()` routine. The value of the argument ptr is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.

ERROR(*val*)

This macro is the abnormal return from the `compile()` routine. The argument val is an error number. This call should never return.

The syntax of the `compile()` routine is as follows:

```
char *compile(char *instring, char *exbuf, const char *endbuf,
              int eof);
```

instring is never used explicitly by the `compile()` routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the `INIT` declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of `(char*)0` for this parameter.

exbuf is a character pointer. It points to the place where the compiled regular expression will be placed.

endbuf is one more than the highest address where the compiled regular expression may be placed. If the compiled expression does not fit in $(endbuf - exbuf)$ bytes, a call to `ERROR(50)` is made.

The parameter *eof* is the character which marks the end of the regular expression. This character is usually a `/`.

Each program that includes the `<regex.h>` header file must have a `#define` statement for `INIT`. It is used

for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for `GETC`, `PEEKC`, and `UNGETC`. Otherwise it can be used to declare external variables that might be used by `GETC`, `PEEKC` and `UNGETC`. (See **EXAMPLE** below.)

The first parameter to the `step()` and `advance()` functions is a pointer to a string of characters to be checked for a match. This string should be null terminated. The second parameter, `expbuf`, is the compiled regular expression which was obtained by a call to the function `compile()`.

The function `step()` returns non-zero if some substring of *string* matches the regular expression in `expbuf`. Otherwise it returns zero. If there is a match, two external character pointers are set as a side effect to the call to `step`. The variable `loc1` points to the first character that matched the regular expression; the variable `loc2` points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire input string, `loc1` will point to the first character of *string* and `loc2` will point to the null at the end of *string*.

The function `advance()` returns non-zero if the initial substring of *string* matches the regular expression in `expbuf`. If a match occurs, an external character pointer, `loc2`, is set as a side effect. The variable `loc2` points to the next character in *string* after the last character that matched.

When `advance()` encounters a `*` or `\{ \}` sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, `advance()` will back up along the string until it finds a match or reaches the point in the string that initially matched the `*` or `\{ \}`. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer `locs` is equal to the point in the string at sometime during the backing up process, `advance()` will break out of the loop that backs up and will return zero.

The external variables `circf`, `sed`, and `nbra` are reserved.

DIAGNOSTICS

The function `compile()` uses the macro `RETURN` on success and the macro `ERROR` on failure (see above). The functions `step()` and `advance()` return non-zero if they find a match. Otherwise they return zero.

Errors are:

- 11 range endpoint too large
- 16 bad number
- 25 `\digit` out of range
- 36 illegal or missing delimiter
- 41 no search string in memory
- 42 `\( \)` imbalance
- 43 too many `\(`
- 44 more than 2 numbers given in `\{ \}`
- 45 `}` expected after `\`
- 46 first number exceeds second in `\{ \}`
- 49 `[ ]` imbalance
- 50 regular expression too extensive

EXAMPLE

The following is an example of how regular expression macros and calls can be defined by an application program:

```
#define INIT register char *sp = instring;
#define GETC  (*sp++)
#define PEEKC  (*sp)
#define UNGETC(c) (--sp)
```



```
#define RETURN(*c) return;
#define ERROR(c) regerr
#include <regexp.h>

...
(void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
if (step(linebuf, expbuf))
    succeed;
```

SEE ALSO

[regcmp\(1\)](#), [regcomp\(3C\)](#), [regex\(3C\)](#), [regcmp\(3G\)](#), [regexpr\(3G\)](#), [expressions\(5\)](#), [regex\(5\)](#).

NAME

`resource` - definitions for resource operations

SYNOPSIS

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

DESCRIPTION

The `<sys/resource.h>` header defines the following symbolic constants as possible values of the *which* argument of `getpriority()` and `setpriority()`:

`PRIO_PROCESS`

Identifies *who* argument as a process ID.

`PRIO_PGRP`

Identifies *who* argument as a process group ID.

`PRIO_USER`

Identifies *who* argument as a user ID.

The following type is defined through `typedef`:

`rlim_t`

Unsigned integral type used for limit values.

The following symbolic constant is defined:

`RLIM_INFINITY`

A value of `rlim_t` indicating no limit.

`RLIM_SAVED_MAX`

A value of type `rlim_t` indicating an unrepresentable saved hard limit.

`RLIM_SAVED_CUR`

A value of type `rlim_t` indicating an unrepresentable saved soft limit.

On implementations where all resource limits are representable in an object of type `rlim_t`, `RLIM_SAVED_MAX` and `RLIM_SAVED_CUR` need not be distinct from `RLIM_INFINITY`.

The following symbolic constants are defined as possible values of the *who* parameter of `getrusage()`:

`RUSAGE_SELF`

Returns information about the current process.

`RUSAGE_CHILDREN`

Returns information about children of the current process.

The `<sys/resource.h>` header defines the `rlimit` structure that includes at least the following members:

`rlim_t rlim_cur`

The current (soft) limit.

`rlim_t rlim_max`

The hard limit.

The `<sys/resource.h>` header defines the `rusage` structure that includes at least the following members:

`struct timeval ru_utime`

User time used.

`struct timeval ru_stime`

System time used.

The `timeval` structure is defined as described in `<sys/time.h>` [see [sys_time\(5\)](#)].

The following symbolic constants are defined as possible values for the *resource* argument of `getrlimit()` and `setrlimit()`:

`RLIMIT_CORE`

Limit on size of core dump file.

`RLIMIT_CPU`

Limit on CPU time per process.

`RLIMIT_DATA`

Limit on data segment size.

`RLIMIT_FSIZE`

Limit on file size.

`RLIMIT_NOFILE`

Limit on number of open files.

`RLIMIT_STACK`

Limit on stack size.

`RLIMIT_AS`

Limit on address space size.

The following are declared as functions and may also be defined as macros:

```
int getpriority(int which, id_t who);
int getrlimit(int resource, struct rlimit *rlp);
int getrusage(int who, struct rusage *r_usage);
int setpriority(int which, id_t who, int priority);
int setrlimit(int resource, const struct rlimit *rlp);
```

The `id_t` type is defined through `typedef` as described in `<sys/types.h>` [see [types\(5\)](#)].

Inclusion of the `<sys/resource.h>` header may also make visible all symbols from `<sys/time.h>` [see [sys_time\(5\)](#)].

The following applies to the "explicit off_t" model.

The following data type is defined through `typedef`:

`rlim64_t`

Type used for limit values.

The type `rlim64_t` must be an extended unsigned arithmetic type that can represent correctly any non-negative value of an `off64_t`.

The following symbolic constants are defined:

`RLIM64_INFINITY`

A value of type `rlim64_t` indicating no limit.

`RLIM64_SAVED_MAX`

A value of type `rlim64_t` indicating an unrepresentable saved hard limit.

`RLIM64_SAVED_CUR`

A value of type `rlim64_t` indicating an unrepresentable saved soft limit.

On implementations where all resource limits are representable in an object of type `rlim64_t`, `RLIM64_SAVED_MAX` and `RLIM64_SAVED_CUR` need not be distinct from `RLIM64_INFINITY`.

The `rlimit64` structure is defined in the same way as the `rlimit` structure with the exception of the following members:

`rlim64_t rlim_cur`

The current (soft) limit.

`rlim64_t rlim_max`

The hard limit.

The following are declared as functions and may also be defined as macros:

```
int getrlimit64(int resource, struct rlimit64 *rlp);
int setrlimit64(int resource, const struct rlimit64 *rlp);
```

SEE ALSO

```
getrlimit(2), getrusage(2), getpriority(3), lfs(5), sys_time(5), types(5),
```

NAME

`sched` - execution scheduling (**REALTIME THREADS**)

SYNOPSIS

```
#include <sched.h>
```

DESCRIPTION

The **REALTIME THREADS** interfaces are not supported!

SEE ALSO

[pthread_intro\(3T\)](#).

NAME

`search` - search tables

SYNOPSIS

```
#include <search.h>
```

DESCRIPTION

The `<search.h>` header provides a type definition, `ENTRY`, for structure `entry` which includes the following members:

```
char *key
void *data
```

and defines `ACTION` and `VISIT` as enumeration data types through type definitions as follows:

```
enum { FIND, ENTER } ACTION;
enum { preorder, postorder, endorder, leaf } VISIT;
```

The `size_t` type is defined as described in `<sys/types.h>`.

Each of the following is declared as a function, or defined as a macro, or both:

```
int hcreate(size_t nel);
void hdestroy(void);
ENTRY *hsearch(ENTRY item, ACTION action);
void insque(void *element, void *pred);
void *lfind(const void *key, const void *base, size_t *nel,
            size_t width, int (*compar)(const void *, const void *));
void *lsearch(const void *key, void *base, size_t *nel,
              size_t width, int (*compar)(const void *, const void *));
void remque(void *element);
void *tdelete(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 *tsearch(const void *key, void **rootp,
              int (*compar)(const void *, const void *));
void twalk(const void *root,
           void (*action)(const void *, VISIT, int ));
```

SEE ALSO

[hsearch\(3C\)](#), [insque\(3C\)](#), [lsearch\(3C\)](#), [remque\(3C\)](#), [tsearch\(3C\)](#), [types\(5\)](#).

NAME

sem - semaphore facility

SYNOPSIS

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

DESCRIPTION

The `<sys/sem.h>` header defines the following constants and structures.

Semaphore operation flags:

`SEM_UNDO`

Set up adjust on exit entry.

Command definitions for the function `semctl()`:

`GETNCNT`

Get `semncnt`.

`GETPID`

Get `sempid`.

`GETVAL`

Get `semval`.

`GETALL`

Get all cases of `semval`.

`GETZCNT`

Get `semzcnt`.

`SETVAL`

Set `semval`.

`SETALL`

Set all cases of `semval`.

The structure `semid_ds` contains the following members:

```
struct ipc_perm    sem_perm
    Operation permission structure.
```

```
unsigned short int sem_nsems
    Number of semaphores in set.
```

```
time_t            sem_otime
    Last semop() time.
```

```
time_t            sem_ctime
    Last time changed by semctl().
```

The `pid_t`, `time_t`, `key_t` and `size_t` types are defined as described in `<sys/types.h>`.

A semaphore is represented by an anonymous structure containing the following members:

```
unsigned short int semval
    Semaphore value.
```

```
pid_t            sempid
    Process ID of last operation.
```

```
unsigned short int semncnt
    Number of processes waiting for semval to become greater than current value.
```

```
unsigned short int semzcnt
    Number of processes waiting for semval to become 0.
```

The structure `sembuf` contains the following members:

```
unsigned short int sem_num
    Semaphore number.
short int        sem_op
    Semaphore operation.
short int        sem_flg
    Operation flags.
```

The following are declared as functions and may also be defined as macros:

```
int  semctl(int semid, int semnum, int cmd, ...);
int  semget(key_t key, int nsems, int semflg);
int  semop(int semid, struct sembuf *sops, size_t nsops);
```

In addition, all of the symbols from `<sys/ipc.h>` will be defined when this header is included.

SEE ALSO

[semctl\(2\)](#), [semget\(2\)](#), [semop\(2\)](#), [types\(5\)](#).

NAME

setjmp - stack environment declarations

SYNOPSIS

```
#include <setjmp.h>
```

DESCRIPTION

The `<setjmp.h>` header contains the type definitions for array types `jmp_buf` and `sigjmp_buf`.

The following are declared as functions and may also be defined as macros:

```
void longjmp(jmp_buf env, int val);
void siglongjmp(sigjmp_buf env, int val);
void _longjmp(jmp_buf env, int val);
```

Each of the following may be declared as a function, or defined as a macro, or both:

```
int  setjmp(jmp_buf env);
int  sigsetjmp(sigjmp_buf env, int savemask);
int  _setjmp(jmp_buf env);
```

SEE ALSO

`setjmp(3C)`, `setjmp(3C-ucb)`, `siglongjmp(3C)`, `sigsetjmp(3C)`.

NAME

shm - shared memory facility

SYNOPSIS

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

DESCRIPTION

The `<sys/shm.h>` header defines the following symbolic constants and structure:

Symbolic constants:

`SHM_RDONLY`

Attach read-only (else read-write).

`SHMLBA`

Segment low boundary address multiple.

`SHM_RND`

Round attach address to `SHMLBA`.

The following data types are defined through `typedef`:

`shmatt_t`

Unsigned integer used for the number of current attaches that must be able to store values at least as large as a type `unsigned short`.

The structure `shmid_ds` contains the following members:

`struct ipc_perm shm_perm`

Operation permission structure.

`size_t shm_segsz`

Size of segment in bytes.

`pid_t shm_lpid`

Process ID of last shared memory operation.

`pid_t shm_cpid`

Process ID of creator.

`shmatt_t shm_nattch`

Number of current attaches.

`time_t shm_atime`

Time of last `shmat()`.

`time_t shm_dtime`

Time of last `shmdt()`.

`time_t shm_ctime`

Time of last change by `shmctl()`.

The `pid_t`, `time_t`, `key_t` and `size_t` types are defined as described in `<sys/types.h>` [see [types\(5\)](#)]. The following are declared as functions and may also be defined as macros:

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

```
int shmdt(const void *shmaddr);
```

```
int shmget(key_t key, size_t size, int shmflg);
```

In addition, all of the symbols from `<sys/ipc.h>` will be defined when this header is included.

SEE ALSO

[shmat\(2\)](#), [shmctl\(2\)](#), [shmdt\(2\)](#), [shmget\(2\)](#), [types\(5\)](#).

NAME

siginfo - signal generation information

SYNOPSIS

```
#include <siginfo.h>
```

DESCRIPTION

If a process is catching a signal, it may request information that tells why the system generated that signal [see [sigaction\(2\)](#)]. If a process is monitoring its children, it may receive information that tells why a child changed state [see [waitid\(2\)](#)]. In either case, the system returns the information in a structure of type `siginfo_t`, which includes the following information:

```
int si_signo /* signal number */
int si_errno /* error number */
int si_code  /* signal code */
```

`si_signo` contains the system-generated signal number. (For the [waitid\(2\)](#) function, `si_signo` is always `SIGCHLD`.)

If `si_errno` is non-zero, it contains an error number associated with this signal, as defined in `errno.h`.

`si_code` contains a code identifying the cause of the signal. If the value of `si_code` is less than or equal to 0, then the signal was generated by a user process [see [kill\(2\)](#) and [sigsend\(2\)](#)] and the `siginfo` structure contains the following additional information:

```
pid_t si_pid /* sending process ID */
uid_t si_uid /* sending user ID */
```

Otherwise, `si_code` contains a signal-specific reason why the signal was generated, as follows:

Signal	Code	Reason
SIGILL	ILL_ILLOPC	illegal opcode
	ILL_ILLOPN	illegal operand
	ILL_ILLADR	illegal addressing mode
	ILL_ILLTRP	illegal trap
	ILL_PRVOPC	privileged opcode
	ILL_PRVREG	privileged register
	ILL_COPROC	coprocessor error
	ILL_BADSTK	internal stack error
SIGFPE	FPE_INTDIV	integer divide by zero
	FPE_INTOVF	integer overflow
	FPE_FLTDIV	floating point divide by zero
	FPE_FLTOVF	floating point overflow
	FPE_FLTUND	floating point underflow
	FPE_FLTRES	floating point inexact result
	FPE_FLTINV	invalid floating point operation
	FPE_FLTSUB	subscript out of range
SIGSEGV	SEGV_MAPERR	address not mapped to object

V	SEGV_ACCERR	invalid permissions for mapped object
SIGBUS	BUS_ADRALN	invalid address alignment
	BUS_ADRERR	non-existent physical address
	BUS_OBJERR	object specific hardware error
SIGTRAP	TRAP_BRKPT	process breakpoint
	TRAP_TRACE	process trace trap
SIGCHLD	CLD_EXITED	child has exited
	CLD_KILLED	child was killed
	CLD_DUMPED	child terminated abnormally
	CLD_TRAPPED	traced child has trapped
	CLD_STOPPED	child has stopped
	CLD_CONTINUED	stopped child had continued
SIGPOLL	POLL_IN	data input available
	POLL_OUT	output buffers available
	POLL_MSG	input message available
	POLL_ERR	I/O error
	POLL_PRI	high priority input available
	POLL_HUP	device disconnected

In addition, the following signal-dependent information is available for kernel-generated signals:

Signal	Field	Value
SIGILL	caddr_t si_addr	address of faulting instruction
SIGFPE		
SIGSEGV	caddr_t si_addr	address of faulting memory
SIGBUS		reference
SIGCHLD	pid_t si_pid	child process ID
	int si_status	exit value or signal
SIGPOLL	long si_band	band event for POLL_IN, POLL_OUT, or POLL_MSG

NOTES

For SIGCHLD signals, if `si_code` is equal to `CLD_EXITED`, then `si_status` is equal to the exit value of the process; otherwise, it is equal to the signal that caused the process to change state. For some implementations, the exact value of `si_addr` may not be available; in that case, `si_addr` is guaranteed to be on the same page as the faulting instruction or memory reference.

SEE ALSO

```
sigaction(2), waitid(2), signal(5).
```

NAME

signal - base signals

SYNOPSIS

```
#include <signal.h>
```

DESCRIPTION

A signal is an asynchronous notification of an event. A signal is said to be generated for (or sent to) a process when the event associated with that signal first occurs. Examples of such events include hardware faults, timer expiration and terminal activity, as well as the invocation of the `kill()` or `sigsend()` system calls. In some circumstances, the same event generates signals for multiple processes. A process may request a detailed notification of the source of the signal and the reason why it was generated [see [siginfo\(5\)](#)].

Each process may specify a system action to be taken in response to each signal sent to it, called the signal's disposition. The set of system signal actions for a process is initialized from that of its parent. Once an action is installed for a specific signal, it usually remains installed until another disposition is explicitly requested by a call to either `sigaction()`, `signal()` or `sigset()`, or until the process `execs` [see [sigaction\(2\)](#) and [signal\(2\)](#)]. When a process `exec`, all signals whose disposition has been set to catch the signal will be set to `SIG_DFL`. Alternatively, a process may request that the system automatically reset the disposition of a signal to `SIG_DFL` after it has been caught [see [sigaction\(2\)](#) and [signal\(2\)](#)].

A signal is said to be delivered to a process when the appropriate action for the process and signal is taken. During the time between the generation of a signal and its delivery, the signal is said to be pending [see [sigpending\(2\)](#)]. Ordinarily, this interval cannot be detected by an application. However, a signal can be blocked from delivery to a process [see [signal\(2\)](#) and [sigprocmask\(2\)](#)]. If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the process, the signal remains pending until either it is unblocked or the signal's disposition requests that the signal be ignored. If the signal disposition of a blocked signal requests that the signal be ignored, and if that signal is generated for the process, the signal is discarded immediately upon generation.

Each process has a signal mask that defines the set of signals currently blocked from delivery to it [see [sigprocmask\(2\)](#)]. The signal mask for a process is initialized from that of its parent.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated.

`signal.h` declares constants referring to system signals:

Name	Value	Default	Event
SIGHUP	1	Exit	Hangup [see termio(7)]
SIGINT	2	Exit	Interrupt [see termio(7)]
SIGQUIT	3	Core	Quit [see termio(7)]
SIGILL	4	Core	Illegal instruction
SIGTRAP	5	Core	Trace/breakpoint trap
SIGIOT	6	Core	Abort
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation trap
SIGFPE	8	Core	Arithmetic exception
SIGKILL	9	Exit	Killed

SIGBUS	10	Core	Bus error
SIGSEGV	11	Core	Segmentation fault
SIGSYS	12	Core	Bad system call
SIGPIPE	13	Exit	Broken pipe
SIGALRM	14	Exit	Alarm clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User signal 1
SIGUSR2	17	Exit	User signal 2
SIGCHLD	18	Ignore	Child status
SIGPWR	19	Ignore	Power fail/restart
SIGWINCH	20	Ignore	Window size change
SIGURG	21	Ignore	Urgent socket condition
SIGIO	22	Ignore	Socket I/O possible (SIGPOLL alias)
SIGPOLL	22	Exit	Pollable event occurred
SIGSTOP	23	Stop	Stopped (signal)
SIGTSTP	24	Stop	Stopped (user) [see termio(7)]
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input) [see termio(7)]
SIGTTOU	27	Stop	Stopped (tty output) [see termio(7)]
SIGVTALRM	28	Exit	Virtual timer expired
SIGPROF	29	Exit	Profiling timer expired
SIGXCPU	30	Core	CPU time limit exceeded [see getrlimit(2)]
SIGXFSZ	31	Core	File size limit exceeded [see getrlimit(2)]

Using the `signal()`, `sigset()` or `sigaction()` system call, a process may specify one of three dispositions for a signal: take the default action for the signal, ignore the signal, or catch the signal.

`signal.h` contains the definitions of the following symbolic constants, each of which expands to an expression of the type `void(*) (int)`, and whose value matches no declarable function:

`SIG_DFL`

Default signal handling.

`SIG_ERR`

Return value from `signal()` in case of error.

`SIG_IGN`

Signal is ignored.

In addition, the following data types are defined in `signal.h`:

`sig_atomic_t`

Integral type of an object that can be accessed even in the presence of an asynchronous interrupt.

`sigset_t`

Integral or structure type of an object used to represent sets of signals.

`pid_t` As described in `<sys/types.h>` [see [types\(5\)](#)].

`signal.h` provides the declaration of `struct sigaction` including at least the following elements.

Only valid if generated with "c89":

```
void(*sa_handler)(int)
```

Action on receipt of a signal

```
sigset_t sa_mask
```

Set of signals to be blocked during execution of a signal handling function

```
int sa_flags
```

Special flags

```
void (*)(int, siginfo_t *, void *) sa_sigaction
```

Pointer to signal handler function or one of the macros `SIG_IGN` or `SIG_DFL`.

Only valid if generated with "cc":

```
void (*sa_handler)();
```

```
sigset_t sa_mask;
```

```
int sa_flags;
```

The storage occupied by `sa_handler` and `sa_sigaction` may overlap, and a portable program must not use both simultaneously.

The following constants are declared in `signal.h`:

`SA_NOCLDSTOP`

Do not generate `SIGCHLD` when child processes stop.

`SIG_BLOCK`

The resulting set is the union of the current set and the complement of the signal set pointed to by the argument *set*.

`SIG_UNBLOCK`

The resulting set is the intersection of the current set and the complement of the signal set pointed to by the argument *set*.

`SIG_SETMASK`

The resulting set is the signal set pointed to by *set*.

`SA_ONSTACK`

Causes signal delivery to occur on an alternate stack.

`SA_RESETHAND`

Causes signal dispositions to be set to `SIG_DFL` on entry to signal handlers.

`SA_RESTART`

Causes certain functions to become restartable.

`SA_SIGINFO`

Causes extra information to be passed to signal handlers at the time of receipt of a signal.

`SA_NOCLDWAIT`

Causes implementations not to create zombie processes on child death.

`SA_NODEFER`

Causes signal not to be automatically blocked on entry to signal handler.

`SS_ONSTACK`

Process is executing on an alternate signal stack.

`SS_DISABLE`

Alternate signal stack is disabled.

`MINSIGSTKSZ`

Minimum stack size for a signal handler.

`SIGSTKSZ`

Default size in bytes for the alternate signal stack.

The `ucontext_t` structure is defined through typedef as described in `<ucontext.h>` [see [ucontext\(5\)](#)].

The `signal.h` header defines the `stack_t` type as a structure that includes at least the following members:

```
void *ss_sp
```


Stack base or pointer.

size_t ss_size
Stack size.

int ss_flags
Flags.

The `signal.h` header defines the `sigstack` structure that includes at least the following members:

int ss_onstack
Non-zero when signal stack is in use.

void *ss_sp
Signal stack pointer.

The `signal.h` header defines the `siginfo_t` type as a structure that includes at least the following members:

int si_signo
Signal number.

int si_errno
if non-zero, an `errno` value associated with this signal, as defined in `<errno.h>`

int si_code
Signal code.

pid_t si_pid
Sending process ID.

uid_t si_uid
Real user ID of sending process.

void *si_addr
Address of faulting instruction.

int si_status
Exit value or signal.

long si_band
Band event for `SIGPOLL`.

The macros specified in the **Code** column of the following table are defined for use as values of `si_code` that are signal-specific reasons why the signal was generated.

Signal	Code	Reason
SIGILL	ILL_ILLOPC	illegal opcode
	ILL_ILLOPN	illegal operand
	ILL_ILLADR	illegal addressing mode
	ILL_ILLTRP	illegal trap
	ILL_PRVOPC	privileged opcode
	ILL_PRVREG	privileged register
	ILL_COPROC	coprocessor error
	ILL_BADSTK	internal stack error
SIGFPE	FPE_INTDIV	integer divide by zero
	FPE_INTOVF	integer overflow
	FPE_FLTDIV	floating point divide by zero
	FPE_FLTOVF	floating point overflow
	FPE_FLTUND	floating point underflow

	FPE_FLTRES	floating point inexact result
	FPE_FLTINV	invalid floating point operation
	FPE_FLTSUB	subscript out of range
SIGSEGV	SEGV_MAPERR	address not mapped to object
	SEGV_ACCERR	invalid permissions for mapped object
SIGBUS	BUS_ADRALN	invalid address alignment
	BUS_ADRERR	non-existent physical address
	BUS_OBJERR	object specific hardware error
SIGTRAP	TRAP_BRKPT	process breakpoint
	TRAP_TRACE	process trace trap
SIGCHLD	CLD_EXITED	child has exited
	CLD_KILLED	child has terminated abnormally and did not create a core file
	CLD_DUMPED	child has terminated abnormally and created a core file
	CLD_KILLED	child was killed
	CLD_DUMPED	child has terminated abnormally
	CLD_TRAPPED	traced child has trapped
	CLD_STOPPED	child has stopped
	CLD_CONTINUE	stopped child has continued
SIGPOLL	POLL_IN	data input available
	POLL_OUT	output buffers available
	POLL_MSG	input message available
	POLL_ERR	I/O error
	POLL_PRI	high priority input available
	POLL_HUP	device disconnected

Implementations may support additional `si_code` values not included in this list, may generate values included in this list under circumstances other than those described in this list, and may contain extensions or limitations that prevent some values from being generated. Implementations will not generate a different value from the ones described in this list for circumstances described in this list.

In addition, the following signal-specific information will be available:

Signal	Member	Value
SIGILL	<code>void *</code>	address of faulting instruction
	<code>si_addr</code>	
SIGFPE		
SIGSEGV	<code>void *</code>	address of faulting memory reference
	<code>si_addr</code>	
SIGBUS		
SIGCHLD	<code>pid_t si_pid</code>	child process ID
	<code>int si_status</code>	exit value or signal

	<code>uid_t si_uid</code>	real user ID of the process that sent the signal
<code>SIGPOL</code>	<code>long si_band</code>	band event for <code>POLL_IN</code> , <code>POLL_OUT</code> , or <code>POLL_MSG</code>
<code>L</code>		

For some implementations, the value of `si_addr` may be inaccurate.

The following functions are declared in `signal.h` and may also be defined as macros:

```
void (*bsd_signal(int sig, void (*func)(int)))(int);
int kill(pid_t pid, int sig);
int killpg(pid_t pgrp, int sig);
int pthread_kill(pthread_t thread, int sig);
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
int raise(int sig);
int sigaction(int sig, const struct
sigaction *act,
int sigaction(int sig, const struct sigaction *act,
                struct sigaction *oact);
int sigaddset(sigset_t *set, int signo);
int sigaltstack(const stack_t *ss, stack_t *oss);
int sigdelset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int
sighold(int sig);
int sigignore(int sig);
int siginterrupt(int sig, int flag);
int sigismember(const sigset_t *set, int);
void (*signal(int sig, void (*func)(int)))(int);
int sigpause(int sig);
int sigpending(sigset_t *set);
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int sigrelse(int sig);
void *sigset(int sig,
void (*disp)(int))(int);
int sigstack(struct sigstack *ss, struct sigstack *oss);
int sigsuspend(const sigset_t *sigmask);
int sigwait(const sigset_t *set, int *sig);
```

Default Action

A disposition of `SIG_DFL` specifies the default action. The default action for each signal is listed in the table above and is selected from the following:

- Exit** When it gets the signal, the receiving process is to be terminated with all the consequences outlined in `exit(2)`.
- Core** When it gets the signal, the receiving process is to be terminated with all the consequences outlined in `exit(2)`. In addition, a core image of the process is constructed in the current working directory.
- Stop** When it gets the signal, the receiving process is to stop.
- Ignore** When it gets the signal, the receiving process is to ignore it. This is identical to setting the disposition to `SIG_IGN`.

Catch Signal

A disposition that is a function address specifies that, when it gets the signal, the receiving process is to

execute the signal handler at the specified address. Normally, the signal handler is passed the signal number as its only argument; if the disposition was set with the `sigaction()` function however, additional arguments may be requested [see `sigaction(2)`]. When the signal handler returns, the receiving process resumes execution at the point it was interrupted, unless the signal handler makes other arrangements. If an invalid function address is specified, results are undefined.

If the disposition has been set with the `sigset()` or `sigaction()` function, the signal is automatically blocked by the system while the signal catcher is executing. If a `longjmp()` [see `setjmp(3C)`] is used to leave the signal catcher, then the signal must be explicitly unblocked by the user [see `signal(2)` and `sigprocmask(2)`].

If execution of the signal handler interrupts a blocked system call, the handler is executed and the interrupted system call returns a `-1` to the calling process with `errno` set to `EINTR`. However, if the `SA_RESTART` flag is set the system call will be transparently restarted.

NOTES

The dispositions of the `SIGKILL` and `SIGSTOP` signals cannot be altered from their default values. The system generates an error if this is attempted.

The `SIGKILL` and `SIGSTOP` signals cannot be blocked. The system silently enforces this restriction.

Whenever a process receives a `SIGCONT` signal, regardless of its disposition, any pending `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, and `SIGTTOU` signals is discarded. In addition, if the process was stopped, it is continued.

`SIGPOLL` is issued when a file descriptor corresponding to a STREAMS file has a "selectable" event pending. A process must specifically request that this signal be sent using the `I_SETSIG` `ioctl` call. Otherwise, the process will never receive `SIGPOLL`.

If the disposition of the `SIGCHLD` signal has been set with `signal()` or `sigset()`, or with `sigaction()` and the `SA_NOCLDSTOP` flag has been specified, it will only be sent to the calling process when its children exit; otherwise, it will also be sent if the calling process' children were stopped or continued due to job control.

The name `SIGCLD` is also defined in this header file and identifies the same signal as `SIGCHLD`. `SIGCLD` is provided for backward compatibility, new applications should use `SIGCHLD`.

The disposition of signals that are inherited as `SIG_IGN` should not be changed.

SEE ALSO

`alarm(2)`, `bsd_signal(2)`, `exit(2)`, `getrlimit(2)`, `intro_prm2(2)`, `ioctl(2)`, `kill(2)`, `pause(2)`, `sigaction(2)`, `sigaltstack(2)`, `signal(2)`, `sigprocmask(2)`, `sigsend(2)`, `sigsuspend(2)`, `wait(2)`, `waitid(2)`, `sigsetops(3C)`, `errno(5)`, `siginfo(5)`, `types(5)`, `ucontext(5)`.

NAME

`stat` - data returned by the `stat()` function

SYNOPSIS

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

DESCRIPTION

The `<sys/stat.h>` header defines the structure of the data returned by the functions `fstat()`, `lstat()`, and `stat()`.

The structure `stat` contains at least the following members:

<code>dev_t</code>	<code>st_dev</code>	ID of device containing file.
<code>ino_t</code>	<code>st_ino</code>	File serial number.
<code>mode_t</code>	<code>st_mode</code>	Mode of file (see below).
<code>nlink_t</code>	<code>st_nlink</code>	Number of links to the file.
<code>uid_t</code>	<code>st_uid</code>	User ID of file.
<code>gid_t</code>	<code>st_gid</code>	Group ID of file.
<code>dev_t</code>	<code>st_rdev</code>	Device ID (if file is character or block special).
<code>off_t</code>	<code>st_size</code>	File size in bytes (if file is a regular file).
<code>time_t</code>	<code>st_atime</code>	Time of last access.
<code>time_t</code>	<code>st_mtime</code>	Time of last data modification.
<code>time_t</code>	<code>st_ctime</code>	Time of last status change.
<code>blksize_t</code>	<code>st_blksize</code>	A file system-specific preferred I/O block size for this object. In some file system types, this may vary from file to file.
<code>blkcnt_t</code>	<code>st_blocks</code>	Number of blocks of a file system-specific size allocated for this object.

File serial number and device ID taken together uniquely identify the file within the system. The `dev_t`, `ino_t`, `mode_t`, `nlink_t`, `uid_t`, `gid_t`, `off_t` and `time_t` types are defined as described in `<sys/types.h>` [see [types\(5\)](#)]. Times are given in seconds since the Epoch.

The following symbolic names for the values of `st_mode` are also defined:

File type:

<code>S_IFMT</code>	type of file
<code>S_IFBLK</code>	block special
<code>S_IFCHR</code>	character special
<code>S_IFIFO</code>	FIFO special
<code>S_IFREG</code>	regular
<code>S_IFDIR</code>	directory
<code>S_IFLNK</code>	symbolic link
<code>S_IFSOCK</code>	socket

File mode bits:

<code>S_IRWXU</code>	read, write, execute/search by owner
<code>S_IRUSR</code>	read permission, owner
<code>S_IWUSR</code>	write permission, owner
<code>S_IXUSR</code>	execute/search permission, owner
<code>S_IRWXG</code>	read, write, execute/search by group
<code>S_IRGRP</code>	read permission, group
<code>S_IWGRP</code>	write permission, group
<code>S_IXGRP</code>	execute/search permission, group
<code>S_IRWXO</code>	read, write, execute/search by others
<code>S_IROTH</code>	read permission, others
<code>S_IWOTH</code>	write permission, others
<code>S_IXOTH</code>	execute/search permission, others
<code>S_ISUID</code>	set-user-ID on execution
<code>S_ISGID</code>	set-group-ID on execution
<code>S_ISVTX</code>	on directories, restricted deletion flag
<code>S_IAMB</code>	access mode bits
<code>S_ENFMT</code>	record locking enforcement flag

The bits defined by `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, `S_IRGRP`, `S_IWGRP`, `S_IXGRP`, `S_IROTH`, `S_IWOTH`, `S_IXOTH`, `S_ISUID`, `S_ISGID` and `S_ISVTX` are unique.

`S_IRWXU` is the bitwise OR of `S_IRUSR`, `S_IWUSR` and `S_IXUSR`.

`S_IRWXG` is the bitwise OR of `S_IRGRP`, `S_IWGRP` and `S_IXGRP`.

`S_IRWXO` is the bitwise OR of `S_IROTH`, `S_IWOTH` and `S_IXOTH`.

Implementations may OR other implementation-dependent bits into `S_IRWXU`, `S_IRWXG` and `S_IRWXO`, but they will not overlap any of the other bits defined in this description. The *file permission bits* are defined to be those corresponding to the bitwise inclusive OR of `S_IRWXU`, `S_IRWXG` and `S_IRWXO`.

The following macros are for POSIX conformance. They will test whether a file is of the specified type. The value *m* supplied to the macros is the value of `st_mode` from a `stat` structure. The macro evaluates to a non-zero value if the test is true, 0 if the test is false.

<code>S_ISBLK(m)</code>	Test for a block special file.
<code>S_ISCHR(m)</code>	Test for a character special file.
<code>S_ISDIR(m)</code>	Test for a directory.
<code>S_ISFIFO(m)</code>	Test for a pipe or FIFO special file.
<code>S_ISREG(m)</code>	Test for a regular file.
<code>S_ISLNK(m)</code>	Test for a symbolic link.
<code>S_ISSOCK(m)</code>	

Test for a socket.

The following are declared as functions and may also be defined as macros:

```
int  chmod(const char *path, mode_t mode);  
int  fchmod(int fildes, mode_t mode);  
int  fstat(int fildes, struct stat *buf);  
int  lstat(const char *path, struct stat *buf);  
int  mkdir(const char *path, mode_t mode);  
int  mkfifo(const char *path, mode_t mode);  
int  mknod(const char *path, mode_t mode, dev_t dev);  
int  stat(const char *path,  
struct stat *buf);  
mode_t umask(mode_t cmask);
```

The `stat64` structure is defined in the same way as the `stat` structure with the exception of the following members:

```
ino64_t st_ino  
    File serial number.
```

```
off64_t st_size  
    File size in bytes.
```

```
blkcnt64_t st_blocks  
    Number of blocks allocated for this object.
```

The following are declared as functions and may also be defined as macros:

```
int  fstat64(int fildes, struct stat64 *buf);  
int  lstat64(const char *, struct stat64 *buf);  
int  stat64(const char *, struct stat64 *buf);
```

SEE ALSO

`chmod(2)`, `fchmod(2)`, `fstat(2)`, `lstat(2)`, `mkdir(2)`, `mknod(2)`, `stat(2)`, `umask(2)`, `mkfifo(3C)`, `lfs(5)`, `types(5)`.

NAME

`statvfs` - file system information structure

SYNOPSIS

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

DESCRIPTION

The `<sys/statvfs.h>` header defines the `statvfs` structure that includes at least the following members:

<code>ulong_t</code>	<code>f_bsize</code>	File system block size.
<code>ulong_t</code>	<code>f_frsize</code>	Fundamental file system block size.
<code>fsblkcnt_t</code>	<code>f_blocks</code>	Total number of blocks on file system in units of <code>f_frsize</code> .
<code>fsblkcnt_t</code>	<code>f_bfree</code>	Total number of free blocks.
<code>fsblkcnt_t</code>	<code>f_bavail</code>	Number of free blocks available to non-privileged process.
<code>fsfilcnt_t</code>	<code>f_files</code>	Total number of file serial numbers.
<code>fsfilcnt_t</code>	<code>f_ffree</code>	Total number of free file serial numbers.
<code>fsfilcnt_t</code>	<code>f_favail</code>	Number of file serial numbers available to non-privileged process.
<code>ulong_t</code>	<code>f_fsid</code>	File system ID.
<code>ulong_t</code>	<code>f_flag</code>	Bit mask of <code>f_flag</code> values.
<code>ulong_t</code>	<code>f_namemax</code>	Maximum file name length.

The following flags for the `f_flag` member are defined:

`ST_RDONLY`
Read-only file system.

`ST_NOSUID`
Does not support `setuid/setgid` semantics.

The header `<sys/statvfs.h>` declares the following functions which may also be defined as macros:

```
int statvfs(const char *path, struct statvfs *buf);  
int fstatvfs(int fildes, struct statvfs *buf);
```

The `statvfs64` structure is defined in the same way as the `statvfs` structure with the exception of the following members:

<code>fsblkcnt64_t</code>	<code>f_blocks</code>	Total number of blocks in the file system in units of <code>f_frsize</code> .
<code>fsblkcnt64_t</code>	<code>f_bfree</code>	Total number of free blocks.
<code>fsblkcnt64_t</code>	<code>f_bavail</code>	Number of free blocks available to non-privileged process.
<code>fsfilcnt64_t</code>	<code>f_files</code>	Total number of file serial numbers.
<code>fsfilcnt64_t</code>	<code>f_ffree</code>	Total number of free file serial numbers.
<code>fsfilcnt64_t</code>	<code>f_favail</code>	Number of free file serial numbers available to non-privileged process.

The following are declared as functions and may also be defined as macros:

```
int statvfs64(const char *path, struct statvfs64 *buf) ;  
int fstatvfs64(int fildes, struct statvfs64 *buf) ;
```


SEE ALSO

`fstatvfs(2)`, `statvfs(2)`, `lfs(5)`.

NAME

stdarg - handle variable argument list

SYNOPSIS

```
#include <stdarg.h>
va_list pvar;
void va_start(va_list pvar, parmN);
type va_arg(va_list pvar, type);
void va_end(va_list pvar);
```

DESCRIPTION

stdarg.h contains macros that permit the development of portable procedures that accept variable numbers of arguments of variable types. Routines that have variable argument lists (such as printf()) but do not use these macros are inherently non-portable, as different machines use different argument-passing conventions.

va_list is a type defined for the variable used to traverse the list.

The va_start() macro is invoked before any access to the unnamed arguments and initializes pvar for subsequent use by va_arg() and va_end(). The parameter parmN is the identifier of the rightmost parameter in the variable parameter list in the function definition. If this parameter is declared with the register storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

The parameter parmN is required under strict ANSI C compilation. In other compilation modes, parmN need not be supplied and the second parameter to the va_start() macro can be left empty (e.g.

va_start(pvar,);). This allows for routines that contain no parameters before the ... in the variable parameter list.

The va_arg() macro expands to an expression that has the type and value of the next argument in the call. The parameter pvar should have been previously initialized by va_start(). Each invocation of va_arg() modifies pvar so that the values of successive arguments are returned in turn. The parameter type is the type name of the next argument to be returned. The type name must be specified in such a way so that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to type. If there is no actual next argument, or if type is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

The va_end() macro is used to clean up.

Multiple traversals, each bracketed by va_start() and va_end(), are possible.

EXAMPLE

This example gathers into an array a list of arguments that are pointers to strings (but not more than MAXARGS arguments) with function f1, then passes the array as a single argument to function f2. The number of pointers is specified by the first argument to f1.

```
#include <stdarg.h>
#define MAXARGS 31
void f1(int n_ptrs, ...)
{
    va_list ap;
    char *array[MAXARGS];
    int ptr_no = 0;
    if (n_ptrs > MAXARGS)
        n_ptrs = MAXARGS;
    va_start(ap, n_ptrs);
    while (ptr_no < n_ptrs)
```

```
        array[ptr_no++] = va_arg(ap, char*);
    va_end(ap);
    f2(n_ptrs, array);
}
```

Each call to `f1` shall have visible the definition of the function or a declaration such as

```
void f1(int, ...)
```

NOTES

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame. For example, `execl()` is passed a null pointer to signal the end of the list. `printf()` can tell how many arguments there are by the format. It is non-portable to specify a second argument of `char`, `short`, or `float` to `va_arg()`, because arguments seen by the called function are not `char`, `short`, or `float`. C converts `char` and `short` arguments to `int` and converts `float` arguments to `double` before passing them to a function.

SEE ALSO

[vprintf\(3S\)](#).

NAME

stddef - standard type definitions

SYNOPSIS

```
#include <stddef.h>
```

DESCRIPTION

The `<stddef.h>` header defines the following:

`NULL` Null pointer constant.

`offsetof (type, member-designator)`

Integral constant expression of type `size_t`, the value of which is the offset in bytes to the structure member (*member-designator*), from the beginning of its structure (*type*).

The `<stddef.h>` header defines through `typedef`:

`ptrdiff_t`

Signed integral type of the result of subtracting two pointers.

`wchar_t`

Integral type whose range of values can represent distinct wide character codes for all members of the largest character set specified among the locales supported by the compilation environment: the null character has the code value 0 and each member of the Portable Character Set has a code value equal to its value when used as the lone character in an integer character constant.

`size_t`

Unsigned integral type of the result of the `sizeof` operator.

SEE ALSO

[types\(5\)](#), [wchar\(5\)](#).

NAME

`stdio` - standard buffered input/output

SYNOPSIS

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

DESCRIPTION

The `<stdio.h>` header defines the following macro names as positive integral constant expressions:

`BUFSIZ`

Size of `<stdio.h>` buffers.

`FILENAME_MAX`

Maximum size in bytes of the longest filename string that the implementation guarantees can be opened.

`FOPEN_MAX`

Number of streams which the implementation guarantees can be open simultaneously. The value will be at least eight.

`_IOFBF`

Input/output fully buffered.

`_IOLBF`

Input/output line buffered.

`_IONBF`

Input/output unbuffered.

`L_ctermid`

Maximum size of character array to hold `ctermid()` output.

`L_cuserid`

Maximum size of character array to hold `cuserid()` output.

`L_tmpnam`

Maximum size of character array to hold `tmpnam()` output.

`SEEK_CUR`

Seek relative to current position.

`SEEK_END`

Seek relative to end-of-file.

`SEEK_SET`

Seek relative to start-of-file.

`TMP_MAX`

Minimum number of unique filenames generated by `tmpnam()`. Maximum number of times an application can call `tmpnam()` reliably. The value of `TMP_MAX` will be at least 10,000.

The following macro name is defined as a negative integral constant expression:

`EOF` End-of-file return value.

The following macro name is defined as a null pointer constant:

`NULL` Null pointer.

The following macro name is defined as a string constant:

`P_tmpdir`

Default directory prefix for `tempnam()`.

The following macro names are defined as expressions of type pointer to `FILE`:

`stderr`

Standard error output stream.

`stdin`

Standard input stream.

`stdout`

Standard output stream.

The following data types are defined through `typedef`:

`FILE` A structure containing information about a file.

`fpos_t`

Type containing all information needed to specify uniquely every position within a file.

`va_list`

As described in `<stdarg.h>` [see [stdarg\(5\)](#)].

`size_t`

As described in `<stddef.h>` [see [stddef\(5\)](#)].

The following are declared as functions and may also be defined as macros:

```
void      clearerr(FILE *stream);
char      *ctermid(char *s);
char      *cuserid(char *s);
int        fclose(FILE *stream);
FILE      *fdopen(int fildes, const char *mode);
int        feof(FILE *stream);
int        ferror(FILE *stream);
int        fflush(FILE *stream);
int        fgetc(FILE *stream);
int        fgetpos(FILE *stream, fpos_t *pos);
char      *fgets(char *s, int n, FILE *stream);
int        fileno(FILE *stream);
void       flockfile(FILE *file);
FILE      *fopen(const char *filename, const char *mode);
int        fprintf(FILE *stream, const char *format, ...);
int        fputc(int c, FILE *stream);
int        fputs(const char *s, FILE *stream);
size_t     fread(void *ptr, size_t size, size_t nitems,
                FILE *stream);
FILE      *freopen(const char *filename, const char *mode,
                FILE *stream);
int        fscanf(FILE *stream, const char *format, ...);
int        fseek(FILE *stream, long int offset, int whence);
int        fseeko(FILE *stream, off_t offset, int whence);
int        fsetpos(FILE *stream, const fpos_t *pos);
long int   ftell(FILE *stream);
off_t      ftello(FILE *stream);
int        ftrylockfile(FILE *file);
void       funlockfile(FILE *file);
size_t     fwrite(const void *ptr, size_t size, size_t nitems,
                FILE *stream);
int        getc(FILE *stream);
int        getchar(void);
int        getc_unlocked(FILE *stream);
int        getchar_unlocked(void);
int        getopt(int argc, char * const argv[],
                const char *optstring);
```

```

char  *gets(char *s);
int    getw(FILE *stream);
int    pclose(FILE *stream);
void   perror(const char *s);
FILE   *popen(const char *command, const char *type);
int    printf(const char *format, ...);
int    putc(int c, FILE *stream);
int    putchar(int c);
int    putc_unlocked(int c, FILE *stream);
int    putchar_unlocked(int c);
int    puts(const char *s);
int    putw(int w, FILE *stream);
int    remove(const char *path);
int    rename(const char *old, const char *new);
void   rewind(FILE *stream);
int    scanf(const char *format, ...);
void   setbuf(FILE *stream, char *buf);
int    setvbuf(FILE *stream, char *buf, int type, size_t size);
int    snprintf(char *s, size_t n, const char *format, /* args */ ...);
int    sprintf(char *s, const char *format, ...);
int    sscanf(const char *s, const char *format, int ...);
char   *tempnam(const char *dir, const char *pfx);
FILE   *tmpfile(void);
char   *tmpnam(char *s);
int    ungetc(int c, FILE *stream);
int    vfprintf(FILE *stream, const char *format, va_list ap);
int    vprintf(const char *format, va_list ap);
int    vsnprintf(char *s, size_t n, const char *format, va_list ap);
int    vsprintf(char *s, const char *format, va_list ap);

```

The following external variables are defined:

```

extern char *optarg;
extern int  opterr;
extern int  optind;
extern int  optopt;

```

Inclusion of the `<stdio.h>` header may also make visible all symbols from `<stddef.h>`.

The following data type is defined through typedef:

```
fpos64_t
```

Type containing all information needed to specify uniquely every position within a file in which the largest offset can be represented in an object of type `off64_t`.

The following are declared as functions and may also be defined as macros:

```

int      fgetpos64(FILE *stream, fpos64_t *pos);
FILE     *fopen64(const char *filename, const char *mode);
FILE     *freopen64(const char *filename, const char *mode,
                    FILE *stream);
int      fseeko64(FILE *stream, off64_t offset, int whence);
int      fsetpos64(FILE *stream, const fpos64_t *pos);
off64_t  ftello64(FILE *stream);
FILE     *tmpfile64(void);

```

SEE ALSO

[rename\(2\)](#), [fgetpos\(3C\)](#), [fsetpos\(3C\)](#), [perror\(3C\)](#), [remove\(3C\)](#), [ctermid\(3S\)](#), [cuserid\(3S\)](#), [fclose\(3S\)](#), [ferror\(3S\)](#), [fflush\(3S\)](#), [fgetpos\(3S\)](#), [fgetc\(3S\)](#), [fgetwc\(3S\)](#), [fopen\(3S\)](#),

fputc(3S), fputs(3S), fread(3S), fseek(3S), fwrite(3S),getc(3S), gets(3S), gets(3S),
pclose(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S),
setvbuf(3S), system(3S), tempnam(3S), tmpfile(3S), tmpnam(3S), ungetc(3S),
vprintf(3S), lfs(5), types(5).

NAME

`stdlib` - standard library definitions

SYNOPSIS

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

DESCRIPTION

The `<stdlib.h>` header defines the following macro names:

`EXIT_FAILURE`

Unsuccessful termination for `exit()`, evaluates to a non-zero value.

`EXIT_SUCCESS`

Successful termination for `exit()`, evaluates to 0.

`NULL` Null pointer.

`RAND_MAX`

Maximum value returned by `rand()`, at least 32,767.

`MB_CUR_MAX`

Integer expression whose value is the maximum number of bytes in a character specified by the current locale.

The following data types are defined through `typedef`:

`div_t`

Structure type returned by `div()` function.

`ldiv_t`

Structure type returned by `ldiv()` function.

`size_t`

As described in `<stddef.h>` [see [stddef\(5\)](#)].

`wchar_t`

As described in `<stddef.h>` [see [stddef\(5\)](#)].

In addition, the following symbolic names and macros are defined as in `<sys/wait.h>` [see [wait\(5\)](#)], for use in decoding the return value from `system()`:

`WNOHANG`

`WUNTRACED`

`WEXITSTATUS()`

`WIFEXITED()`

`WIFSIGNALED()`

`WIFSTOPPED()`

`WSTOPSIG()`

`WTERMSIG()`

The following are declared as functions and may also be defined as macros:

```
long    a64l(const char *s);
```

```
void    abort(void);
```

```
int     abs(int i);
```

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

```
double  atof(const char *str);
```

```
int     atoi(const char *str);
```

```
long int atol(const char *str);
```

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

```

void    *calloc(size_t nelem, size_t elsize);
div_t    div(int numer, int denom);
double   drand48(void);
double   erand48(unsigned short int xsubi[3]);
char     *ecvt (double value, int ndigit, int *decpt, int *sign);
void     exit(int status);
char     *fcvt (double value, int ndigit, int *decpt, int *sign);
void     free(void *ptr);
char     *gcvt (double value, int ndigit, char *buf);
char     *getenv(const char *name);
int      getsubopt(char **optionp, char *const *tokens, char **valuep);
int      grantpt(int fildev);
char     *initstate(unsigned seed, char *state, size_t size);
long int jrand48 (unsigned short int xsubi[3]);
char     *l64a(long value);
long int labs(long int j);
void     lcong48(unsigned short int param[7]);
ldiv_t   ldiv(long int numer, long int denom);
long int lrand48 (void);
void     *malloc(size_t size);
int      mblen (const char *s, size_t n);
size_t   mbstowcs (wchar_t *pwcs, const char *s, size_t n);
int      mbtowc (wchar_t *pwc, const char *s, size_t n);
char     *mktemp(char *template);
int      mkstemp(char *template);
long int mrand48 (void);
long int nrand48 (unsigned short int xsubi[3]);
char     *ptsname(int fildev);
int      putenv(const char *string);
void     qsort(void *base, size_t nel, size_t width,
               int (*compar)(const void *, const void *));
int      rand(void);
int      rand_r(unsigned int *seed);
long     random(void);
void     *realloc(void *ptr, size_t size);
char     *realpath(const char *file_name, char *resolved_name);
unsigned short int *seed48 (unsigned short int seed18v[3]);
void     setkey(const char *key);
char     *setstate(const char *state);
void     srand(unsigned int seed);
void     srand48(long int seedval);
void     srandom(unsigned seed);
double   strtod(const char *str, char **ptr);
long int strtol(const char *str, char **ptr, int base);
unsigned long int strtoul(const char *str, char **ptr, int base);
int      system(const char *string);
int      ttyslot(void);
int      unlockpt(int fildev);
void     *valloc(size_t size);
size_t   wcstombs(char *s, const wchar_t *pwcs, size_t n);
int      wctomb(char *s, wchar_t wchar);

```

Inclusion of the `stdlib()` header may also make visible all symbols from `<stddef.h>`, `<limits.h>`, `<math.h>` and `<sys/wait.h>`.

SEE ALSO

`exit(2)`, `initstate(3)`, `mkstemp(3)`, `setstate(3)`, `srandom(3)`, `a64l(3C)`, `abort(3C)`, `abs(3C)`, `atexit(3C)`, `atof(3C)`, `atoi(3C)`, `atol(3C)`, `bsearch(3C)`, `calloc(3C)`, `div(3C)`, `drand48(3C)`, `ecvt(3C)`, `erand48(3C)`, `fcvt(3C)`, `free(3C)`, `gcvt(3C)`, `getenv(3C)`, `getsubopt(3C)`, `grantpt(3C)`, `jrand48(3C)`, `l64a(3C)`, `labs(3C)`, `lcong48(3C)`, `ldiv(3C)`, `lrand48(3C)`, `malloc(3C)`, `mblen(3C)`, `mbstowcs(3C)`, `mbtowc(3C)`, `mktemp(3C)`, `mrnd48(3C)`, `nrand48(3C)`, `ptsname(3C)`, `putenv(3C)`, `qsort(3C)`, `rand(3C)`, `realloc(3C)`, `realpath(3C)`, `srand(3C)`, `srand48(3C)`, `strtod(3C)`, `strtol(3C)`, `strtoul(3C)`, `ttyslot(3C)`, `unlockpt(3C)`, `valloc(3C)`, `wcstombs(3C)`, `wctomb(3C)`, `types(5)`.

NAME

string - string operations

SYNOPSIS

```
#include <string.h>
```

DESCRIPTION

The `<string.h>` header defines the following:

`NULL` Null pointer constant.

`size_t`

As described in `<stddef.h>` [see [stddef\(5\)](#)].

The following are declared as functions and may also be defined as macros:

```
void *memcpy(void *s1, const void *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 *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
void *memset(void *s, int c, size_t n);
char *strcat(char *s1, const char *s2);
char *strchr(const char *s, int c);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
char *strcpy(char *s1, const char *s2);
size_t strcspn(const char *s1, const char *s2);
char *strdup(const char *s1);
char *strerror(int errnum);
size_t strlen(const char *s);
char *strncat(char *s1, const char *s2, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *s1, const char *s2, size_t n);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
char *strtok_r(char *s, const char *sep, char **lasts);
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Inclusion of the `<string.h>` header may also make visible all symbols from `<stddef.h>`.

SEE ALSO

[memory\(3C\)](#), [string\(3C\)](#), [types\(5\)](#).

NAME

strings - string operations

SYNOPSIS

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

DESCRIPTION

The following are declared as functions and may also be defined as macros:

```
int  bcmp(const void *s1, const void *s2, size_t n);
void bcopy(const void *s1, void *s2, size_t n);
void bzero(void *s, size_t n);
int  ffs(int i);
char *index(const char *s, int c);
char *rindex(const char *s, int c);
int  strcasecmp(const char *s1, const char *s2);
int  strncasecmp(const char *s1, const char *s2, size_t n);
```

The `size_t` type is defined through `typedef` as described in `<stddef.h>`.

SEE ALSO

`bstring(3)`, `index(3)`, `rindex(3)`, `ffs(3C)`, `strcasecmp(3C)`.

NAME

stropts - STREAMS interface

SYNOPSIS

```
#include <stropts.h>
```

DESCRIPTION

The `<stropts.h>` header defines the `bandinfo` structure that includes at least the following members:

```
unsigned char bi_pri
int          bi_flag
```

The `<stropts.h>` header defines the `strpeek` structure that includes at least the following members:

```
struct strbuf ctlbuf
struct strbuf databuf
t_uscalar_t   flags
```

The `<stropts.h>` header defines the `strbuf` structure that includes at least the following members:

```
int          maxlen
             maximum buffer length
int          len
             length of data
char         *buf
             pointer to buffer
```

The `<stropts.h>` header defines the `strfdinsert` structure that includes at least the following members:

```
struct strbuf ctlbuf
struct strbuf databuf
t_uscalar_t   flags
int           fildes
int           offset
```

The `<stropts.h>` header defines the `strioctl` structure that includes at least the following members:

```
int           ic_cmd
int           ic_timeout
int           ic_len
char          *ic_dp
```

The `<stropts.h>` header defines the `strrecvfd` structure that includes at least the following members:

```
int           fd
uid_t         uid
gid_t         gid
```

The `uid_t` and `gid_t` types are defined through typedef as described in `<sys/types.h>` [see [types\(5\)](#)].

The `<stropts.h>` header defines the `str_list` structure that includes at least the following members:

```
int           sl_nmods
struct str_mlist *sl_modlist
```

The `<stropts.h>` header defines the `str_mlist` structure that includes at least the following member:

```
char          l_name[FMNAMESZ+1]
```

At least the following macros are defined for use as the *request* argument to `ioctl()`:

```
I_PUSH
```

Push STREAMS module onto the top of the current STREAM, just below the STREAM head.

```
I_POP
```

Remove STREAMS module from just below the STREAM head.

I_LOOK

Retrieve the name of the module just below the STREAM head and place it in a character string. At least the following macros are defined for use as the *arg* argument:

FMNAMESZ

The minimum size in bytes of the buffer referred to by the *arg* argument.

I_FLUSH

This request flushes all input and/or output queues, depending on the value of the *arg* argument. At least the following macros are defined for use as the *arg* argument:

FLUSHR

Flush read queues.

FLUSHW

Flush write queues.

FLUSHRW

Flush read and write queues.

I_FLUSHBAND

Flush only band specified.

I_SETSIG

Informs the STREAM head that the process wants the SIGPOLL signal issued [see [signal\(2\)](#) and [signal\(5\)](#)] when a particular event has occurred on the STREAM.

The header `<stropts.h>` defines these possible values for *arg* when I_SETSIG is specified:

S_RDNORM

A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue.

S_RDBAND

A message with a non-zero priority band has arrived at the head of a STREAM head read queue.

S_INPUT

A message, other than a high-priority message, has arrived at the head of a STREAM head read queue.

S_HIPRI

A high-priority message is present on a STREAM head read queue.

S_OUTPUT

The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.

S_WRNORM

Same as S_OUTPUT.

S_WRBAND

The write queue for a non-zero priority band just below the STREAM head is no longer full.

S_MSG

A STREAMS signal message that contains the SIGPOLL signal reaches the front of the STREAM head read queue.

S_ERROR

Notification of an error condition reaches the STREAM head.

S_HANGUP

Notification of a hangup reaches the STREAM head.

S_BANDURG

When used in conjunction with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the STREAM head read queue.

I_GETSIG

Returns the events for which the calling process is currently registered to be sent a `SIGPOLL` signal.

I_FIND

Compares the names of all modules currently present in the STREAM to the name pointed to by *arg*.

I_PEEK

Allows a process to retrieve the information in the first message on the STREAM head read queue without taking the message off the queue. At least the following macros are defined for use as the *arg* argument:

RS_HIPRI

Only look for high-priority messages.

I_SRDOPT

Sets the read mode. At least the following macros are defined for use as the *arg* argument:

RNORM

Byte-STREAM mode, the default.

RMSGD

Message-discard mode.

RMSGN

Message-nondiscard mode.

RPROTNORM

Fail `read()` with `EBADMSG` if a message containing a control part is at the front of the STREAM head read queue.

RPROTDAT

Deliver the control part of a message as data when a process issues a `read()`.

RPROTDIS

Discard the control part of a message, delivering any data part, when a process issues a `read()`.

I_GRDOPT

Returns the current read mode setting.

I_NREAD

Counts the number of data bytes in data blocks in the first message on the STREAM head read queue.

I_FDINSERT

Creates a message from the specified buffer(s), adds information about another STREAM, and sends the message downstream.

I_STR

Constructs an internal STREAMS `ioctl()` message and sends that message downstream.

I_SWROPT

Sets the write mode. At least the following macros are defined for use as the *arg* argument:

SNDZERO

Send a zero-length message downstream when a `write()` of 0 bytes occurs.

I_GWROPT

Returns the current write mode setting.

I_SENDFD

Requests the STREAM associated with *fd* to send a message, containing a file pointer, to the STREAM head at the other end of a STREAMS pipe.

I_RECVFD

Retrieves the file descriptor associated with the message sent by an `I_SENDFD ioctl()` over a STREAMS pipe.

I_LIST

This request allows the process to list all the module names on the STREAM, up to and including the topmost driver name.

I_ATMARK

This request allows the process to see if the current message on the STREAM head read queue is "marked" by some module downstream. At least the following macros are defined for use as the *arg* argument:

ANYMARK

Check if the message is marked.

LASTMARK

Check if the message is the last one marked on the queue.

I_CKBAND

Check if the message of a given priority band exists on the STREAM head read queue.

I_GETBAND

Return the priority band of the first message on the STREAM head read queue.

I_CANPUT

Check if a certain band is writable.

I_SETCLTIME

Allows the process to set the time the STREAM head will delay when a STREAM is closing and there is data on the write queues.

I_GETCLTIME

Returns the close time delay.

I_LINK

Connects two STREAMs.

I_UNLINK

Disconnects the two STREAMs. The header defines at least the following value for *all*:

MUXID_ALL

Unlink all STREAMs linked to the STREAM associated with *fd*.

I_PLINK

Connects two STREAMs with a persistent link.

I_PUNLINK

Disconnects the two STREAMs that were connected with a persistent link.

The following macros are defined for `getmsg()`, `getpmsg()`, `putmsg()` and `putpmsg()`:

MSG_ANY

Receive any message.

MSG_BAND

Receive message from specified band.

MSG_HIPRI

Send/Receive high priority message.

MORECTL

More control information is left in message.

MOREDATA

More data is left in message.

The header `<stropts.h>` may make visible all of the symbols from `<unistd.h>`.

The following are declared as functions in the `<stropts.h>` header and may also be defined as macros:

```
int isastream(int fildes);
```

```
int getmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr,
           int *flagsp);
```

```
int getpmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr,
            int *bandp, int *flagsp);
```

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

```
int putmsg(int fd, const struct strbuf *ctlptr,
```

```
    const struct strbuf *dataptr, int flags);  
int putpmsg(int fd, const struct strbuf *ctlptr,  
    const struct strbuf *dataptr, int band, int flags);  
int fattach(int fildes, const char *path);  
int fdetach(const char *path);
```

SEE ALSO

```
close(2), fcntl(2), getmsg(2), ioctl(2), open(2), pipe(2), poll(2), putmsg(2), read(2),  
signal(2), write(2).
```

NAME

sys_time - time types

SYNOPSIS

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

DESCRIPTION

The `<sys/time.h>` header defines the `timeval` structure that includes at least the following members:

```
time_t      tv_sec
    Seconds since Jan. 1, 1970
suseconds_t tv_usec
    Microseconds
```

The `<sys/time.h>` header defines the `itimerval` structure that includes at least the following members:

```
struct timeval it_interval
    Timer interval
struct timeval it_value
    Current value
```

The `time_t` and `suseconds_t` types are defined as described in `<sys/types.h>` [see [types\(5\)](#)].

The `<sys/time.h>` header defines the `fd_set` type as a structure that includes at least the following member:

```
long fds_bits[]
    Bit mask for open file descriptions
```

The `<sys/time.h>` header defines the following values for the `which` argument of `getitimer()` and `setitimer()`:

```
ITIMER_REAL
    Decrements in real time.
```

```
ITIMER_VIRTUAL
    Decrements in process virtual time (user mode).
```

```
ITIMER_PROF
    Decrements both in process virtual time and when the system is running on behalf of the process (user and system mode).
```

The following macros are defined:

```
void FD_CLR(int fd, fd_set *fdset)
    Clears the bit for the file descriptor fd in the file descriptor set fdset.
```

```
int  FD_ISSET(int fd, fd_set *fdset)
    Returns a non-zero value if the bit for the file descriptor fd is set in the file descriptor set by fdset, and 0 otherwise.
```

```
void FD_SET(int fd, fd_set *fdset)
    Sets the bit for the file descriptor fd in the file descriptor set fdset.
```

```
void FD_ZERO(fd_set *fdset)
    Initializes the file descriptor set fdset to have zero bits for all file descriptors.
```

```
FD_SETSIZE
    Maximum number of file descriptors in an fd_set structure.
```

The following are declared as functions and may also be defined as macros:

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

```
    struct itimerval *ovalue);  
int  gettimeofday(struct timeval *tp, void * tzp);  
int  select(int nfds, fd_set *readfds, fd_set *writefds,  
           fd_set *errorfds, struct timeval *timeout);  
int  utimes(const char *path, const struct timeval times[2]);
```

SEE ALSO

```
time(1), utimes(3), getitimer(3C), gettimeofday(3C), select(3C), setitimer(3C),  
settimeofday(3C).
```

NAME

syslog - definitions for system error logging

SYNOPSIS

```
#include <syslog.h>
```

DESCRIPTION

The `<syslog.h>` header defines the following symbolic constants, zero or more of which may be OR-ed together to form the *logopt* option of `openlog()`:

`LOG_PID`
Log the process ID with each message.

`LOG_CONS`
Log to the system console on error.

`LOG_NDELAY`
Connect to syslog daemon immediately.

`LOG_ODELAY`
Delay open until `syslog()` is called.

`LOG_NOWAIT`
Don't wait for child processes.

The following symbolic constants are defined as possible values of the *facility* argument to `openlog()`:

`LOG_KERN`
Reserved for message generated by the system.

`LOG_USER`
Message generated by a process.

`LOG_MAIL`
Reserved for message generated by mail system.

`LOG_NEWS`
Reserved for message generated by news system.

`LOG_UUCP`
Reserved for message generated by UUCP system.

`LOG_DAEMON`
Reserved for message generated by system daemon.

`LOG_AUTH`
Reserved for message generated by authorization daemon.

`LOG_CRON`
Reserved for message generated by the clock daemon [see [cron\(1M\)](#)].

`LOG_LPR`
Reserved for message generated by printer system.

`LOG_LOCAL0`
Reserved for local use.

`LOG_LOCAL1`
Reserved for local use.

`LOG_LOCAL2`
Reserved for local use.

`LOG_LOCAL3`
Reserved for local use.

`LOG_LOCAL4`

Reserved for local use.

LOG_LOCAL5

Reserved for local use.

LOG_LOCAL6

Reserved for local use.

LOG_LOCAL7

Reserved for local use.

The following are declared as macros for constructing the *maskpri* argument to `setlogmask()`. The following macros expand to an expression of type `int` when the argument *pri* is an expression of type `int`:

LOG_MASK(*pri*)

A mask for priority *pri*.

LOG_UPTO(*pri*)

A mask for all priorities through *pri*.

The following constants are defined as possible values for the *priority* argument of `syslog()`:

LOG_EMERG

A panic condition was reported to all processes.

LOG_ALERT

A condition that should be corrected immediately.

LOG_CRIT

A critical condition.

LOG_ERR

An error message.

LOG_WARNING

A warning message.

LOG_NOTICE

A condition requiring special handling.

LOG_INFO

A general information message.

LOG_DEBUG

A message useful for debugging programs.

The following are declared as functions and may also be defined as macros:

```
void closelog(void);
```

```
void openlog(const char *id, int logopt, int facility);
```

```
int setlogmask(int maskpri);
```

```
void syslog(int priority, const char *format, ...);
```

SEE ALSO

[syslogd\(1M\)](#), [syslog\(3C\)](#).

tar(5) - wstat(5)

NAME

tar - extended tar definitions

SYNOPSIS

```
#include <tar.h>
```

DESCRIPTION

Header block definitions are:

General definitions:

Name	Description	Value
TMAGIC	"ustar"	ustar plus null byte.
TMAGLEN	6	Length of the above.
TVERSIO N	"00"	00 without a null byte.
TVERSLE N	2	Length of the above.

typeflag field definitions:

Name	Description	Value
REGTYPE	'0'	Regular file.
AREGTYPE	'\0'	Regular file.
LNKTYPE	'1'	Link.
SYMTYPE	'2'	Symbolic link.
CHRTYPE	'3'	Character special.
BLKTYPE	'4'	Block special.
DIRTYPE	'5'	Directory.
FIFOTYPE	'6'	FIFO special.
CONTTYPE	'7'	Reserved.

mode field bit definitions (octal):

Name	Description	Value
TSUID	04000	Set UID on execution.
TSGID	02000	Set GID on execution.
TSVTX	01000	On directories, restricted deletion flag.
TUREAD	00400	Read by owner.
TUWRIT E	00200	Write by owner special.
TUEXEC	00100	Execute/search by owner.
TGREAD	00040	Read by group.
TGWRIT E	00020	Write by group.
TGEXEC	00010	Execute/search by group.
TOREAD	00004	Read by other.
TOWRIT E	00002	Write by other.
TOEXEC	00001	Execute/search by other.

NAME

termios - define values for termios

SYNOPSIS

#include <termios.h>

DESCRIPTION

The <termios.h> header contains the definitions used by the terminal I/O interfaces.

The termios Structure

The following data types are defined through typedef:

cc_t Used for terminal special characters.

speed_t Used for terminal baud rates.

tcflag_t Used for terminal modes.

The above types are all unsigned integral types.

The termios structure is defined, and includes at least the following members:

- tcflag_t c_iflag input modes
- tcflag_t c_oflag output modes
- tcflag_t c_cflag control modes
- tcflag_t c_lflag local modes
- cc_t c_cc[NCCS] control chars

A definition is given for:

NCCS Size of the array c_cc for control characters.

The following subscript names for the array c_cc are defined:

Subscript Usage		
Canonical Mode	Non-Canonical Mode	Description
VEOF		EOF character
VEOL		EOL character
VERASE		ERASE character
VINTR	VINTR	INTR character
VKILL		KILL character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character
VSUSP	VSUSP	SUSP character

VTIME

TIME character

Input Modes

The subscript values are unique, except that the `VMIN` and `VTIME` subscripts may have the same values as the `VEOF` and `VEOL` subscripts, respectively. The `c_iflag` field describes the basic terminal input control:

`BRKINT`

Signal interrupt on break.

`ICRNL` Map CR to NL on input.`IGNBRK`

Ignore break condition.

`IGNCR` Ignore CR.`IGNPAR`

Ignore characters with parity errors.

`INLCR` Map NL to CR on input.`INPCK` Enable input parity check.`ISTRIP`

Strip character.

`IUCLC` Map upper case to lower case on input.`IXANY` Enable any character to restart output.`IXOFF` Enable start/stop input control.`IXON` Enable start/stop output control.`PARMRK`

Mark parity errors.

Output Modes

The `c_oflag` field specifies the system treatment of output:

`OPOST` Post-process output.`OLCUC` Map lower-case to upper-case on output.`ONLCR` Map NL to CR-NL on output.`OCRNL` Map CR to NL on output.`ONOCR` No CR output at column 0.`ONLRET`

NL performs CR function.

`OFILL` Use fill characters for delay.`NLDLY` Select newline delays:`NL0` Newline character type 0.`NL1` Newline character type 1.`CRDLY` Select carriage-return delays:`CR0` Carriage-return delay type 0.`CR1` Carriage-return delay type 1.`CR2` Carriage-return delay type 2.`CR3` Carriage-return delay type 3.`TABDLY`

Select horizontal-tab delays:

`TAB0` Horizontal-tab delay type 0.

TAB1 Horizontal-tab delay type 1.
 TAB2 Horizontal-tab delay type 2.
 TAB3 Expand tabs to spaces.

BSDLY Select backspace delays:
 BS0 Backspace-delay type 0.
 BS1 Backspace-delay type 1.

VTDLY Select vertical-tab delays:
 VT0 Vertical-tab delay type 0.
 VT1 Vertical-tab delay type 1.

FFDLY Select form-feed delays:
 FF0 Form-feed delay type 0.
 FF1 Form-feed delay type 1.

Baud Rate Selection

The input and output baud rates are stored in the `termios` structure. These are the valid values for objects of type `speed_t`. The following values are defined, but not all baud rates need be supported by the underlying hardware.

B0 Hang up
 B50 50 baud
 B75 75 baud
 B110 110 baud
 B134 134.5 baud
 B150 150 baud
 B200 200 baud
 B300 300 baud
 B600 600 baud
 B1200 1200 baud
 B1800 1800 baud
 B2400 2400 baud
 B4800 4800 baud
 B9600 9600 baud
 B19200 19200 baud
 B38400 38400 baud

Control Modes

The `c_cflag` field describes the hardware control of the terminal; not all values specified are required to be supported by the underlying hardware:

CSIZE Character size:
 CS5 5bits.
 CS6 6bits.
 CS7 7bits.
 CS8 8bits.

CSTOPB
 Send two stop bits, else one.
CREAD Enable receiver.
PARENB
 Parity enable.
PARODD
 Odd parity, else even.
HUPCL Hang up on last close.
CLOCAL
 Ignore modem status lines.

Local Modes

The `c_lflag` field of the argument structure is used to control various terminal functions:

ECHO Enable echo.
ECHOE Echo erase character as error-correcting backspace.
ECHOK Echo KILL.
ECHONL
 Echo NL.
ICANON
 Canonical input (erase and kill processing).
IEXTEN
 Enable extended input character processing.
ISIG Enable signals.
NOFLSH
 Disable flush after interrupt or quit.
TOSTOP
 Send `SIGTTOU` for background output.
XCASE Canonical upper/lower presentation.

Attribute Selection

The following symbolic constants for use with `tcsetattr()` are defined:

TCSANOW
 Change attributes immediately.
TCSADRAIN
 Change attributes when output has drained.
TCSAFLUSH
 Change attributes when output has drained; also flush pending input.

Line Control

The following symbolic constants for use with `tcflush()` are defined:

TCIFLUSH
 Flush pending input.
TCOFLUSH
 Flush untransmitted output.
TCIOFLUSH
 Flush both pending input and untransmitted output.

The following symbolic constants for use with `tcflow()` are defined:

TCIOFF
 Transmit a STOP character, intended to suspend input data.

TCION Transmit a START character, intended to restart input data.

TCOOFF

Suspend output.

TCOON Restart output.

The following are declared as functions and may also be defined as macros:

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

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

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

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

```
int tcdrain(int fildes);
```

```
int tcflow(int fildes, int action);
```

```
int tcflush(int fildes, int queue_selector);
```

```
int tcgetattr(int fildes, struct termios *termios_p);
```

```
pid_t
```

```
tcgetsid(int fildes);
```

```
int tcsendbreak(int fildes, int duration);
```

```
int tcsetattr(int fildes, int optional_actions,
              struct termios *termios_p);
```

APPLICATION USAGE

The following names are commonly used as extensions to the above. They are therefore reserved and portable applications should not use them.

CBAUD	EXTB	VDSUSP
DEFECHO	FLUSHO	VLNEXT
ECHOCTL	LOBLK	VREPRINT
ECHOKE	PENDIN	VSTATUS
ECHOPRT	SWTCH	VWERASE
EXTA	VDISCARD	

SEE ALSO

termios(2).

NAME

time - time types

SYNOPSIS

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

DESCRIPTION

The `<time.h>` header declares the structure `tm`, which includes at least the following members:

```
int    tm_sec
        Seconds [0,61]
int    tm_min
        Minutes [0,59]
int    tm_hour
        Hour [0,23]
int    tm_mday
        Day of month [1,31]
int    tm_mon
        Month of year [0,11]
int    tm_year
        Years since 1900
int    tm_wday
        Day of week [0,6] (Sunday = 0)
int    tm_yday
        Day of year [0,365]
int    tm_isdst
        Daylight savings flag
```

The value of `tm_isdst` is positive if Daylight Saving Time is in effect, 0 if Daylight Saving Time is not in effect, and negative if the information is not available.

This header defines the following symbolic names:

`NULL` Null pointer constant.

`CLK_TCK`
Number of clock ticks per second returned by the `times()` function.

`CLOCKS_PER_SEC`
A number used to convert the value returned by the `clock()` function into seconds.

The `clock_t`, `size_t` and `time_t` types are defined as described in `<sys/types.h>` [see [types\(5\)](#)].

Although the value of `CLOCKS_PER_SEC` is required to be 1 million on all XSI-conformant systems (XSI = X/Open System Interface), it may be variable on other systems and it should not be assumed that `CLOCKS_PER_SEC` is a compile-time constant.

The value of `CLK_TCK` is currently the same as the value of `sysconf(_SC_CLK_TCK)`; however, new applications should call `sysconf()` because the `CLK_TCK` macro will be withdrawn.

The `<time.h>` header provides a declaration for `getdate_err`.

The following are declared as functions and may also be defined as macros:

```
char    *asctime(const struct tm *timeptr);
char    *asctime_r(const struct tm *tm, char *buf);
clock_t clock(void);
char    *ctime(const time_t *clock);
```

```

char    *ctime_r(const time_t *clock, char *buf);
double   difftime(time_t time1, time_t time0);
struct tm *getdate(const char *string);
struct tm *gmtime(const time_t *timer);
struct tm *gmtime_r(const time_t *clock, struct tm *result);
struct tm *localtime(const time_t *timer);
struct tm *localtime_r(const time_t *clock, struct tm *result);
time_t   mktime(struct tm *timeptr);
size_t   strftime(char *s, size_t maxsize, const char *format,
                  const struct tm *timptr);
char     *strptime(const char *buf, const char *format,
                  struct tm *tm);
time_t   time(time_t *tloc);
void     tzset(void);

```

The following are declared as variables:

```

extern int      daylight;
extern long int timezone;
extern char     *tzname[];

```

APPLICATION USAGE

The range [0,61] for `tm_sec` allows for the occasional leap second or double leap second.

`tm_year` is a signed value, therefore years before 1900 may be represented.

SEE ALSO

`utime(2)`, `time(2)`, `asctime(3C)`, `clock(3C)`, `ctime(3C)`, `daylight(3C)`, `difftime(3C)`, `getdate(3C)`, `gmtime(3C)`, `localtime(3C)`, `mktime(3C)`, `strftime(3C)`, `strptime(3C)`, `sysconf(3C)`, `timezone(3C)`, `tzname(3C)`, `tzset(3C)`, `sys_time(5)`.

NAME

timeb - additional definitions for date and time

SYNOPSIS

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

DESCRIPTION

The `<sys/timeb.h>` header defines the `timeb` structure that includes at least the following members:

`time_t` `time`

 The seconds portion of the current time

`unsigned short` `millitm`

 The milliseconds portion of the current time

`short` `timezone`

 The local timezone in minutes west of Greenwich

`short` `dstflag`

 TRUE if Daylight Savings Time is in effect

The `time_t` type is defined as described in `<sys/types.h>`.

The header `<sys/timeb.h>` declares the following as a function which may also be defined as a macro:

```
int ftime(struct timeb *tp);
```

SEE ALSO

[ftime\(3C\)](#), [time\(5\)](#).

NAME

times - file access and modification times structure

SYNOPSIS

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

DESCRIPTION

The `<sys/times.h>` header defines the structure `tms`, which is returned by `times()` and includes at least the following members:

```
clock_t tms_utime
```

User CPU time

```
clock_t tms_stime
```

System CPU time

```
clock_t tms_cutime
```

User CPU time of terminated child processes

```
clock_t tms_cstime
```

System CPU time of terminated child processes

The `clock_t` type is defined as described in `<sys/types.h>`.

The following is declared as a function and may also be defined as a macro:

```
clock_t times(struct tms *buffer);
```

SEE ALSO

[times\(2\)](#), [types\(5\)](#).

NAME

types - primitive system data types

SYNOPSIS

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

DESCRIPTION

The header file `sys/types.h` contains definitions for the following data types:

`blkcnt_t`

Used for file block counts.

`blksize_t`

Used for block sizes.

`clock_t`

Used for system time in clock ticks or `CLOCKS_PER_SEC` [see [time\(5\)](#)].

`dev_t`

Used for device IDs.

`fsblkcnt_t`

Used for file system block counts.

`fsfilcnt_t`

Used for file system file counts.

`gid_t`

Used for group IDs.

`id_t` Used for as a general identifier; can be used to contain at least a `pid_t`, `uid_t` or a `gid_t`.

`ino_t`

Used for file serial numbers.

`key_t`

Used for interprocess communication.

`mode_t`

Used for file attributes.

`nlink_t`

Used for link counts.

`off_t`

Used for file sizes.

`pid_t`

Used for process IDs and process group IDs.

`pthread_attr_t`

Used to identify a thread attribute object.

`pthread_cond_t`

Used for condition variables.

`pthread_condattr_t`

Used to identify a condition attribute object.

`pthread_key_t`

Used for thread-specific data keys.

`pthread_mutex_t`

Used for mutexes.

`pthread_mutexattr_t`

Used to identify a mutex attribute object.

`pthread_once_t`
Used for dynamic package initialisation.

`pthread_rwlock_t`
Used for read-write locks.

`pthread_rwlockattr_t`
Used for read-write lock attributes.

`pthread_t`
Used to identify a thread.

`size_t`
Used for sizes of objects.

`ssize_t`
Used for a count of bytes or an error indication.

`suseconds_t`
Used for time in microseconds

`time_t`
Used for time in seconds.

`uid_t`
Used for user IDs.

`useconds_t`
Used for time in microseconds.

`ushort_t`
Size of an unsigned short value.

`ulong_t`
Size of an unsigned long value.

All of the types are defined as arithmetic types of an appropriate length, with the following exceptions: `key_t`, `pthread_attr_t`, `pthread_cond_t`, `pthread_condattr_t`, `pthread_key_t`, `pthread_mutex_t`, `pthread_mutexattr_t`, `pthread_once_t`, `pthread_rwlock_t` and `pthread_rwlockattr_t`.

Additionally, `blkcnt_t` and `off_t` are extended signed integral types, `fsblkcnt_t`, `fsfilcnt_t` and `ino_t` are defined as extended unsigned integral types, `size_t` is an unsigned integral type, and `blksize_t`, `pid_t` and `ssize_t` are signed integral types. The type `ssize_t` is capable of storing values at least in the range $[-1, \text{SSIZE\_MAX}]$. The type `useconds_t` is an unsigned integral type capable of storing values at least in the range $[0, 1,000,000]$. The type `suseconds_t` is a signed integral type capable of storing values at least in the range $[-1, 1,000,000]$.

There are no defined comparison or assignment operators for the types `pthread_attr_t`, `pthread_cond_t`, `pthread_condattr_t`, `pthread_mutex_t`, `pthread_mutexattr_t`, `pthread_rwlock_t` and `pthread_rwlockattr_t`.

The following data types are defined through typedef:

`blkcnt64_t`
Used for file block counts.

`fsblkcnt64_t`
Used for file system block counts.

`fsfilcnt64_t`
Used for file system file counts.

`ino64_t`
Used for file serial numbers.

`off64_t`
Used for file sizes.

The types `blkcnt64_t` and `off64_t` are defined as extended signed integral types.

The types `fsblkcnt64_t`, `fsfilcnt64_t`, and `ino64_t` are defined as extended unsigned integral types.

NOTES

It is not required that `ino64_t` be a 64-bit type. However, the NFS version 3 protocol allows for 64-bit file serial numbers. For NFS interoperability with systems making use of 64-bit file serial numbers, 64-bit `ino_t` support is necessary. DCE also may make use of 64-bit file serial numbers.

SEE ALSO

`chmod(2)`, `chown(2)`, `creat(2)`, `fcntl(2)`, `getgroups(2)`, `getpid(2)`, `getuid(2)`, `kill(2)`, `lseek(2)`, `mkdir(2)`, `msgctl(2)`, `msgget(2)`, `msgop(2)`, `open(2)`, `semctl(2)`, `semget(2)`, `semop(2)`, `setpgid(2)`, `setsid(2)`, `setuid(2)`, `shmctl(2)`, `shmget(2)`, `shmop(2)`, `stat(2)`, `termios(2)`, `umask(2)`, `utime(2)`, `waitid(2)`, `getpriority(3)`, `bsearch(3C)`, `mkfifo(3C)`, `readdir(3C)`, `tcsetpgrp(3C)`, `fs(4)`, `lfs(5)`.

NAME

ucontext - user context

SYNOPSIS

```
#include <ucontext.h>
```

DESCRIPTION

The `<ucontext.h>` header defines the `mcontext_t` type through `typedef`.

The `<ucontext.h>` header defines the `ucontext_t` type as a structure that includes at least the following members:

<code>ucontext_t *uc_link</code>	Pointer to the context that will be resumed when this context returns.
<code>sigset_t uc_sigmask</code>	The set of signals that are blocked when this context is active.
<code>stack_t uc_stack</code>	The stack used by this context.
<code>mcontext_t uc_mcontext</code>	A machine-specific representation of the saved context.

The types `sigset_t` and `stack_t` are defined as in `<signal.h>`.

The following are declared as functions and may also be defined as macros:

```
int getcontext(ucontext_t *ucp);  
int setcontext(const ucontext_t *ucp);  
void makecontext(ucontext_t *ucp, (void *func)(), int argc, ...);  
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

SEE ALSO

[getcontext\(2\)](#), [sigaction\(2\)](#), [sigprocmask\(2\)](#), [sigaltstack\(2\)](#), [makecontext\(3C\)](#), [signal\(5\)](#).

NAME

uio - definitions for vector I/O operations

SYNOPSIS

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

DESCRIPTION

The `<sys/uio.h>` header defines the `iovec` structure that includes at least the following members:

```
void    *iov_base
```

Base address of a memory region for input or output.

```
size_t  iov_len
```

The size of the memory pointed to by `iov_base`.

The following are declared as functions and may also be defined as macros:

```
ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
```

```
ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

SEE ALSO

`read(2)`, `write(2)`.

NAME

`ulimit` - `ulimit` commands

SYNOPSIS

```
#include <ulimit.h>
```

DESCRIPTION

The `<ulimit.h>` header defines the symbolic constants used in the `ulimit()` function.

Symbolic constants:

`UL_GETFSIZE`

Get maximum file size.

`UL_SETFSIZE`

Set maximum file size.

The following is declared as a function and may also be defined as a macro:

```
long int ulimit (int cmd, ...);
```

SEE ALSO

`ulimit(2)`.

NAME

utime - access and modification times structure

SYNOPSIS

```
#include <utime.h>
```

DESCRIPTION

The <utime.h> header declares the structure `utimbuf`, which includes the following members:

```
time_t  actime
```

Access time

```
time_t  modtime
```

Modification time

The times are measured in seconds since the Epoch.

The type `time_t` is defined as described in <sys/types.h>.

The following is declared as a function and may also be defined as a macro:

```
int utime(const char *path, const struct utimbuf *times);
```

SEE ALSO

[utime\(2\)](#), [types\(5\)](#).

NAME

utsname - system name structure

SYNOPSIS

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

DESCRIPTION

The `<sys/utsname.h>` header defines structure `utsname`, which includes at least the following members:

char sysname[]

Name of this implementation of the operating system.

char nodename[]

Name of this node within an implementation-dependent communications network.

char release[]

Current release level of this implementation.

char version[]

Current version level of this release.

char machine[]

Name of the hardware type on which the system is running.

The character arrays are of unspecified size, but the data stored in them is terminated by a null byte.

The following is declared as a function and may also be defined as a macro:

```
int uname (struct utsname *name);
```

SEE ALSO

[uname\(2\)](#).

NAME

values - machine-dependent values

SYNOPSIS

```
#include <values.h>
```

DESCRIPTION

This file contains a set of manifest constants, conditionally defined for particular processor architectures. The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit.

BITS (*type*)

The number of bits in a specified type (e.g. `int`).

HIBITS

The value of a `short` integer with only the high-order bit set.

HIBITL

The value of a `long` integer with only the high-order bit set.

HIBITI

The value of an integer with only the high-order bit set.

MAXSHORT

The maximum value of a signed `short` integer.

MAXLONG

The maximum value of a signed `long` integer.

MAXINT

The maximum value of a signed integer.

MAXFLOATLN_MAXFLOAT

The maximum value of a single-precision floating-point number, and its natural logarithm.

MAXDOUBLE, LN_MAXDOUBL

The maximum value of a double-precision floating-point number, and its natural logarithm.

MINFLOAT, LN_MINFLOAT

The smallest positive value of a single-precision floating-point number, and its natural logarithm.

MINDOUBLE, LN_MINDOUBLE

The minimum positive value of a double-precision floating-point number, and its natural logarithm.

FSIGNIF

The number of significant bits in the mantissa of a single-precision floating-point number.

DSIGNIF

The number of significant bits in the mantissa of a double-precision floating-point number.

BITSPERBYTE

The number of bits in a byte

_EXPBASE

The base of the exponent

DMAEXP

The maximum exponent of a double value

FMAEXP

The maximum exponent of a float value

DMINEXP

The minimum exponent of a double value

FMINEXP

The minimum exponent of a `float` value

`DMAXPOWTWO`

The maximum power of two representable exactly as `double`

`FMAXPOWTWO`

The maximum power of two representable exactly as `float`

`_IEEE`

1, when IEEE standard representation is used

`_DEXPLEN`

The number of bits for the exponent of a `double`

`_FEXPLEN`

The number of bits for the exponent of a `float`

`_HIDDENBIT`

1, when the significant bit of the mantissa is implicit

SEE ALSO

`limits(4)`, `math(5)`.

NAME

varargs - handle variable argument list

SYNOPSIS

```
#include <varargs.h>
va_alist
va_dcl
va_list pvar;
void va_start(va_list pvar);
type va_arg(va_list pvar, type);
void va_end(va_list pvar);
```

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists [such as `printf(3S)`] but do not use `varargs.h` are inherently non-portable, as different machines use different argument-passing conventions.

The following data types are defined in `varargs.h`:

`va_alist`
is used as the parameter list in a function header.

`va_dcl`
is a declaration for `va_alist`. No semicolon may follow `va_dcl`.

`va_list`
is a type defined for the variable used to traverse the list.

The following macros are contained in `varargs.h`:

`va_start()`
is called to initialize `pvar` to the beginning of the list.

`va_arg()`
returns the next argument in the list pointed to by `pvar`. `type` is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

`va_end()`
is used to clean up.

Multiple traversals, each bracketed by `va_start()` and `va_end()`, are possible.

EXAMPLE

This example is a possible implementation of `execl` [see `exec(2)`].

```
#include <unistd.h>
#include <varargs.h>
#define MAXARGS 100
/* execl is called by execl(file, arg1, arg2, ..., (char *)0); */
execl(va_alist)
va_dcl
{
    va_list ap;
    char *file;
    char *args[MAXARGS];          /* assumed big enough */
    int argno = 0;
    va_start(ap);
    file = va_arg(ap, char *);
```

```
    while ((args[argno++] = va_arg(ap, char *)) != 0)
    ;
    va_end(ap);
    return execv(file, args);
}
```

NOTES

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame. For example, `execl()` is passed a null pointer to signal the end of the list. `printf()` can tell how many arguments are there by the format.

It is non-portable to specify a second argument of `char`, `short`, or `float` to `va_arg()`, since arguments seen by the called function are not `char`, `short`, or `float`. C converts `char` and `short` arguments to `int` and converts `float` arguments to `double` before passing them to a function.

`varargs.h` is provided for reasons of compatibility with earlier versions. You are recommended to use `stdarg.h` instead.

SEE ALSO

`exec(2)`, `printf(3S)`, `vprintf(3S)`, `stdarg(5)`.

NAME

`wait` - wait status

SYNOPSIS

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

DESCRIPTION

When a process waits for status from its children via either the `wait()` or `waitpid()` function, the status returned may be evaluated with the following macros, defined in `sys/wait.h`. These macros evaluate to integral expressions. The *stat* argument to these macros is the integer value returned from `wait()` or `waitpid()`.

WIFEXITED (*stat*)

Evaluates to a non-zero value if status was returned for a child process that terminated normally.

WEXITSTATUS (*stat*)

If the value of **WIFEXITED** (*stat*) is non-zero, this macro evaluates to the exit code that the child process passed to `_exit` or `exit`, or the value that the child process returned from `main`.

WIFSIGNALED (*stat*)

Evaluates to a non-zero value if status was returned for a child process that terminated due to the receipt of a signal.

WTERMSIG (*stat*)

If the value of **WIFSIGNALED** (*stat*) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED (*stat*)

Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG (*stat*)

If the value of **WIFSTOPPED** (*stat*) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

WIFCONTINUED (*stat*)

Evaluates to a non-zero value if status was returned for a child process that has continued.

WCOREUMP (*stat*)

If the value of **WIFSIGNALED** (*stat*) is non-zero, this macro evaluates to a non-zero value if a core image of the terminated child was created.

The following symbolic constants for `waitpid()` are also defined in `wait.h`:

WEXITED

Wait for processes that have exited.

WSTOPPED

Status will be returned for any child that has stopped upon receipt of a signal.

WCONTINUED

Status will be returned for any child that was stopped and has been continued.

WNOWAIT

Keep the process whose status is returned in *info*p in a waitable state.

WNOHANG

Return if no status is available.

WUNTRACED

Status of stopped child process.

The type `idtype_t` is defined as an enumeration type whose possible values include at least the following:

P_ALL

P_PID
P_PGID

The `id_t` type is defined as described in `<sys/types.h>`.

The `siginfo_t` type is defined as described in `<signal.h>`.

The `rusage` structure is defined as described in `<sys/resource.h>`.

The `pid_t` type is defined as described in `<sys/types.h>`.

Inclusion of the `<sys/wait.h>` header may also make visible all symbols from `<signal.h>` and `<sys/resource.h>`.

The following are declared as functions and may also be defined as macros:

```
pid_t wait(int *stat_loc);  
pid_t wait3(int *stat_loc, int options, struct rusage *resource_usage);  
int waitid(idtype_t idtype, id_t id, siginfo_t *infp, int options);  
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

SEE ALSO

`wait(2)`, `waitid(2)`, `wait3(3)`, `resource(5)`, `types(5)`.

NAME

wchar - wide character types

SYNOPSIS

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

DESCRIPTION

The <wchar.h> header defines the following data types through typedef:

wchar_t

See <stddef.h>.

wint_t

An integral type capable of storing any valid value of wchar_t, or WEOF.

wctype_t

A scalar type of a data object that can hold values which represent locale-specific character classification.

mbstate_t

An object type other than an array type that can hold the conversion state information necessary to convert between sequences of (possibly multibyte) characters and wide characters. If a code set is being used such that an mbstate_t needs to preserve more than 2 levels of reserved state, the results are unspecified.

FILE As described in <stdio.h>.

size_t

As described in <stddef.h>.

The <wchar.h> header declares the following as functions and may also define them as macros.

Function prototypes must be provided for use with an ISO C compiler.

```
wint_t  btowc(int);
int      fwprintf(FILE *, const wchar_t *, ...);
int      fwscanf(FILE *, const wchar_t *, ...);
int      iswalnum(wint_t);
int      iswalpha(wint_t);
int      iswcntrl(wint_t);
int      iswdigit(wint_t);
int      iswgraph(wint_t);
int      iswlower(wint_t);
int      iswprint(wint_t);
int      iswpunct(wint_t);
int      iswspace(wint_t);
int      iswupper(wint_t);
int      iswxdigit(wint_t);
int      iswctype(wint_t, wctype_t);
wint_t  fgetwc(FILE *);
wchar_t *fgetws(wchar_t *, int, FILE *);
wint_t  fputwc(wchar_t, FILE *);
int      fputws(const wchar_t *, FILE *);
int      fwide(FILE *, int);
wint_t  getwc(FILE *);
wint_t  getwchar(void);
size_t  mbsinit(const mbstate_t *);
size_t  mbrlen(const char *, size_t, mbstate_t *);
size_t  mbrtowc(wchar_t *, const char *, size_t, mbstate_t *);
size_t  mbsrtowcs(wchar_t *, const char **, size_t, mbstate_t *);
```

```

wint_t  putwc(wchar_t, FILE *);
wint_t  putwchar(wchar_t);
int      swprintf(wchar_t *, size_t, const wchar_t *, ...);
int      swscanf(const wchar_t *, const wchar_t *, ...);
wint_t  tolower(wint_t);
wint_t  toupper(wint_t);
wint_t  ungetwc(wint_t, FILE *);
int      vfwprintf(FILE *, const wchar_t *, va_list);
int      vwprintf(const wchar_t *, va_list);
int      vswprintf(wchar_t *, size_t, const wchar_t *, va_list);
size_t   wcrntomb(char *, wchar_t, mbstate_t *);
wchar_t  *wcscat(wchar_t *, const wchar_t *);
wchar_t  *wcschr(const wchar_t *, wchar_t);
int      wcscmp(const wchar_t *, const wchar_t *);
int      wscoll(const wchar_t *, const wchar_t *);
wchar_t  *wcscpy(wchar_t *, const wchar_t *);
size_t   wcsncpy(const wchar_t *, const wchar_t *);
size_t   wcslen(const wchar_t *);
wchar_t  *wcsncat(wchar_t *, const wchar_t *, size_t);
int      wcsncmp(const wchar_t *, const wchar_t *, size_t);
wchar_t  *wcsncpy(wchar_t *, const wchar_t *, size_t);
wchar_t  *wcsprbrk(const wchar_t *, const wchar_t *);
wchar_t  *wcsrchr(const wchar_t *, wchar_t);
size_t   wcsrtombs(char *, const wchar_t **, size_t, mbstate_t *);
size_t   wcsspncpy(const wchar_t *, const wchar_t *);
wchar_t  *wcsstr(const wchar_t *, const wchar_t *);
double   wcstod(const wchar_t *, wchar_t **);
long int  wcstol(const wchar_t *, wchar_t **, int);
unsigned long int  wcstoul(const wchar_t *, wchar_t **, int);
wchar_t  *wcswcs(const wchar_t *, const wchar_t *);
int      wcswidth(const wchar_t *, size_t);
size_t   wcsxfrm(wchar_t *, const wchar_t *, size_t);
int      wctob(wint_t);
wctype_t wctype(const char *);
int      wcwidth(wchar_t);
wchar_t  *wmemchr(const wchar_t *, wchar_t, size_t);
int      wmemcmp(const wchar_t *, const wchar_t *, size_t);
wchar_t  *wmemcpy(wchar_t *, const wchar_t *, size_t);
wchar_t  *wmemmove(wchar_t *, const wchar_t *, size_t);
wchar_t  *wmemset(wchar_t *, wchar_t, size_t);
int      wprintf(const wchar_t *, ...);
int      wscanf(const wchar_t *, ...);
size_t   wcsftime(wchar_t *, size_t, const wchar_t *, const struct tm *);
wchar_t  *wcstok(wchar_t *, const wchar_t, wchar_t **);

```

<wchar.h> defines the following macro names:

WCHAR_MAX

The maximum value representable by an object of type `wchar_t`.

WCHAR_MIN

The minimum value representable by an object of type `wchar_t`.

WEOF Constant expression of type `wint_t` that is returned by several WP functions to indicate end-of-file.

NULL As described in <stddef.h>.

The tag `tm` is declared as naming an incomplete structure type, the contents of which are described in the header <time.h>.

Inclusion of the <wchar.h> header may make visible all symbols from the headers <ctype.h>, <stdio.h>, <stdarg.h>, <stdlib.h>, <string.h>, <stddef.h> and <time.h>.

NOTES

For the `wcsftime()` and `wcstok()` functions the definition is in XPG4 mode as follows:

```
size_t  wcsftime(wchar_t *, size_t, const char *, const struct tm *);  
wchar_t *wcstok(wchar_t *, const wchar_t *);
```

SEE ALSO

`btowc(3C)`, `conv(3C)`, `fwide(3C)`, `mbchar(3C)`, `mbsinit(3C)`, `mbstring(3C)`, `memory(3C)`, `towctrans(3C)`, `wctob(3C)`, `wctrans(3C)`, `wctype(3C)`, `wstring(3C)`, `wtype(3C)`, `fwprintf(3S)`, `fwscanf(3S)`, `vprintf(3S)`, `ctype(5)`, `stdarg(5)`, `stddef(5)`, `stdio(5)`, `stdlib(5)`, `string(5)`, `time(5)`.

NAME

wctype - wide character classification and mapping utilities

SYNOPSIS

```
#include <wctype.h>
```

DESCRIPTION

The `<wctype.h>` header defines the following data types through `typedef`:

`wint_t`

As described in `<wchar.h>`.

`wctrans_t`

A scalar type that can hold values which represent locale-specific character mappings.

`wctype_t`

As described in `<wchar.h>`.

The `<wctype.h>` header declares the following as functions and may also define them as macros.

Function prototypes must be provided for use with an ISO C compiler.

```
int      iswalnum(wint_t);
int      iswalpha(wint_t);
int      iswcntrl(wint_t);
int      iswdigit(wint_t);
int      iswgraph(wint_t);
int      iswlower(wint_t);
int      iswprint(wint_t);
int      iswpunct(wint_t);
int      iswspace(wint_t);
int      iswupper(wint_t);
int      iswxdigit(wint_t);
int      iswctype(wint_t, wctype_t);
wint_t   towctrans(wint_t, wctrans_t);
wint_t   tolower(wint_t);
wint_t   toupper(wint_t);
wctrans_t wctrans(const char *);
wctype_t wctype(const char *);
```

`<wctype.h>` defines the following macro names:

WEOF Constant expression of type `wint_t` that is returned by several MSE functions to indicate end-of-file.

For all functions described in this header that accept an argument of type `wint_t`, the value will be representable as a `wchar_t` or will equal the value of **WEOF**. If this argument has any other value, the behavior is undefined.

The behavior of these functions is affected by the `LC_CTYPE` category of the current locale.

Inclusion of the `<wctype.h>` header may make visible all symbols from the headers `<ctype.h>`, `<stdio.h>`, `<stdarg.h>`, `<stdlib.h>`, `<string.h>`, `<stddef.h>`, `<time.h>` and `<wchar.h>`.

SEE ALSO

[conv\(3C\)](#), [wctrans\(3C\)](#), [wctype\(3C\)](#), [wtype\(3C\)](#), [locale\(5\)](#), [wchar\(5\)](#).

NAME

wordexp - word-expansion types

SYNOPSIS

```
#include <wordexp.h>
```

DESCRIPTION

The `wordexp.h` header defines the structures and symbolic constants used by the `wordexp()` and `wordfree()` functions.

The structure type `wordexp_t` contains 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 we_wordv.
```

The *flags* argument to the `wordexp()` function is the bitwise inclusive OR of the following flags:

```
WRDE_APPEND
    Append words to those previously generated.

WRDE_DOOFS
    Number of null points to prepend to we_wordv.

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 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_NOSYS
    The implementation does not support the function.

WRDE_SYNTAX
    Shell syntax error, such as unbalanced parentheses or unterminated string.
```

The following are declared as functions and may also be declared as macros:

```
int  wordexp(const char *words, wordexp_t *pwordexp, int flags);  
void wordfree(wordexp_t *pwordexp);
```

The implementation may define additional macros or constants using names beginning with `WRDE_`.

SEE ALSO

[wordexp\(3C\)](#).

NAME

wstat - wait status

SYNOPSIS

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

DESCRIPTION

When a process waits for status from its children through either the `wait` or `waitpid` function, the status returned may be evaluated with the following macros, defined in `sys/wait.h`. These macros evaluate to integral expressions. The *stat* argument to these macros is the integer value returned from `wait` or `waitpid`.

`WIFEXITED (stat)`

Evaluates to a non-zero value if status was returned for a child process that terminated normally.

`WEXITSTATUS (stat)`

If the value of `WIFEXITED (stat)` is non-zero, this macro evaluates to the exit code that the child process passed to `_exit` or `exit`, or the value that the child process returned from `main`.

`WIFSIGNALED (stat)`

Evaluates to a non-zero value if status was returned for a child process that terminated due to the receipt of a signal.

`WTERMSIG (stat)`

If the value of `WIFSIGNALED (stat)` is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

`WIFSTOPPED (stat)`

Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

`WSTOPSIG (stat)`

If the value of `WIFSTOPPED (stat)` is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

`WIFCONTINUED (stat)`

Evaluates to a non-zero value if status was returned for a child process that has continued.

`WCOREDUMP (stat)`

If the value of `WIFSIGNALED (stat)` is non-zero, this macro evaluates to a non-zero value if a core image of the terminated child was created.

SEE ALSO

`exit(2)`, `wait(2)`, `waitpid(2)`.